BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF INFORMATION SYSTEMS
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

VPN CONTROLLER
VPN KONTROLÉR

MASTER'S THESIS
DIPLOMOVÁ PRÁCE

AUTHOR **ONDŘEJ FABIÁNEK**
AUTOR PRÁCE

SUPERVISOR **Ing. MATĚJ GRÉGR, Ph.D.**
VEDOUCÍ PRÁCE

BRNO 2019

Brno University of Technology
Faculty of Information Technology

Department of Information Systems (DIFS) Academic year 2018/2019

Master's Thesis Specification

21892

Student: **Fabiánek Ondřej, Bc.**
Programme: Information Technology Field of study: Computer Networks and Communication
Title: **VPN Controller**
Category: Networking
Assignment:

1. Study current approaches for managing networking nodes.
2. Design a controller for managing a large number of VPN routers. The controller should support at least the following features: authentication, management of router's networks, clustering routers to different groups, monitoring routers health and configuring custom filtering rules.
3. Implement the controller and evaluate required functionality.
4. Analyze results and discuss possible extensions.

Recommended literature:

- RFC 7018: Auto-Discovery VPN Problem Statement and Requirements. V. Manral, S. Hanna. September 2013. DOI: 10.17487/RFC7018)
- C. Adjih *et al.*, "FIT IoT-LAB: A large scale open experimental IoT testbed," *2015 IEEE 2nd World Forum on Internet of Things (WF-IoT)*, Milan, 2015, pp. 459-464.doi: 10.1109/WF-IoT.2015.7389098
- T. Huang, F. R. Yu, C. Zhang, J. Liu, J. Zhang and Y. Liu, "A Survey on Large-Scale Software Defined Networking (SDN) Testbeds: Approaches and Challenges," in *IEEE Communications Surveys & Tutorials*, vol. 19, no. 2, pp. 891-917, Secondquarter 2017

Requirements for the semestral defence:

- Items 1 and 2.

Detailed formal requirements can be found at http://www.fit.vutbr.cz/info/szz/

Supervisor: **Grégr Matěj, Ing., Ph.D.**
Head of Department: Kolář Dušan, doc. Dr. Ing.
Beginning of work: November 1, 2018
Submission deadline: May 22, 2019
Approval date: October 31, 2018

Abstract

The subject of this diploma thesis is to design architecture and develop implementation of a flexible, scalable and secure system for managing virtual private networks, which would provide networking of otherwise possibly disconnected routers and their respective local networks. While the project is primarily focused around routers of Advantech manufacturer, support of additional types of devices may be added later on.

Abstrakt

Tato práce se zabývá návrhem architektury a implementací flexibilního, škálovatelného a bezpečného systému pro správu virtuálních privátních sítích, který by umožnil propojení jinak nedostupných routerů a zařízení v jejich lokálních sítích. Ačkoli je systém primárně určen pro použití s routery od výrobce Advantech, podpora jiných zařízení může být později přidána.

Keywords

VPN, networking, routing, configuration

Klíčová slova

VPN, sítě, směrování, konfigurace

Reference

FABIÁNEK, Ondřej. *VPN Controller*. Brno, 2019. Master's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Ing. Matěj Grégr, Ph.D.

Rozšířený abstrakt

Tato diplomová práce popisuje návrh, implementaci a zhodnocení systému pro propojování směrovačů a zařízení za nimi do dynamicky konfigurovatelných virtuálních sítí. Pro tyto účely byly nastudovány existující nástroje pro konfiguraci vzdálených prvků a technologie pro vytváření VPN sítí. Pozornost byla též věnována podobnému již existujícímu řešení s názvem SmartCluster.

Systém byl navržen jako hvězdicová topologie, kde jednotlivé směrovače navazují spojení s centrálním prvkem, který určuje, kdo může komunikovat s kým. Pro jednotlivé směrovače byla vyvinuta aplikace ve formě tzv. uživatelského modulu, což je speciální balíček, určený pro vkládání rozšiřujícího softwaru do routerů od výrobce Advantech. Tato aplikace obstarává navázání komunikace s centrálním prvkem (prostřednictvím OpenVPN) a následně zpracovává příchozí požadavky.

Jako autentizační mechanismus jsou použity X.509 certifikáty, jenž jsou distribuovány prostřednictvím další entity, zvané Dispatch server, ke které se každý centrální prvek registruje, a jejímž certifikátem disponuje každý směrovač (je součástí instalace uživatelského modulu).

Skrze centrální prvek lze směrovače dynamicky dělit do skupin a konfigurovat tak, s kým mohou komunikovat. Pro každou skupinu lze navíc vkládat firewallová pravidla, jenž budou ovlivňovat pouze komunikaci v rámci té dané skupiny.

Každý směrovač pro připojení do systému nahlásí seznam svých rozhraní a jejich aktuální konfiguraci. Tuto lze kdykoli prostřednictvím centrálního prvku měnit (mezi nastavitelné položky spadá např. konfigurace DHCP či IP adresy rozhraní). Každé rozhraní lze navíc nastavit do jednoho ze 4 následujících módů, ovlivňujícího za ním se nacházející zařízení:

- **Local** – zařízení nemají přístup do virtuální sítě a nejsou z ní adresovatelné
- **Public** – zařízení mají přístup do virtuální sítě a jsou adresovatelné jejich lokálními IP adresami.
- **1:1 NAT** – zařízení mají přístup do virtuální sítě a jsou adresovatelné pod přidělenými virtuálními adresami. Toto umožňuje, aby spolu komunikovaly prvky, které by v případě *public* módu nemohly, protože mají totožné lokální IP adresy.
- **Ignored** – zařízení nemají přístup do virtuální sítě a centrální prvek nebude sahat na konfiguraci tohoto rozhraní.

V případě změny konfigurace zařízení, jenž je momentálně offline, si systém změny pamatuje a doručí je do směrovače v momentě jeho opětovného připojení.

Pro každé nově přidané zařízení je vygenerováno URL pro přístup na jeho webové rozhraní skrze existující tunel, prostřednictvím centrální entity, fungující jako proxy server. Tohoto je docíleno skrze konfigurační soubory webového serveru.

Ke všem zařízení je vedeno množství statistik, zahrnující mimo jiné údaj o množství procházejících dat (skrze tunel) a jeho aktuální status (zda-li je online).

Ačkoli výsledné řešení obsahuje i webové stránky (pro ovládání centrálního prvku), jejich tvorba nebyla součástí této diplomové práce, je to produkt třetí strany a nebude v této práci nijak diskutován.

Po dokončení popisovaného systému byla provedena řada zátěžových testů. Jedná se o test se 100, 300 a se 600 (simulovanými) směrovači. Pro potřeby těchto testů byla vytvořena speciální verze uživatelského modulu, jenž je spustitelná na PC. Tento upravený modul

byl následně na několika fyzických strojích spuštěn v mnoha instancích, prostřednictvím virtualizačního softwaru zvaného Docker. Z naměřených údajů vyplynulo, že systém je velmi dobře škálovatelný z hlediska zabrané paměti a procesorové zátěže. Nejslabším prvkem se ukázala být špatně optimalizovaná databáze, což při některých uživatelských akcích nad velkými počty zařízení způsobuje špatnou odezvu uživatelského rozhraní.

Do budoucna je možné systém rozšířit i o jiné typy spravovaných zařízení. Řadu aspektů systému lze dále výrazně optimalizovat a zkrátit tak trvání některých operací.

VPN Controller

Declaration

I declare that I have prepared this Master‘s thesis independently, under the supervision of Ing. Matěj Grégr, Ph.D. Further advice was provided to my by my work colleagues. I listed all of the literary sources and publications that I have used.

.....
Ondřej Fabiánek
May 21, 2019

Acknowledgements

I hereby want to thank to my supervisor and to everyone who provided me with proffesional advice.

Contents

Chapter 1

Introduction

Many industrial routers, spread all over the world, are placed in remote locations where the only way to access the internet is through a mobile connection. When a network administrator needs to access one of these routers, which are often located in busses, trains, mountains, and other places that are difficult to physically access, he faces a problem: mobile devices often do not have public IP addresses. They are hidden behind a network address translation (NAT). Rather than having to always travel to the router or paying more money for a better deal with the internet service provider, the administrator has several choices. He could use IPv6 addresses. Those are more plentiful and they are significantly cheaper. However, since not all internet providers offer IPv6 addresses and the fact that IPv6 NAT also exists, the cheapest and broadly usable solution can often be the use of a virtual private network (VPN). This not only solves the initial problem, but also brings the advantage of higher security.

The aim of this diploma thesis is to design and implement a VPN based system that would allow administrators easy access to such routers, and also provided a way to make dynamic changes to the topology of the network. Through clustering routers into groups, the administrator would be able to enable traffic flow between some routers, while denying it elsewhere, without a regard to whether the affected routers are currently online. Further, settings of each LAN interface, of every router, will be managed by a central station, and access into/from the VPN can be granted to devices behind some interfaces and denied to those behind others. Where 2 or more LANs use overlapping ranges of IP addresses, a user may simply set the relevant interfaces into a virtualized mode (on the central station), upon which they will be assigned ranges of virtual addresses, making the local devices addressable from the VPN. Those and many other features are described in greater detail in chapter 2. In chapter 3 can be found examination of similar existing systems and tools. Their shortcomings and strengths will be evaluated and their usability for our purposes discussed.

Chapter 4 lays out a solution that can be used for implementation of the desired system. Various design choices are presented there and the reasoning behind them explained. Chapter 5 then offers a closer look at a selection of problems that were encountered during implementation and to how they were resolved.

Chapter 6 is dedicated to assessment of the created system. Various measurements of performance are conducted and their results discussed.

The solution presented in this paper is tailored primarily for use with routers of Advantech manufacturer, nevertheless, every attempt will be made to make it possible to add support for other types of devices in the future.

Chapter 2

Analysis of requirements

The final system is intended to meet real life needs of a router-manufacturing company. For this reason there are several requirements on its architecture and functionality, which will be introduced in this chapter.

2.1 Architecture overview

The system architecture can be broken down into 3 main components: Customer Server (CS), which is the central hub where all the devices connect to and where all the management takes place; Dispatch Server (DS), used for licensing purposes and to help routers locate their CS; and routers with their respective LANs, who want to communicate with each other.

The system must be easily scalable and capable of handling thousands of devices at the same time. All communication between the components must be encrypted and secure against Man-in-the-middle and other types of attacks. Before any new device is allowed to participate in communication within the VPN, its identity must first be verified, so as to prevent any unauthorized and possibly malicious devices from gaining access.

Figure 2.1 shows an overview of the mentioned components. There are 3 main lines of communication where authentication is important:

- **CS to DS** - Registration of CS's current IP address.
- **DS to router** - Passing of information about the CS to which the router belongs.
- **Router to CS** - Establishment of a secure channel through which all future communication would flow.

IPv6 support will not be included in this thesis. Nevertheless it is planned to be added later on and as such it needs to be considered during all major decisions.

2.2 Customer Server

This is where the brunt of the work is done. All the traffic between connected routers (and their LANs) flows through CS and is routed from here to where it belongs. It must allow communication only between the routers that are grouped together. Each router can be in any number of groups. It must be possible to create new groups, delete them and

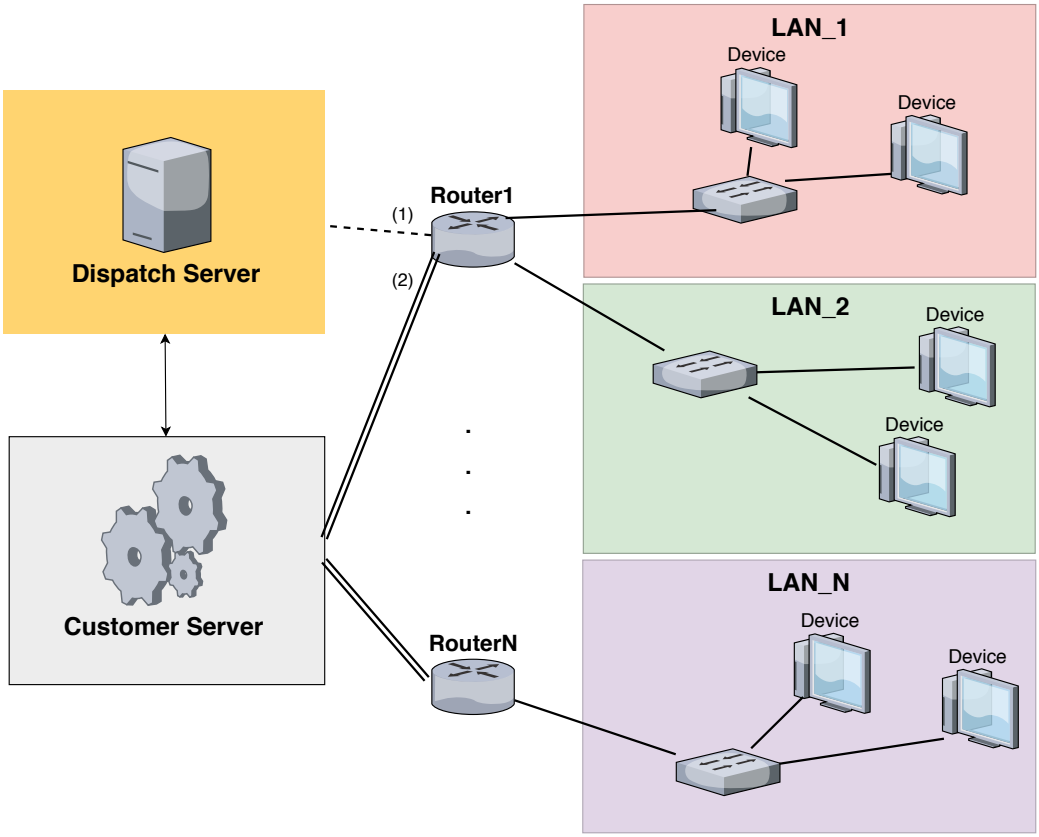

Figure 2.1: Main components of the system.

to insert/remove devices from them. CS must also support adding and deleting custom filtering rules, which would be applied on all traffic between devices in the selected group.

CS needs to monitor which routers are online and which are not. Another requirement is that it must be able to configure router's LAN settings for its non-WAN interfaces, including DHCP, IP address and netmask.

All management operations must be possible even when the connection between CS and a device is lost. In such scenario, the CS must remember the changes and deliver them to router when it becomes online. In case of any failures, the system must be able to revert to a previous state. Whenever a new router is connected, it must declare what non-WAN configured interfaces it currently has. Only those should be modifiable through CS. Each individual interface can be used in 1 of the following modes:

- public mode,
- private mode,
- 1:1 NAT mode,
- ignored mode.

Public mode is used when the LAN of which the interface is member, should be visible within the VPN. Devices that are within the same group (as the router whose interface we are referring to) can then address members of this LAN under their local IPs, and members of this LAN gain access into the VPN.

Private mode enables remote configuration for the given interface without granting its LAN access into the VPN. The controller will ensure presence of that configuration on the router and will attempt to correct any changes done locally. This feature is also present in public mode.

1:1 NAT mode solves the issue of having multiple LANs with conflicting IP addresses that need to communicate with each other. It must make devices behind such interface addressable under virtual IP addresses, instead of their local ones.

Ignored mode forbids the CS from managing the given interface.

It is required that routers' websites should be accessible from the internet via CS working as a proxy, therefore some sophisticated web server is probably needed on CS. Deleting or quarantining a device that is no longer wanted to have access into the system must also be an option.

2.3 Dispatch Server

The main desired functionality of this component is to help routers to locate their owner's CS (providing its IP address). It must also provide a way for CS to register its current IP address there.

A mechanism needs to be designed that would allow DS to recognize what CS does a given router belong to, so that it can provide him the right address. It must not be possible to trick DS into revealing address of another customer.

Unlike in case of CS, whose IP address (or domain name) can change, the DS should always be available at the same public address. Because of this, routers can have its address pre-configured and through communication with it be able to obtain current address of their CS. This allows the customer to change address of his CS without any need to go and reconfigure the routers.

2.4 Routers

The system is primarily designated for use with Advantech routers of v3 generation or newer. These devices are running a Linux distribution and offer an easy way to install additional software in the form of packages called "user modules". In theory any C/C++ written program that runs under Linux can be cross-compiled and installed there as a user module. However, there are strict storage size limitations that need to be taken into account. Total space that can be taken by user modules differs across router types, but at times can be very restrictive and it is desirable to use as little of it as possible for our purpose.

All of the routers already contain within their firmware various networking tools that we are free to use. Foremost of those is plethora of tunneling software, including *OpenVPN*, *IPsec* and *GRE*[1]. Each router can have up to 3 physical Ethernet ports. Most of them also have a WiFi module, although some do not. All of the routers have an interface for communication through a mobile network.

[1] GRE - Generic Routing Encapsulation

Chapter 3

Similar systems and existing tools

This chapter presents an overview of available tools whose use for major parts of the system had to be considered. Attention will also be given to researching systems similar to ours to identify their flaws and to try to learn from them.

3.1 Configuration management tools

The central node (Customer Server) needs to send commands to routers. These commands may range from simple changes of LAN configuration to arbitrary shell commands whose need is not yet apparent. Besides flexibility of the communication protocol, 2 more aspects need to be considered: amount of traffic it produces, and how well it can be combined with a tunneling technology (it would be detrimental to scaling to have, for example, 2 layers of encryption where 1 would suffice).

From among many existing applications for remote configuration management, two have been chosen to be described here as representatives of *push-down* (Ansible) and *pull-down* (Puppet) architecture.

3.1.1 Ansible

Ansible is a simple and yet powerful open source automation tool that can be used for configuration management, software provisioning and application deployment. Its main advantage and distinguishing feature is the use of an *agentless* architecture (it does not require a special client application to be installed on the devices that we want to manage) [19].

After setting up its configuration on the central station, it works by running easy to understand commands that can be aggregated into special scripts called *playbooks*. On their execution those are transformed into potentially more complicated, platform specific code that is executed on remote devices via SSH [32]. OpenSSH (or other implementation of it) is present in nearly all devices that could be used as routers in our system and therefore its need is inconsequential. However, to benefit from full strength of Ansible, all clients need to also have Python installed [14].

The simplicity of usage is achieved through use of so called modules. When constructing an Ansible command, the user types in the name of a specific module with a set of parameters. The complexity is hidden within implementation of that module. It can use information known about the target's platform to choose how to accomplish its task. The user types *what* needs to be done and the module takes care of *how* [23]. Ansible comes with

many different modules, including one that can be used to run arbitrary shell command on the remote machine. Most modules require Python to be installed on remote devices [14]. There is also possibility of writing a new module when necessary [16].

Each command (task) prints, upon completion, information about its status and results. To target multiple devices, there is a file containing their IP addresses. The file can be divided into named sections. To perform a task on multiple remote machines, the name of a section can be used to select a particular group of devices. It is then executed in parallel for all targets [15].

When a playbook with multiple tasks is run, it always attempts to complete the current task for all selected devices before starting another [18]. Behavior on task's failure during playbook execution is highly customizable. It can skip failed hosts in the remaining tasks, ignore the failure or abort the playbook's execution altogether. It is also possible to specify the number of retries for each task and many other parameters.

```
[root@centos7 ~]# ansible-playbook /etc/ansible/playbooks/apache.yml

PLAY [webservers] ************************************************************

GATHERING FACTS **************************************************************
ok: [192.168.0.30]
ok: [192.168.0.29]

TASK: [ensure apache is at the latest version] *******************************
ok: [192.168.0.29]
ok: [192.168.0.30]

TASK: [replace default index.html file] **************************************
ok: [192.168.0.30]
ok: [192.168.0.29]

PLAY RECAP *******************************************************************
192.168.0.29               : ok=4    changed=1    unreachable=0    failed=0
192.168.0.30               : ok=4    changed=1    unreachable=0    failed=0
```

Figure 3.1: Example of the output's format when running an ansible playbook.

Ansible supports also a limited use of other ways of communication than SSH. One example of it would be using *httpapi* plugin, which enables communication via http(s) protocol [17]. This, however, requires a client application implementing *REST* API to be present on the routers [17].

As can be seen in figure 3.1, information about client devices are gathered (this can be disabled) at the beginning of a playbook's execution. Those facts can then be referred to through variables in definitions of the tasks that follow.

3.1.2 Puppet

Puppet is an agent-based configuration management tool, available in an open-source version [24]. It requires Puppet software to be installed on all of the managed machines, called Puppet agents [24]. Each agent periodically sends data about itself to a central station (called Puppet master) and pulls down information about relevant configuration changes [11]. After applying any configuration update, the agent sends in a report informing the Puppet master about the result [11].

Configuration instructions for agents are written in Puppet's custom declarative language, heavily inspired by Ruby [24]. They are stored on the master station in files called *manifests*. Manifests support the use of variables, templates and conditional logic. There is no direct mapping between separate manifest files and Puppet agents. Rather there is a single main manifest file (or a directory of files treated as one) that governs all the agents (usually by importing contents of other manifests) [24].

Whenever any device asks for a configuration update, the manifest files are compiled into an information package called *catalog*. Unlike the manifests, a catalog contains only data relevant to a single device [24].

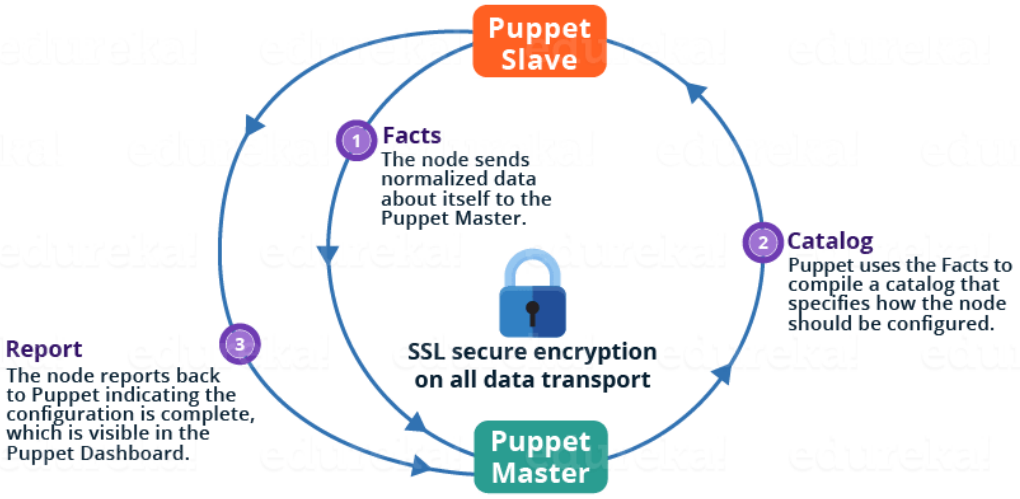

Figure 3.2: Diagram depicting the Master-Slave architecture of Puppet (reproduced from [33]).

Figure 3.3 shows an example of a simple manifest file. Notice the ability to directly specify dependencies within the manifest. There are 3 modules used: *exec*, *package* and *service*. Similarly to Ansible, many more modules are available and it is also possible to write new modules [1]. If we wanted to specify which nodes (agents) are to be affected, we could specify their names within the manifest.

Agent nodes use a tool called *Facter* (which can be used as a standalone application) to collect information about their operating system [24]. Those can be used as variables within the manifests [5]. After pulling a configuration update, the agent evaluates whether any actions needs to be taken to achieve the desired state [7]. If the machine already was configured as desired, then it does nothing [7].

All configuration related communication is done via an HTTPS protocol [11]. There does not seem to be a way to disable encryption and authentication without changes to Puppet's source code. When adding a new agent, there is a built-in support for creation and validation of its X.509 certificate. With a single command, the agent generates a key pair and sends a certificate signing request to the Puppet master [2]. Once it is there, it awaits to be manually validated by an administrator. Upon validation, the certificate is automatically signed and delivered back to the agent, who is then ready to start pulling configuration updates [2].

A Puppet's significant disadvantage is its inability to force an immediate push of configuration into the managed devices. It always has to wait for the time of a scheduled pull

request (or until someone with access to the agents forces them to pull a new configuration through a special command) [8].

```
# execute 'apt-get update'
exec { 'apt-update':                    # exec resource named 'apt-update'
  command => '/usr/bin/apt-get update'  # command this resource will run
}
# install apache2 package
package { 'apache2':
  require => Exec['apt-update'],        # require 'apt-update' before installing
  ensure => installed,
}
# ensure apache2 service is running
service { 'apache2':
  ensure => running,
}
```

Figure 3.3: Example of a *manifest* file that ensures that apache2 is installed and running

3.1.3 NETCONF

NETCONF is a network management protocol. It specifies client-server communication and the way configuration should be interacted with. It does not, in itself, define syntax for writing configuration instructions [21]. YANG language has been developed and is widely accepted for this purpose [20]. NETCONF's main strength is in support of network-wide, transactional configuration changes. It means that if some network-wide change fails on a single device, it can automatically revert the change back on all devices and thus keep their state synchronized [21].

NETCONF protocol can be divided into 4 layers, as shown in figure 3.4. The *Secure Transport layer* ensures message delivery that upholds authentication, data integrity, confidentiality, and replay protection [21]. While these properties are mandatory, NETCONF'S popular implementation in form of *libnetconf2* C library offers the option to reuse an already existing, pre-authenticated transport protocol connection by passing it its file descriptor [31]. This means that if there already is a secure tunnel, use of NETCONF will not necessarily require creation of a redundant *TLS* layer within it.

Messages layer provides a mechanism for encoding RPCs (Remote Procedure Calls) and their replies [21]. Each message is a XML document, containing a mandatory *message_id* parameter, which is used to match requests with responses [21]. Every request must contain a name of an RPC with its parameters and must later be followed by a reply, detailing information about the success and possibly also the output of that operation [21].

Operations layer defines a set of operations that can be invoked as RPC [21]. Each device needs to be running a daemon that handles them (typically by registering callback functions to a NETCONF library). Support of the following basic operations is mandatory for all devices [21]:

- get - retrieve a running configuration
- get-config - retrieve part of a specified configuration datastore

- edit-config - change parts of a specified datastore
- copy-config - copy an entire datastore into another datastore
- delete-config - delete a datastore
- lock, unlock - temporarily lock a device's datastore from changes
- close-session - request graceful termination of a session
- kill-session - force the termination of a session

The list of operations can be expanded via use of NETCONF's *capabilities* mechanism [21]. Each device can, during a session establishment, declare its capabilities. Some of those modify behavior of basic operations, others add new operations altogether. An example of this could be an *Event Notifications* capability, which adds operations for starting and canceling subscription to receiving asynchronous event notifications from a device.

Content layer represents the configuration data that are being carried (for example as a response to *get-config* operation). NETCONF does not define their format and will accept any textual configuration data [21]. The protocol does, however, define existence of one or more configuration datastores on each device [21]. Each such datastore is defined as a complete set of configuration data that is required to get device from its initial default state into a desired operational state [21]. The protocol does not specify how the datastores should be implemented. In the base model there is only 1 datastore present: *running-config*. Additional datastores may be defined through capabilities [21].

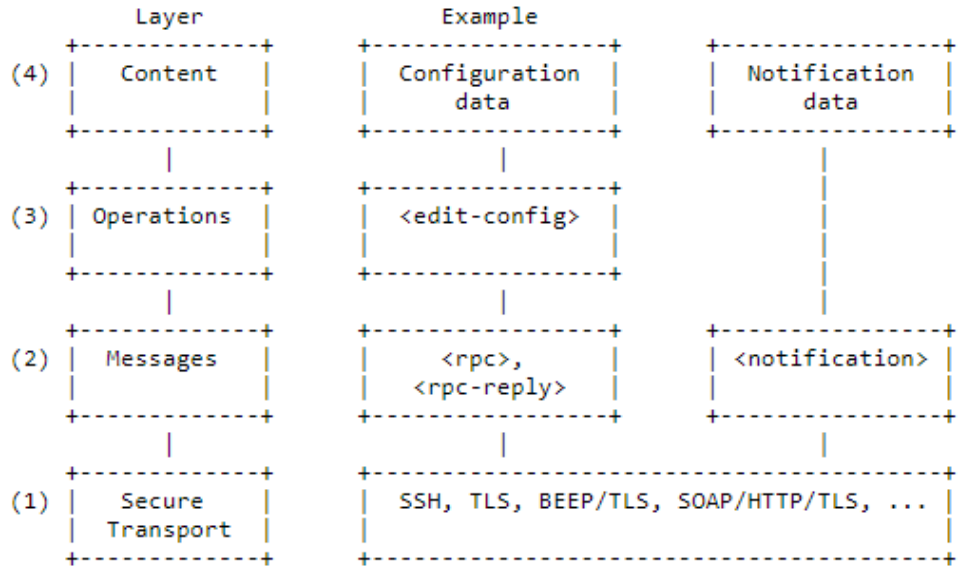

Figure 3.4: NETCONF Protocol Layers (reproduced from [21]).

3.2 Current VPN Technologies

Choosing the right underlying VPN protocol will have a great impact on many aspect of our system. For this reason, relevant characteristics of several different VPN protocols will be examined.

3.2.1 Virtual Private Network

Virtual Private Network (VPN) is a generic term that covers the use of public or private networks to create groups of devices that are separated from other network devices and that may communicate among themselves as if they were on a private network [13]. Most (although not all) VPN solutions further enhance the level of separation by providing end-to-end encryption and thus ensure confidentiality (data remains secret) and integrity (data remains unaltered) of the transported data [29]. VPN is often used for gaining access to devices that are otherwise unreachable due to existence of a firewall or NAT (Network Address Translation) on an intermediary network machine.

Various solutions may differ in their operational layer. There are 2 layers of OSI/ISO model that a VPN can operate at:

- Data Link layer (L2),
- Network layer (L3).

A VPN that works on layer L2 will encapsulate and transmit data frames. It's similar to a cable connecting two switches. The VPN has to handle all basic properties of an Ethernet network: learning MAC addresses, replicating broadcast and multicast frames, etc. Devices on both ends of a tunnel will have addresses from the same network range. These properties make L2 design potentially less scalable than L3. On the other hand, when operating on L3, each side of the connection is on a different subnet and IP packets are routed through the VPN. Broadcast, messages of ARP and other L2 protocols will generally not get to the devices on the other end of the tunnel.

Based on the network topology there are 3 types of connections [30]:

- Site-to-Site,
- Host-to-Host,
- Host-to-Site.

Not all tunneling software supports all 3 mentioned types of connection [29].

Among other metrics that need to be considered when selecting a VPN solution are:

- speed (throughput),
- security (cipher suites),
- customization options,
- supported platforms.

3.2.2 OpenVPN

OpenVPN is a highly customizable, secure and reliable VPN solution that is developed with an Open Source license [6]. It can be installed on almost all commonly used operating systems and has an excellent documentation, which makes its setup quite easy.

It can be can be configured to operate on either OSI Layer 2 or OSI Layer 3. OpenVPN provides peer authentication, data-origin authentication, data integrity, data confidentiality (encryption), and replay protection through use of X.509 certificates or a pre-shared

password. It is possible to turn the security features off and run the OpenVPN without encryption and authentication [34].

The number of available configuration options is unusually large and allows for an extensive customization [34]. Among some of its more prominent features are:

- **Management console** – Server-side interface for collecting VPN statistics, listing connected devices, kicking devices, etc.
- **Event-related scripts** – The daemon can be configured to run a given script whenever certain event happens (e.g. whenever a new device tries to connect) and its behavior can be influenced through the return values of such scripts.
- **Device-specific configuration** – The server side can have a separate configuration for each device and thus have a different set of configuration options to be used in communication with different devices.
- **Push options to clients** – The server can push any configuration options to clients and thus overwrite their local configuration when the need for it arises.

OpenVPN runs fully in user space and is comparatively slower (longer response time and lesser throughput) than most of its competitors [27].

3.2.3 IPsec

IPsec is a set of network protocols that together ensure secure communication between two endpoints. Unlike many other protocols and applications, it does not use TLS protocol for key exchange and for cryptographic algorithm negotiations [22]. Instead, it provides its own implementation of that functionality. IPsec always operates at OSI Layer 3 [22].

Based on the security needs there are 2 different transfer protocols that IPsec may be configured to use for communication:

A) **Authentication Headers (AH)** – provides data origin authentication, data integrity and protection against replay attacks, but no confidentiality [25].

B) **Encapsulating Security Payloads (ESP)** – provides data origin authentication, data integrity, protection against replay attacks, and confidentiality [26].

AH or ESP protocol is always used in conjunction with *Security Associations* (SA) protocol, which provides algorithms and key exchange mechanisms for obtaining parameters that are needed by AH and ESP [22]. Authentication is done either via certificates, or through a pre-shared key, and cannot be disabled [22].

Based on the network topology there are 2 encapsulation modes that IPsec may be configured to use for communication:

A) **Tunnel mode** – In this mode, an additional IP header is added on top of the existing one. This is typically used for *site-to-site* topology. When used in conjunction with ESP security protocol, the encapsulated IP address is encrypted and thus the real destination of the packet cannot be read by anyone while it is traveling between the sites.

B) **Transport mode** - This mode is used for *host-to-host* topology. There is no additional IP encapsulation. Irrespective of the encapsulation mode chosen, when a NAT

is on the path between the two end points, a NAT-traversal mode, which encapsulates the packet with one more UDP header, must be enabled. This is done so that the NAT can safely modify the header without changing the hash-protected IP header.

IPsec is generally considered to be faster (higher throughput, lower latency) than OpenVPN when similarly configured [27]. Its ability to encrypt destination IP address puts IPsec above OpenVPN also in terms of security.

3.2.4 SoftEther

SoftEther is an open-source, multi-protocol VPN software, optimized for high speeds. It can be run on many different platforms, including Windows, Linux, Mac, iPhone and Android [10]. The first implementation of SoftEther came out in 2013, which makes it significantly newer than many of its competitors [10]. Among its main advantages is a support for many different VPN protocols. This includes its own implementation of OpenVPN protocol (supports both L2 and L3 functionality), L2TP/IPsec, MS-SSTP, L2TPv3, EtherIP and a highly speed optimized SoftEther protocol [10]. Encapsulating all these protocols into a single application enables for more efficient use of computer resources and easier setup than if they were run all separately.

While host-to-host connections can be kept without having to install SoftEther software on the client devices (those can continue using other VPN protocols for connecting to the SoftEther server), when there is a need for a site-to-site topology, *SoftEther VPN Bridge* has to be installed on the edge node of each remote site [4]. An exception is the site where *SoftEther Server* is running (as shown in the picture 3.5).

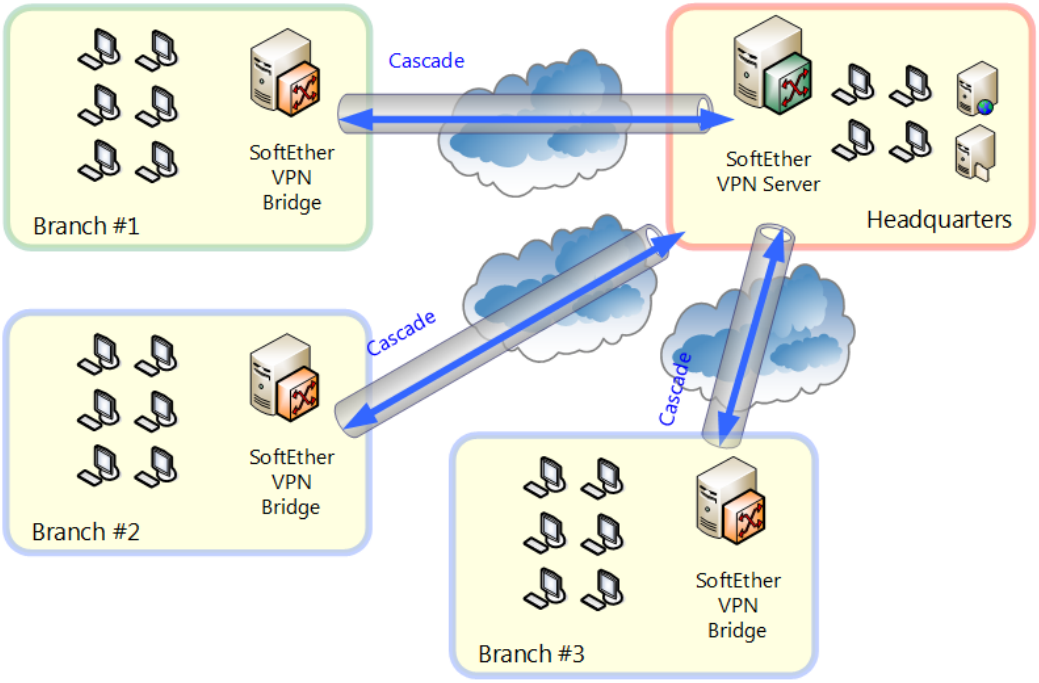

Figure 3.5: SoftEther architecture for site-to-site topology reproduced from [**?**].

SoftEther protocol operates on port 443 and camouflages itself as a HTTPS traffic. This lets it pass through most firewalls. It supports IPv6, has no issues with NAT traversal

and offers many authentication methods, including pre-shared password, X.509 certificates, RADIUS server authentication, and more [9].

According to a testing done in 2013, SoftEther's implementation of OpenVPN protocol has a higher throughput than the original implementation (approximately 15 % higher for L3, 10 % higher for L2) [3]. The same tests also revealed that the speed optimized SoftEther protocol has nearly 10 times higher throughput than OpenVPN [12].

3.3 SmartCluster

SmartCluster is an application with many similarities to the one that will be designed in this thesis. It can be used to create and manage VPN connections between groups of routers. It supports monitoring routers' health and also uses 1:1 NAT to enable communication between devices with identical IP addresses. Among its other prominent features are:

- Access to routers' web interface from browser via SmartCluster proxy.
- Support of Road-Warriors (non-router devices with access into the VPN network).

Currently only routers of a single manufacturer (*Advantech*) are supported [28].

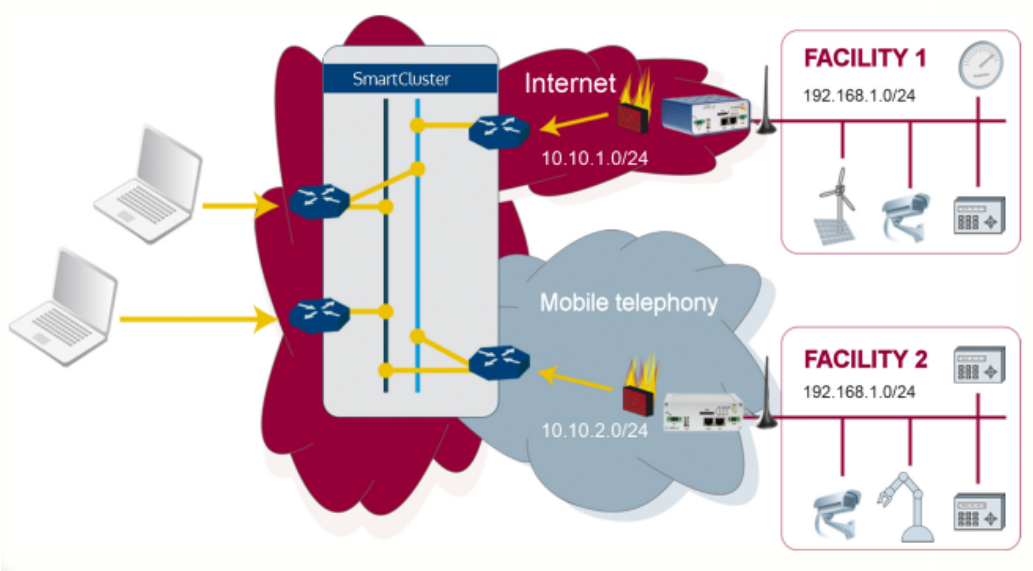

Figure 3.6: Example of SmartCluster's networking scheme (reproduced from [28]).

3.3.1 Tunneling scheme

SmartCluster uses OpenVPN to implement secure connections between routers (or Road-Warriors) and the central VPN controller [28]. It operates in *tun* mode, which encapsulates OSI Layer 3 (as opposed to *tap* mode, which would encapsulate OSI Layer 2).

During the initial setup of SmartCluster, the administrator has to specify the netmask for blocks of virtual IP addresses that will be available for individual routers [28]. If the mask is too long, there will be not enough addresses for routers with large LANs. If the mask is too short, it will put a strict limit on the number of routers that can be added to the system. It does not support the option of assigning blocks of different size to different routers [28].

3.3.2 Configuration scheme

SmartCluster does not require any special program to be installed on clients [28]. To add a new router, an administrator has to first set it up on the VPN controller's management website. After entering necessary configuration details, including router's local and virtual IP range, he can then proceed to downloading a configuration file. This file is constructed for that single router and must be manually uploaded there. It contains OpenVPN settings, together with encryption keys and short *shell* script which inserts iptables rule, thus implementing 1:1 NAT. SmartCluster is designated for use with routers whose firmware has the ability to apply configuration from a file. After the configuration is applied, an OpenVPN tunnel is created and the router gains access into the VPN.

3.3.3 Communication scheme

For each router (and Road-Warrior) the administrator can specify what other routers (and their LANs) it can communicate with [28]. Whenever these settings are modified, OpenVPN's *push* mechanism is used to update device's routing table. When this happens, tunnels with the affected devices are restarted for the change to take an effect. The re-establishment of the tunnels can take up to dozens of seconds.

All routers and devices behind them are addressable only by their assigned virtual IP addresses. To implement 1:1 NAT, it uses an iptables' *NETMAP* extension on routers. *N-th* address in one block of addresses is translated to *N-th* address of the second one.

3.3.4 Important differences

SmartCluster does not provide dynamic changes of router configuration. To apply any changes (besides routing table updates), the administrator needs to manually upload a new configuration to the router [28]. It also does not support creation of custom firewall rules and does not distinguish individual interfaces [28]. It behaves as if each router were connected to a single LAN and supported only 1:1 NAT mode.

3.4 Summary

Despite some of the configuration management tools seeming viable at first, each suffers from its own assortment of issues that make it a sub-optimal solution for our problem.

Puppet's inability to initiate a configuration push, combined with its potentially redundant encryption layer is a big problem. It is possible to use very short pull intervals, and select GRE or other tunneling method that does not provide encryption to solve the second issue, but then the non-configuration traffic would not be encrypted, which is unacceptable.

Ansible brings the problem with redundant encryption layer too. Since it uses *ssh* for all communication, it could prove very costly with high numbers of routers. It also requires Python to be installed on all of the managed routers. Overall the best solution appears to be writing a custom application for routers. Configuration changes that will need to be propagated onto routers are probably sufficiently limited in scope that the utility brought by existing robust configuration tools does not outweigh their disadvantages. Although source codes for Ansible and Puppet are publicly available and could be modified for our purpose, it would require extensive changes and make updates to new versions difficult.

Custom new application can be made very small in size, which is an important requirement. It can be installed as a user module on routers and with relatively small changes be

ported to a different type of device with Linux. Unlike Ansible or Puppet it would not add additional level of encryption. Using NETCONF implementation as a communication protocol would spare some time and remove opportunities for creation of bugs, but given the fact that a very simple communication protocol is sufficient, it would also cause unnecessary and significant increase of the module's size.

Chapter 4

System design

In this chapter, the information learned from the research of existing technologies will be used to design all component of the system.

4.1 Tunneling scheme

OpenVPN was chosen as the underlying tunneling method for its huge amount of useful features and configuration options. Its Open Source license also gives us the freedom to modify the source code if such need ever arises. Despite SoftEther implementation having higher throughput, the original implementation was chosen for its superior number of configuration features.

4.1.1 Setup

OpenVPN tunnel will be created and held active between each router and the Customer Server (star topology). There shall only be 1 tunnel interface on CS and all routers will belong into the same virtual network (visibility restrictions can be done on a different level). The tunneling will be done on OSI layer 3 (*tun mode*). To make it possible for Windows clients to be able to connect (when support for road-warriors is added), the server daemon must use *–topology subnet* parameter.

For every router, there will be a configuration file present on CS, where client-specific OpenVPN settings can be stored (enabled by *–client-config-dir* parameter).

4.1.2 Addresing

By default, each router receives a new virtual IP whenever it re-connects. We will modify this behavior by using *–nopool* parameter and storing persistent addresses into their configuration files.

Since we are using topology *subnet*, instead of specifying a single IP address for each router, we can specify a whole range of addresses. This is used to assign a block of virtual addresses to each router. They then use the first IP address from the given block for themselves and the rest is reserved for devices in their local networks (see *1:1 NAT* interface mode). For the purposes of this thesis, the number of virtual addresses available for devices behind any given router will be fixed at 254. However, there are no obstacles to implementing a more sophisticated address assignment in the future. All blocks of virtual addresses that are assigned to routers, belong to a single huge virtual network. The network address

is configurable. Figure 4.1 shows example of the address assignment for the largest possible network.

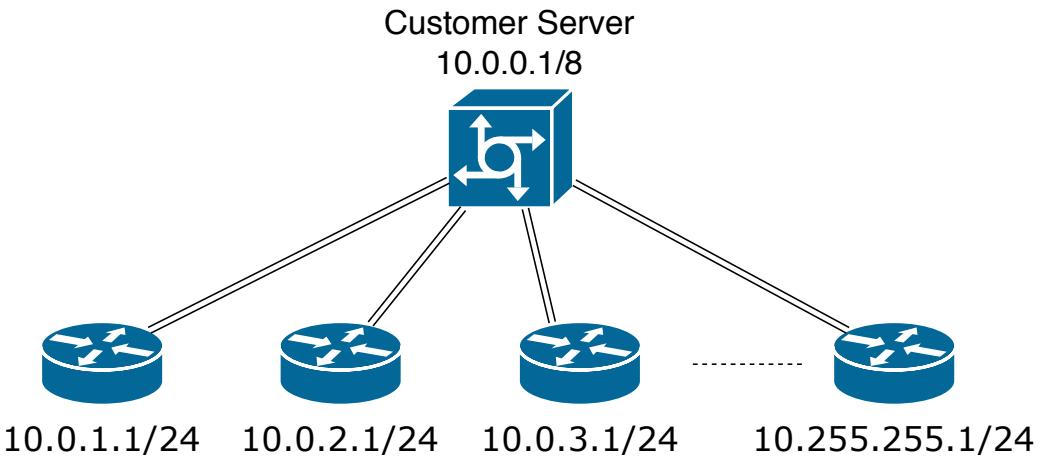

Figure 4.1: Example of a virtual address assignment for 10.0.0.0/8 network.

4.1.3 Routing

Since all routers will receive virtual IP addresses belonging into the same network, routing between them can be handled directly by OpenVPN without participation of any external application. However, to direct traffic meant for some LAN (set in *public* mode) onto a specific router, the following actions must be taken by CS:

1. Store the IP address and netmask of that LAN into the particular router's configuration file.
2. Add a static route leading into a tunnel.
3. Terminate and re-establish tunnel connection with that router.

The re-establishment of the tunnel is required in order for OpenVPN to apply the changes done in the device's configuration file. To accomplish restart of a particular tunnel connection, a device can be kicked through OpenVPN's management console. It will, however, take a long time before the device notices it and initiates a new connection. The precise timing is dependent on the interval for ping checks that OpenVPN uses to determine health of the tunnel, but can take up to over a minute. Rather than configuring smaller ping intervals, which could lead to unnecessarily many restarts (if internet connectivity is bad), this problem will be solved by replacing the kick with a control message sent to the application on router. It will then initiate a restart of the tunnel from the client's side, which works very fast (usually 3 or 4 seconds until the connection is up again).

The concept, as described so far, enables a packet that enters the tunnel on one router, to be routed through CS to its destination. Nevertheless, it is not evident that the packet would actually enter the tunnel. It is often undesirable to have a default gateway on each router leading into the tunnel. Therefore the routing table of each router needs to be managed, to only send into the tunnel those packets, whose destination is reachable through the VPN. There are 3 ways to accomplish this:

A) Manage it through OpenVPN client configuration files.

B) Use a routing protocol (OSPF, RIP, ...).

C) Manage it through a remote configuration protocol.

The first option would require a restart of the tunnel connection whenever there is a change to the routing table of some device. The consequence of this would be that after joining or leaving a group with a device that has at least one interface set into *public* mode, connection would have to be restarted with all other devices in that group (those that do not share any other group with the one who is leaving). The same effect would also be caused by any member of the group changing settings of its LAN, which is set in *public* mode. This is unacceptable.

The second option would require installation of additional software on routers and wouldn't be that much beneficial over the last option, to mitigate that disadvantage. Therefore the last option (to manage it through the configuration protocol) was chosen, as it is best suited for our purposes.

4.1.4 Online detection

OpenVPN offers a way for other applications to detect what devices (identified by common name in their certificate) are currently connected. For this purpose it can create a daemon (requires *–management* parameter to be used) that listens on *localhost* interface and responds to queries with a list of online devices. This method, however, does not suit us. The raw text returned by the console contains a lot of unnecessary information and is difficult to parse. With high numbers of routers it might prove computationally demanding to read it as often as would be necessary. There would also be a delay between a device connecting and us learning about it.

A superior solution of using *–on-connect* parameter was chosen instead. As the last step of each authentication process, OpenVPN will call a custom script, which is given information about the device through environmental variables. This script will then set the device as online in the database, and may, if needed, initiate other actions that should be done in response to it becoming online.

Similar approach was chosen for detection of terminating a connection. A custom script will be run each time a device disconnect, and set the device in the database as offline. This will be achieved by using *–on-disconnect* parameter.

4.1.5 Authentication

A certificate-based method will be used for authentication of routers. The Customer Server will, for these purposes, provide routers with an X.509 certificate, private key and CA certificate through a secure channel, before a VPN tunnel may be established.

4.2 Firewall and groups

Iptables application in combination with its extension called *ipset* was chosen for implementing a packet filter, whose task it is to prevent a network traffic flow between ungrouped devices, and applying custom firewall rules wanted by the user. A server daemon *cs-controller* will dynamically add or delete the iptables rules (and *ipset* entries) to always reflect the current configuration.

4.2.1 Structure of iptables

The iptables consist of 4 tables, each serving a different purpose and offering slightly different tools. Those are: *filter*, *nat*, *mangle* and *raw*. Since what we need is to drop misbehaving packets, all our rules will be placed into the *filter* table. The filter table contains 3 predefined main chains, called: *INPUT*, *OUTPUT*, and *FORWARD*. The *cs-controller* will only modify the FORWARD chain and leave the other two unchanged. This is done to affect just the traffic between the devices themselves, by our rules, and not the communication between them and the *cs-controller*. Only a single rule will be added there:

```
iptables -I FORWARD -i tun0 -j cs-service
```

The rule will apply our custom chain *cs-service* onto all traffic that arrives from the tunnel, before it is forwarded to the next host. Within the *cs-service* is then located the hierarchy of rules that implement the group concept and a user firewall. The entire packet filtering logic, which is implemented through iptables, is depicted in Figure 4.2.

Many modern Linux distributions have IP forwarding disabled and it may be necessary to enable it first, before the rule becomes functional. The following command ensures that:

```
echo 1  > /proc/sys/net/ipv4/ip_forward
```

4.2.2 Group management

Only the routers that are placed in the same group shall be able to communicate with each other (and with devices behind those routers). Each networking group will be represented by one rule in the *cs-service* chain. All these rules will execute jumps into a single *cs-fw* chain, if the source and destination address both belong to devices in the same group. The following symbolic rules show the structure of the *cs-service* chain in greater detail:

```
iptables -I cs-service -m conntrack --ctstate ESTABLISHED -j ACCEPT
iptables -I cs-service -m set -set <grp_1> src -set  <grp_1> dst -j cs-fw
...
iptables -I cs-service -m set -set <grp_N> src -set  <grp_N> dst -j cs-fw
iptables -I cs-service -j DROP
```

The first rule significantly reduces the computational overhead by removing traversal of most of the rules for packets belonging to already established communication streams. The last rule drops all packets that are sent between devices that do not share any group.

To implement the concept of groups without using an excessive number of rules, an extension of iptables called *ipset* was chosen. Ipset is a tool that enables large numbers of networks to be referenced from within a single iptables rule. Unlike normal iptables chains, which are stored and traversed linearly, IP sets are stored in indexed data structures, making lookups very efficient even when dealing with large sets.

The cs-controller will create and manage an IP set for each group. It will contain addresses of all routers within that group, together with addresses of those LANs that were made public. If there are any changes in group membership, only the IP set needs to be modified to implement them.

Table 4.1 shows all group operations and their respective iptables implementations. Note that these will be modified further in the chapter about customizable firewall.

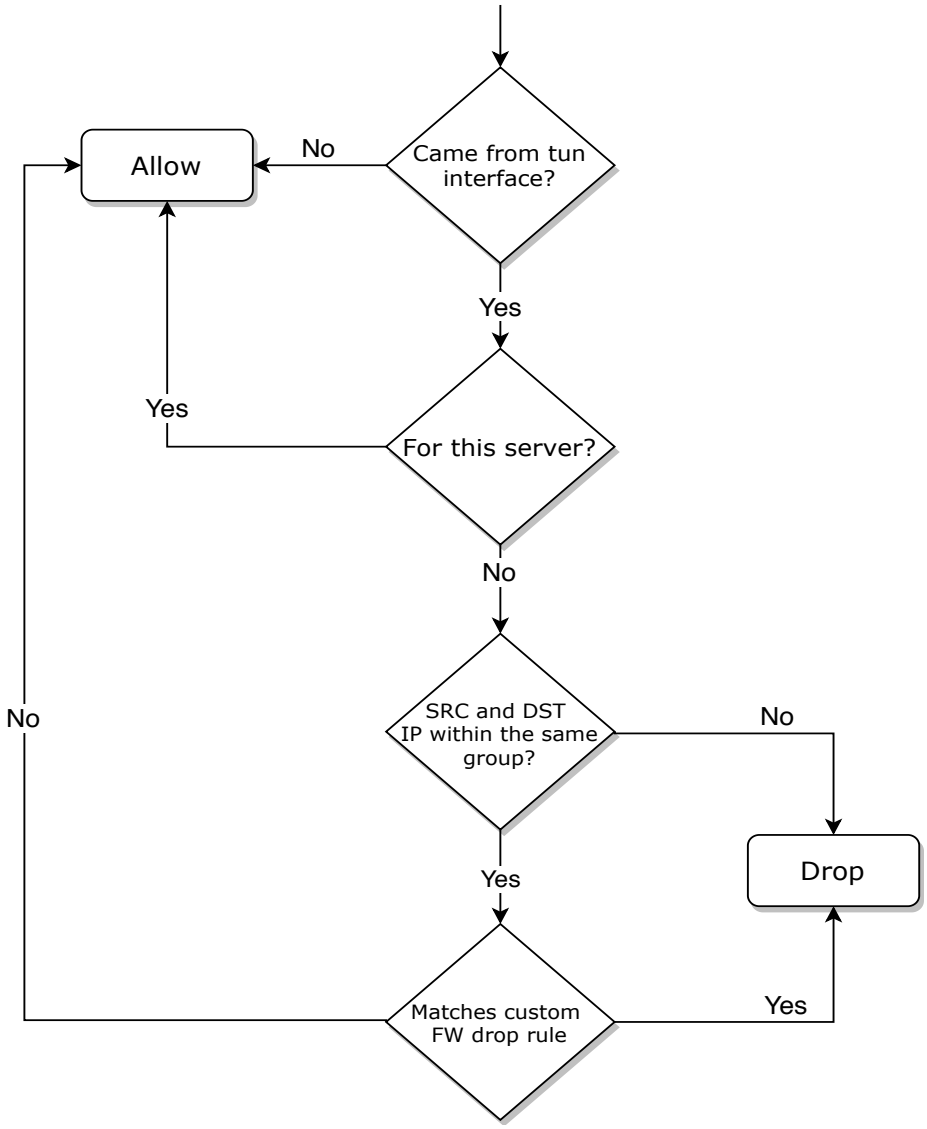

Figure 4.2: Diagram outlining the decision logic for packet filtering.

4.2.3 Custom filtering rules

A user can add rules that will be applied on traffic within some group, however he cannot use them to grant some device access into a group that it is not member of. If two devices share more than one group for which there exist custom filtering rules, rules from all such groups will be applied to the traffic between them.

To add support for creating custom filtering rules, a previously mentioned iptables' *cs-fw* chain shall be utilized. Its structure will be similar to cs-service chain:

```
iptables -I cs-fw -m set -set <grp_1> src -set  <grp_1> dst -j fw-grp_1
...
iptables -I cs-fw -m set -set <grp_N> src -set  <grp_N> dst -j fw-grp_N
iptables -I cs-fw -j ACCEPT
```

Operation	Implementation
Join group	• Update IP set of the group.
Leave group	• Update IP set of the group.
Create group	• Create an IP set. • Insert 1 rule into *cs-service* chain.
Delete group	• Remove 1 rule from *cs-service* chain. • Delete an IP set.

Table 4.1: Depiction of how the basic operations for management of groups can be implemented.

Every group shall have its own chain, where filtering rules will be stored. The only allowed action for these filters is *DROP*. If no filter within one group chain matches a packet, the execution returns into *cs-fw*, and other group chains may be traversed. Thus, filters from multiple groups may be applied if some devices share more than one group.

Note the last rule, which accepts all traffic that was not dropped by any of the custom filters. It represents the default firewall policy. By changing it to *DROP*, and custom filters to *ACCEPT*, the default policy could be switched.

An unfortunate effect of having a rule that executes *ACCEPT* on established connections in the higher level chain (*cs-service*) is that when a new filter is added, it does not affect traffic that is marked as *established* in *conntrack*. However, once all such connection expire, it will be applied without exception.

4.3 Router management

Only those router settings, which are directly tied to the functionality of the VPN controller, will be remotely managed. To implement it, a simple application (user module) shall be created and installed on every router. This application will be handling requests it receives from the Customer Server.

The control messages sent between the CS and routers will travel through an encrypted tunnel. As such, a simple TCP connection will be used for communication. At any time there will always be at most 1 TCP connection open with a particular (validated) router. This TCP connection will be closed right after a response from the router is received (or after timeout) and re-opened only when a new request needs to be delivered.

To prevent unauthorized devices from manipulating routers through the application, the daemon listening on routers will be bound to *tun* interface and compare the source address of incoming messages with the one it expects CS to have (always the first address of the virtual network).

Whenever a router's status changes from *offline* to *online*, the *cs-controller* will send him a message, containing all configuration information needed to reach the expected state. This mechanism will, to a limited extent, protect routers' configuration from unwanted changes by their local administrators.

Operation	Implementation
Create group	• Create an IP set. • Insert 1 rule into *cs-service* chain. • Create a new group chain. • Insert 1 rule into *cs-fw* chain.
Delete group	• Remove 1 rule from *cs-service* chain. • Delete an IP set. • Delete a group chain. • Delete 1 rule from *cs-fw* chain.
Add custom filter	• Insert 1 rule into the given group's chain.
Delete custom filter	• Delete 1 rule from the given group's chain.

Table 4.2: Implementation of group operations after custom filtering rules are introduced.

4.3.1 Communication protocol

JSON (JavaScript Object Notation) was chosen as the format for encoding of the message content. Each message is divided into blocks. Every block represents one request, which the router must handle, together with its parameters. After receiving a message, the daemon will attempt to handle each request and send back a response. The response shall contain information about the success of the operation and possibly also additional information for each block.

Every message must also contain information about the version of the protocol, to make it possible to detect routers with deprecated version of the user module.

Table 4.3 shows the requests types that need be supported by the user module.

4.3.2 Set LAN operation

While *cs-controller* has to distinguish between 4 different modes, in which router's interface can be operating, the router does not. Router is not even informed about what the new mode is. It will take the same action when setting a LAN into *public* mode as when setting it into *private* mode. This action includes changing the appropriate settings file and restarting its related system services.

Setting an interface into *ignored* mode can be disregarded on router's side entirely, because it only affects the behavior of *cs-controller* and does not lead to creation of any message for router.

1:1 NAT mode not only modifies LAN settings, but it also causes creation of several iptables rules on router. It uses NETMAP extension of iptables to create a mapping between a range of virtual addresses and the local network. Currently there are up to 253 virtual

Type	Parameters	Description
Setup LAN	• Interface name. • IP and netmask. • DHCP enabled. • DHCP pool range. • (virtual_ip, netmask).	Configures LAN settings of a single interface. When parameters *virtual_ip* and *netmask* are present, 1:1 NAT is configured.
Routing update	• Routes to add. • Routes to delete.	Creates and/or deletes static routes, leading into the tunnel.
Reconnect		Restarts the OpenVPN connection.
Retrieve configuration		Retrieves a list of supported interfaces, together with their current configuration.

Table 4.3: Types of requests that are sent to routers and must be implemented by the application that runs there.

addresses allocated for each router and it is up to the user how he will divide them between the interfaces. If he decides to give all of them to a single interface, 3 rules will be created:

1. A rule for translating destination address of packets, which are coming from the tunnel, from virtual IP address to local IP address.
2. Rule that prevents translation of the virtual IP address that belongs to the router itself. Without this, the address translation would prevent *cs-controller* from contacting the router.
3. A rule for translating source address of packets (from real to virtual), which are coming from the specific interface and are heading into the tunnel.

However, there is a problem. The 3rd rule needs to be placed into *POSTROUTING* chain (or one of its sub-chains) of a *nat* table. It has to be the *nat* table because we want to use *NETMAP* target, which cannot be used in other tables. It also has to be a *POSTROUTING* chain, because it's the only place, where the output interface is known (and we want to affect specifically those packets that are heading into the tunnel interface). Unfortunately, while the output interface is known here, the input interface is not. So we cannot directly create a NETMAP rule that would affect only packets that are heading into the tunnel and came from a specific interface. To get around this problem, one additional rule will be introduced. In the *PREROUTING* chain of the *mangle* table (here the input interface is known), a rule will be added, which will mark each packet that comes from the specified interface. The rule in the *POSTROUTING* chain will then check the input interface by matching only those packets that have been marked.

4.3.3 Routing update operation

Whenever a new LAN becomes accessible or inaccessible to a router (because of a change in group membership or through change in some device's LAN settings), the router is sent a message, telling it to add or remove a particular route. This management of routes is needed only for LANs that are behind interfaces configured in 'public' mode. Interfaces operating in 1:1 NAT mode use virtual IP addresses (from the block of addresses that are assigned to the given router), and as such belong to a single huge virtual network that can be routed into a tunnel with a single rule, and do not require routing updates.

4.3.4 Reconnect operation

The purpose of the *Reconnect* request is to restart a tunnel in order for changes done to OpenVPN client file (stored on CS) to take an effect. This could also be done by forcefully terminating the connection from the CS's side, however, it would take significantly longer time before the connection would be automatically re-established.

4.3.5 Retrieve configuration operation

Whenever a new router is added to the system, the *cs-controller* will send it a request for retrieving its LAN configuration. In response to this request, the router will send a list of the names of interfaces that it supports and their current configuration. Any interface, which is used as a default gateway, will not be reported with the others. This is to prevent *cs-controller* from managing such interface and offering a way to accidentally cut away our access to that device.

4.4 Security concept

Before any new router is added into the system, several security-related actions need to occur. While the CS leaves verification of the router to the user (there is a manual validation required before it gains access into the system), the router also needs to be able to verify that it is contacting the real CS and not one belonging to an attacker. To achieve this, the router begins by contacting a Dispatch Server, asking for the CS's certificate. This is just a one steps in a long list of actions and exchanged messages, which result in the router being successfully added into the system.

4.4.1 Certificate placement

The entire security concept is based around combination of manual validation and X.509 certificates. Some of the certificates come with the application, some are created on start-up and still others are received during runtime. Table 4.4 shows for each entity, what certificates and keys will get into its possession at some point in time. Certificates and keys that begin with "*CS_*" are a special case. They are installed on CS during the installation process, but unlike DS certificate, they are added into the installation package by the administrator of the Customer Server. This is not the best way of importing own key pairs, but it is sufficient as a proof of concept. All "*CS_*" certificates are signed by *CS_CA* (where *CS_CA* is an exception – it is either self-signed, or signed by a third party CA).

Each router will have 2 different key pairs of its own. When the user module is started for the first time (or when keys are deleted), it creates *Router_TLS* key and a self-signed

certificate. This key pair is then used for communication with DS and with CS. The other key pair it receives from the Customer Server and uses it only for OpenVPN. This key pair includes a certificate signed by *CS_CA*.

Entity	Comes with installation	Created on start-up	Comes after start-up
Router	DS_crt	Router_TLS_crt, Router_TLS_key	Router_OVPN_crt, Router_OVPN_key, CS_CA_crt, CS_TLS_crt
Customer Server	CS_CA_crt, CS_CA_key, CS_OVPN_crt, CS_OVPN_key, CS_TLS_crt, CS_TLS_key, CS_WEB_crt, CS_WEB_key DS_crt		Router_TLS_crt
Dispatch Server	DS_crt, DS_key		CS_TLS_crt

Table 4.4: Placement of keys and certificates.

4.4.2 Initial actions

Figure 4.3 shows in numbered steps, what communication will occur after a new Customer Server is installed. Note that for each component, the certificates and keys that came with its installation are displayed next to it. Individual steps are in further detail described as follows:

1. After a new CS is started, it will send a registration request via TLS connection to the Dispatch Server (DS). This is possible because CS has DS's certificate. This will create a new entry in the database located at the DS, storing the following information:
   - CS's external IP address,
   - CS's UID (Unique ID, different for each Customer Server),
   - CS's certificate.
2. While the CS could authenticate the DS in the previous step, since it knew its certificate in advance, the same is not true for the DS. The customer who runs the CS has his own key pair (imported during installation of the CS), that we do not have access to, and will contact us via email or telephone to identify his registration request, stored in the database. Once a license is issued for the customer, the matched record in the database will then manually be set as *Validated* and becomes available for step 3.

3. After the installation and start of the user module in a router (and after each restart), it will contact the DS and ask for a certificate and IP of its CS. This communication will occur within a TLS channel. Unlike in case of CS, the router can verify identity of the Dispatch Server right from the start, because its certificate will come with installation of the user module. This is possible because there will be only 1 DS across all customers (while there may be many CSs). On the other hand, DS is not capable of verifying identity of the router, but this is not a concern, because DS does not provide anything that would need to be kept secret. If the DS gives details about a CS to an attacker, then the attacker's router still needs to be manually validated on CS, before it can do any harm.

4. To decide what CS the router belongs to, the Dispatch Server uses a unique ID (UID) that the router sends in its message. The UID comes with installation of the user module, but can be changed any time if needed. For this UID the Dispatch Server looks up an appropriate address and certificate of a Customer Server, and sends it back to the router. If the CS isn't validated on DS, then a negative response is sent instead and the router repeats its request after a pre-set amount of time has passed There is a reason why UID can be stored into the router during installation of a user module, but CS's certificate cannot. It is because this certificate isn't created or known until the customer sets up his CS, while the UID and DS's certificate are known before that.

5. Once the router receives details about its CS, it starts sending it queries about whether it is validated or not. These queries are sent in a configurable interval, which is set on the CS and delivered to the router in each response. A UDP protocol without any security layer is used for this polling to minimize use of CS's resources. When the CS receives first of those queries, it creates a new entry in its database, storing the SSL certificate there.

6. Until the router is manually validated on the CS, it will be receiving negative responses and repeat sending the validation requests. After router's validation, a positive response is given and the router proceeds to step 7.

7. Once the router is validated by a user, the CS generates a new OpenVPN certificate and key. The router then attempts to create a secure connection and the CS uses the certificate, which the router sends in its TLS handshake, to look-up whether it is one of the validated routers. If it is so, then the CS sends him the OpenVPN certificate, key and *CS_CA* certificate, which are necessary for establishing OVPN connection.

8. The router creates an OpenVPN tunnel with the CS. This connection remains open indefinitely.

4.5 Proxy

To make full use of tunnels between VPN controller and routers, the Customer Server shall serve as a proxy server that redirects https requests onto the routers. Since the system is not meant to be a full configuration management tool (although it will manage parts of it), this feature provides administrators a way to browse routers' full web interface from anywhere in the internet.

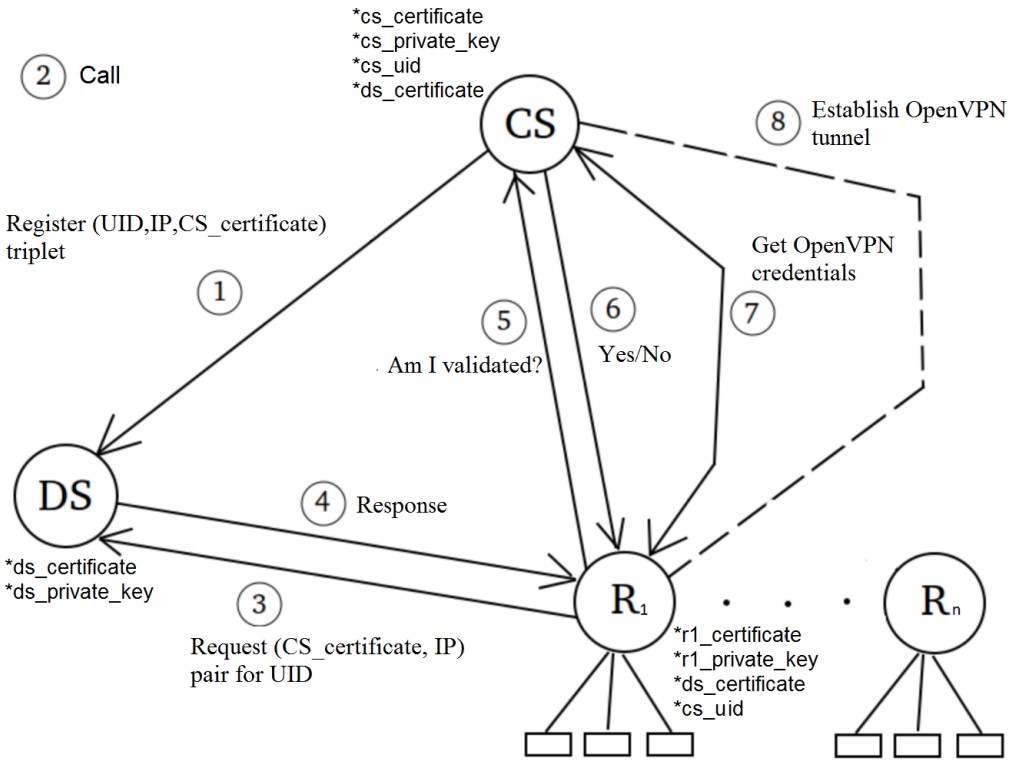

Figure 4.3: Diagram of communication between the main components. For each entity, a list of the resources that come with its installation is displayed.

For each new router with access into the VPN, a unique URL will be generated. Even though each router's website is protected by a login and password, to maintain higher security, their URL must be difficult (if not impossible) to guess. At the same time it is desirable to make it apparent, what router the URL refers to. For this reason, the following format was chosen:

```
<virtual_ip> - <30_rnd_letters> . <2nd_level_domain> . <1st_level_domain>
```

For proxy access to work, it requires an appropriate record to be present in a publicly available domain name server (DNS). *Apache2* web server was chosen as the tool for implementing redirections of web requests. For each router, there will be a *VirtualHost* record within a separate configuration file. Those files will then be imported into the *Apache*'s main configuration file. Thus, if proxy access to some router needs to be removed, it suffices to remove one file (no parsing of configuration is necessary), and issue a *reload* command to *Apache*.

4.6 Transaction system

As a method for managing the *cs-controller* from external applications (web pages, scripts), a database table of transactions was chosen. This provides serialization to all incoming requests. Since many operations affect the models of multiple routers, it would be extremely complicated to execute them in parallel. The transaction system also provides persistence

to those requests that cannot be executed immediately (for example those that require a router to be online).

4.6.1 Database

An SQLite relational database was chosen over the more complex databases like MySQL or PostgreSQL. We don't need a network access to the database. There will also not be many database operations that would need to be done concurrently. In fact, our requirements are fully met by SQLite and there is no need to use more complicated engines that would, in all likelihood, operate more slowly under our conditions than this simple but fast C library.

4.6.2 External API

All external requests for creation of transactions will go through a single interface called *External API*. The task of this interface is to validate the parameters of requests and then prepare and write transactions into the database. In case of any errors, it will propagate an error string through an exception back to the caller, so that the precise reason of failure can be displayed. This layer detects any errors that have something to do with improper input from the user (for example if he/she attempts to assign overlapping network addresses to multiple LANs in *public* mode). Figure 4.4 shows how External API fits to the rest of the system.

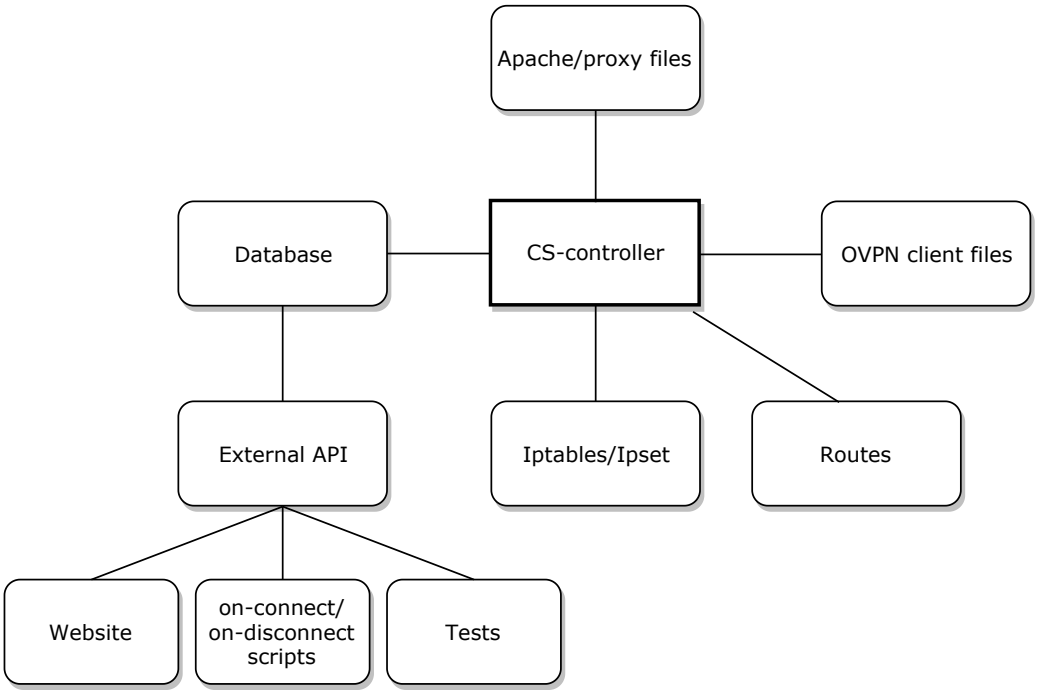

Figure 4.4: CS architecture overview

4.6.3 Pending configuration

For those parts of configuration that require synchronization with router, the database must not only hold information about the current state (what is configured on router right now),

but also the information about the future state (what configuration user wants there). The future state of a configuration is called pending configuration. This is specifically required for LAN settings and static routes. Each pending configuration has its own database table. A user will always be displayed the *pending* configuration. If it does not exist, he will be shown the current configuration.

4.6.4 Execution of transactions

The main loop of *cs-controller* will periodically load all transactions from the database and execute them. The transactions may be divided into 2 categories: those that require communication with routers and those that do not. If a transaction requires communication with router, a pending configuration will be created and the transaction deleted. If a new pending configuration is to be created while another one already exists, the new one will overwrite the old one. If a communication with router isn't required, the transaction will be immediately executed and then also deleted.

For each router that has at least 1 pending configuration, is online and doesn't already have a delivery thread, a new thread will be created, whose goal it is to communicate the changes to the router. After the router sends back a response, the thread is closed and the pending configuration is changed into a permanent configuration.

Execution of transactions of a particular router is skipped while there exists its delivery thread. The existence of the thread will normally be very short and the temporary skipping of transactions should not negatively affect the user experience. This small concession decreases the complexity of the system. If it wasn't done this way, it would be necessary to allow having multiple pending configurations of the same type for each router. A pending configuration may only be created through a transaction, never directly.

4.6.5 Error handling

When a transaction fails due to a problem on the side of the Customer Server (for example failure of some database operation), the information about the error is logged into a syslog file and the transaction is marked as failed. The failed transaction then remains in the database (for debugging purposes), but is ignored by the *cs-controller*. This can be done because unlike the pending configurations, the existing transactions are not visible to the user in any way. If an operation fails due to a problem on the side of the router, the associated pending configuration, which is visible to the user, is marked as failed. The user can then change the configuration, which will lead to replacement of the failed pending configuration with a new one.

Chapter 5

Implementation

As a programming language was chosen C++, because most user modules for routers are written in C/C++ and there exist several C libraries for Advantech routers that could be needed in the future. After choosing this language, it was advantageous to also write Customer Server and Dispatch Server with that, so that the code can be re-used across the entities. Although a website is an integral part of the system, its implementation will not be discussed here, because it was created fully by a third party. The same is also true for an installation script and a build system.

5.1 External API

The API between the CS-controller and other applications is divided into 2 layers (shown in figure 5.1). The inner layer is a C++ class *ExtClientApi* that implements checks of the input. It also handles creation of transactions and reading of data from DB. If any other action needs to be taken in response to an API call, it would also be placed here. The outer layer consists of multiple classes, whose sole purpose is to make methods of the inner layer available from different scripting languages. Currently 2 languages are supported:

- *PHP* (needed for website),
- *Python 3* (needed for regression tests).

This outer layer of the API receives arguments from the caller (usually script) wrapped in a special class, different for each language. It converts the arguments into common C++ types, calls a method of *ExtClientApi* and passes it the converted parameters. Third party libraries *Python/C*[1] and *PHP-cpp*[2] were used to implement this. The reason for using *Python* for regression tests (instead of *php*, which was already needed for the website) is to make use of a pre-existing test framework, available within the company.

5.2 User module

The main loop within the routers' user module is implemented as a finite-state machine shown in figure 5.2. Before this loop is started, several actions occur.

- Daemonizing activities are performed.

[1] docs.python.org/3/c-api
[2] www.php-cpp.com

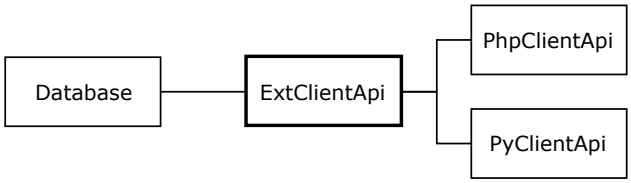

Figure 5.1: Architecture scheme with both API layers.

- Syslog is initialized.)

Also, a self-signed certificate and key are generated with the use of OpenSSL binary. This is, however, done only once, as it is placed within the module's installation script. From the all states shown in figure 5.2, the most time is usually spent inside the *s_listening state*. This is where all incoming messages are handled. Function *select()* with a timeout is used there, so that when there are no messages for a long period of time, a piece of a code that checks the state of OpenVPN tunnel can be run. Those checks are currently done by scanning log messages of the OpenVPN daemon. If, for example, CS migrates to a new address, the user module is able to eventually arrive back into *s_getting_cs_ip* state and obtain the new IP address from the Dispatch Server.

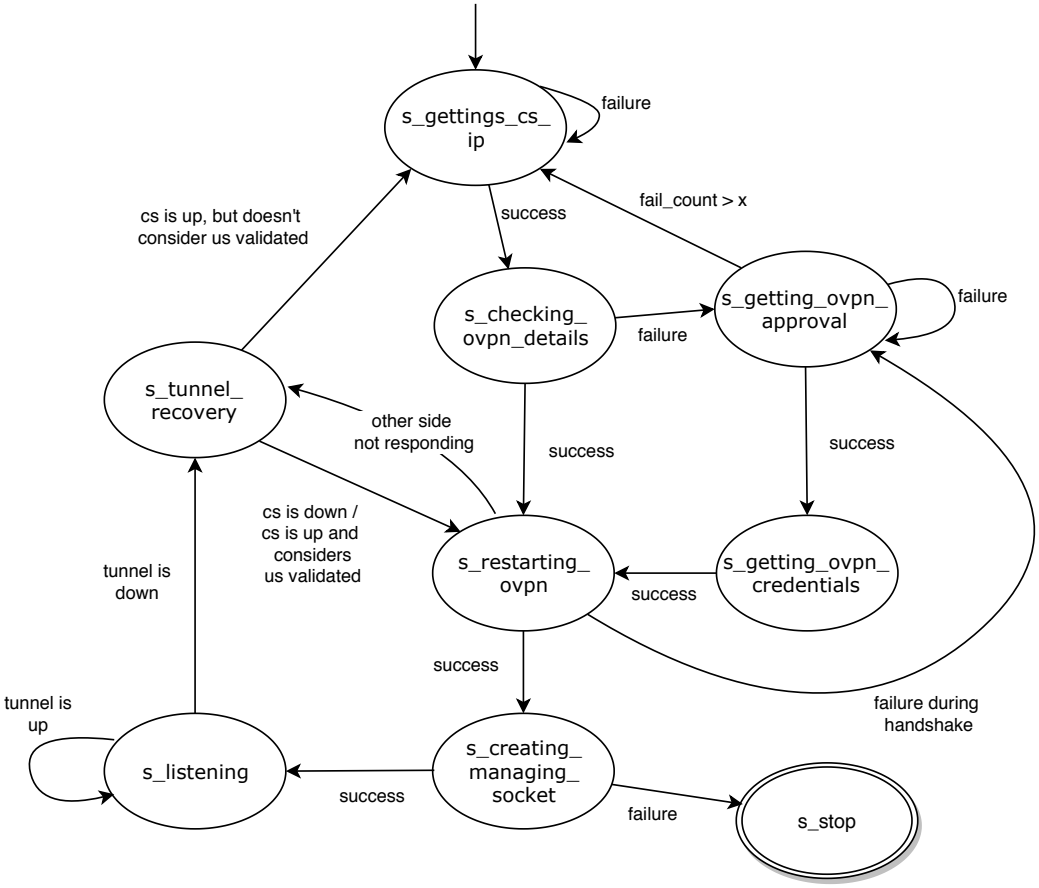

Figure 5.2: FSM representation of the router's main loop..

Web UI

As a method of configuring the router module (user needs to fill in the address of the Dispatch Server), simple web pages were created for routers. It was done through the use of *cgi* scripts, since that is the standard way of creating web UI for user modules. There are 3 different pages:

- Page with the logs of the user module,
- Page with the logs of the OpenVPN,
- Configuration page (can be seen in figure 5.3).

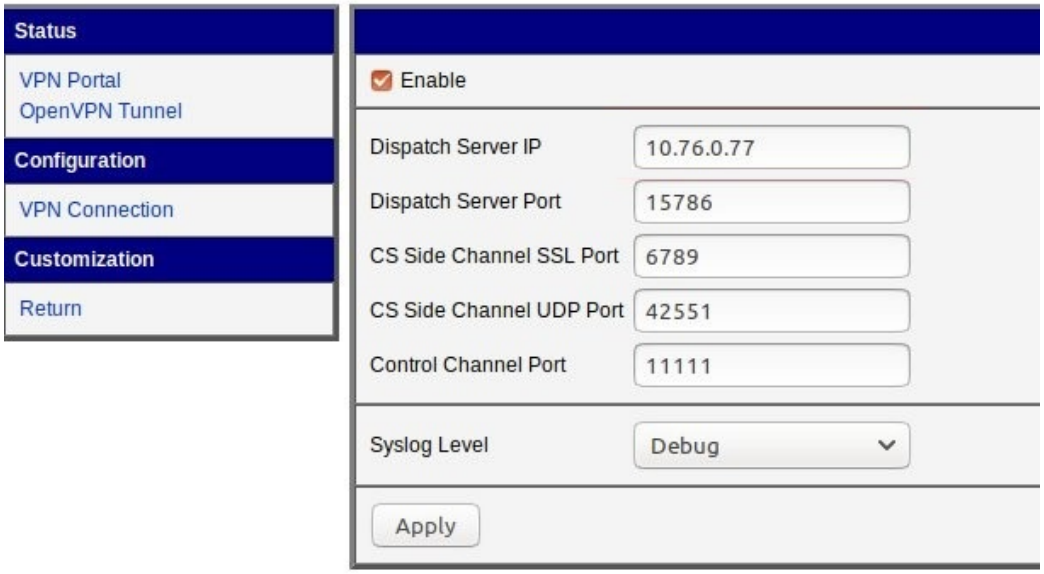

Figure 5.3: User interface for the router module.

5.3 Customer Server

The main daemon is named *vpnportal-cs* and it is implemented as a *systemd* service. On its start-up, several initialization actions are executed via a shell *init* script, including:

- Start of OpenVPN.
- Creation of main iptables chains.

When the service is stopped, everything related to the daemon is flushed from iptables and *ipsets*, and OpenVPN is terminated. Static routes do not need to be erased, because once the OpenVPN is stopped and tunnel goes down, they are removed automatically by the operating system. Any messages that need to be printed, are printed via the *syslog()* function. When printing a message, 4 different priority levels are used:

- Debug,

- Warning,
- Error,
- CriticalError (Leads to immediate shut down of the service).

After the *init* script finishes initialization actions and starts the main daemon, the following actions are taken:

1. Syslog connection is initialized. Signal handlers are registered. Database is initialized (values like '*time_of_start*').
2. Rules for iptables, entries for ipsets, and static routes are created, based on the network and LAN setting information from DB.
3. IP address and certificate are registered at DS (OpenSSL library is used for secure communication).
4. A sub-process called *SideChannelDaemon* is created via *fork()*. The purpose of this daemon is to respond to incoming validation queries from the routers. It uses a shell script to generate new OpenVPN keys and certificates for routers that ask for them. For each incoming request/query, a new thread that will handle it is created. However, the certificate generation script can't be run multiple times in parallel and is thus considered a critical section, which is enforced via locking of *std::mutex*.
5. The main loop is entered, where the process starts executing transactions and sending commands to routers. To prevent creation of inconsistent state on service shut-down, a custom function was created as a handler of incoming *SIGTERM* signal. This function will only set a global variable *SIGTERM_ARRIVED*, which is periodically checked by both processes. Once a change of this variable is detected, a safe shut down is carried out.

The main loop of *vpnportal-cs* has the following structure:

1. Response messages from routers are taken from an object of class *DeliveryService*, whose task it is to oversee communication with routers.
2. For those requests that succeeded on routers, their related pending configuration is changed in the database into permanent configuration. For those that failed, an error flag is set in their pending configuration.
3. Any transactions belonging to online routers are loaded from DB.
4. Those transactions that do not require communication with routers are immediately executed and deleted.
5. Those transactions that do require communication with routers lead to creation of pending configuration in DB.
6. If there are any pending configurations, messages for routers are created.
7. For each router with a message, a communication thread is created.

Statistics

Various statistics about the system are collected and may be displayed on the Customer Server's web page. Based on the way they are collected, there are 2 types of statistics (although web accesses them both in the same way – through PhpClientApi):

1. Statistics that can be read/inferred from the current state of the database (number of online devices, number of network entries, uptime, etc.).
2. Statistics that are read from OpenVPN's management console (throughput for each device, global throughput).

A separate C++ application was created to handle all statistics-related queries. Based on the parameters given, it prints relevant statistics in structured format. It uses TCP connection to read data from OpenVPN's management console (which is listening on a localhost interface). To access necessary data, "status 2" command is sent via the connection and then the answer is read from the socket, parsed and printed.

5.4 Dispatch Server

Dispatch server is implemented as a *sysmted* service *vpnportal-ds*. It is a simple, small application, consisting of a single loop, where it handles incoming requests. It uses *sqlite3* database for storing data about Customer Servers and *OpenSSL* library for TLS communication.

5.5 Security measures

During the implementation, several new potential security vulnerabilities of the system were discovered and had to be analyzed.

5.5.1 "Pretending to be a CS" attack

An attacker that would get access to a router A, which is in a group with router B, could, in theory, craft a message that would look like the one that Customer Server sends when it wants to change router's configuration. The attacker would then send such a request to the router A and make it change its LAN configuration or do anything else that the protocol supports at the time. This attack is relatively easy to execute, it requires only:

- Knowledge of the protocol.
- Knowledge of the port where the routers are listening.
- Access to one of the routers that are grouped together.

To remove this vulnerability, a check of source IP address was added to handling an incoming message on routers. Since the address of a CS is always known by the router (at that stage), it is possible to drop any control message whose source address does not match it.

5.5.2 IP spoofing

IP spoofing means creation of a packet with a false source IP address. It was postulated, that someone with access to one of the routers could bypass the concept of groups by faking the source IP address of a packet. The theory was, that since iptables rules on a Customer Server work on the bases of looking at source and destination IPs, it would allow this packet to reach a device that is in a different group. However, after thoroughly testing it, the conclusion was reached that this is not possible. The OpenVPN would drop any packet whose source IP is not among the LANs that the given router is responsible for. It knows which routers are responsible for what LANs, because this information is stored in the OpenVPN client files.

Chapter 6

Stress tests

The system must undergo a series of stress tests in order to measure how well it scales with increased number of client devices. It is also important to find out where its bottleneck is. For these purposes, a modified version of the routers' user module will be compiled for PC. Every such user module, called *router agent*, will then be run in its own virtual container through an application called Docker. Several test scenarios will be conducted and performance measured for a setup with 100, 350 and 600 router agents.

6.1 Docker

Docker is a virtualization software used for running small packages of software called *containers*. Containers differ from virtual machines in the fact that they are very lightweight, because they share OS's kernel across multiple instances. They are created from executable files called *images*. When building a new Docker image, a special configuration file *Dockerfile* first needs to be prepared and the following options specified:

- What existing image to use as a template for this one's OS.
- What actions to execute when a new container is started from this image.
- What additional layers of changes to add on top of the template during creation of image.

6.2 Fargate

Originally, Amazon's Fargate service was considered for running all the router agents. It is a cloud service that offers an easy way to start and manage high numbers of Docker containers. One of its advantages is the ability to precisely define the amount of resources that each instance would have. It also makes it easy to decrease or increase number of instances at any time and provides management of their log files. However, it turns out that a crucially needed parameter *–cap-add* (passed when starting a new Docker container) is not currently supported by this service. Without this, the OpenVPN within the container wouldn't function properly and the idea of using Fargate service had to be abandoned.

6.3 Router agents

The user module created for PC differs from the original in its reactions to requests from the Customer Server. It will skip modifications of settings, restarts of services and other actions that are normally done in response to various requests. Instead, it will just send back a positive response. A special case is a response where the router sends back a list of the interfaces it supports and their current configuration. This response will be hardcoded to 2 made up interfaces with randomly chosen settings. One additional change is in generation of a unique name (used for generation of an SSL certificate), for which an application called *status* is normally used. This application is present only in real routers and to simulate its behavior, a *BASH* script was created and added into our Docker image. Its purpose is to print router's serial number, which the script generates randomly.

Ubuntu 16.04 was chosen as a template for the Docker image. Through a *Dockerfile*, additional software, including *rsyslog*, *ping* , *openvpn*, *openssl* and a PC version of the user module had to be installed. The final size of the image is approximately 240 MB. When starting a container (instance of the image), the following parameters must be used for OpenVPN to function properly:

```
1) --cap-add=NET_ADMIN
2) --device=/dev/net/tun
```

6.4 Setup

Amazon's EC2 cloud service was used for hosting a Customer Server and a Dispatch Server. Both were installed and run on a single virtual machine, whose specifications can be seen in table 6.1.

Instance type	Processor	Cores	Memory	Network performance
T2.large	3.0 GHz Intel Scalable Processor	4	16 GB	Moderate [1]

Table 6.1: Spefications of the Amazon's instance that was used for hosting both CS and DS.

Docker containers were hosted across several different machines on multiple locations. Since very little resources are needed for each router agent, it was possible to host hundreds of such agents per computer. The specifications of the machines used can be seen in table 6.2.

All the machines that were mentioned (including the one hosting DS and CS) were using Ubuntu 18.04 as their operating system. Devices that would normally be located behind routers will be ignored in the stress test. From the load it puts CS under, there is no noticeable difference between having those devices behind routers, and generating traffic directly from the router agents.

6.5 Preparation

Before the tests could be conducted, a long list of scripts needed to be prepared to automate as many tasks as possible. Otherwise the amount of attention and work required would

ID	Processor	Cores	Memory
Laptop1 (virtual machine)	Intel(R) Core(TM) i7-8750H @ 2.20 GHz	6	13 GB
PC1 (virtual machine)	Intel(R) Xeon(R) E3-1245 v5 @ 3.50GHz	4	16 GB
Amazon1 (t2.micro)	3.3 GHz Intel Scalable Processor	1	1 GB

Table 6.2: Specifications of the machines that were used for hosting router agents.

make those tests extremely difficult and time consuming. The scripts were used for the following tasks:

- Starting and stopping X docker containers on a given machine.
- Measuring time it takes until routers are all online and synchronized.
- Assigning LANs to all routers.
- Initiating ping between all routers.
- Creating networks and populating them with routers.
- Creating X firewall rules for all networks
- Validating all devices.

Also, since the OpenVPN does not, per default, support a netmask higher than /16 (which would allow only 255 routers, since for each router there are 255 reserved addresses), a patched version was created and this limitation removed. All tests were then conducted with a netmask of /8.

6.6 Measurements

The following statistics were measured. In the brackets are listed the tools that were used for its measurement.

- used memory (top),
- processor's load (top),
- throughput (nload),
- time (custom scripts / manual measurements).

All the measurements were done on the Customer Server. Memory usage was measured for the system as a whole, not for specific processes. Throughput information was collected for 2 interfaces: *tun* (traffic going through the OpenVPN tunnel) and *eth* (traffic outside of the tunnel + traffic inside the tunnel).

6.7 Scenarios

For each setup with a different number of routers, multiple test scenarios were conducted to see how the system behaves under different circumstances. Measurements were taken for every one of them. The main scenarios were:

- All routers are asking for validation.
- Validating all routers at once.
- All routers are connected via OpenVPN, but idle (no operations are currently done by any user and none or minimal traffic is generated by routers).
- Routers are generating traffic by flooding pings at each other (while randomly divided into 10 networks).
- Setting LAN into *public* mode on all devices at once.

6.8 Results

In total, tests with 3 different numbers of routers were run, which should be enough to see the trends. In the end it was not possible to do very precise measurements and all the results should be taken only as approximations of what the real performance would be. Due to time difficulty of re-doing these tests and fixing problems, the highest number of devices that was completed in time for this thesis is 600.

During the execution of the stress tests, several flaws in the implementation were encountered, improved and the tests then re-run. What will be discussed here are the results after the most critical issues were fixed. These included:

- **Running out of file descriptors (reaching the system limit)** – This was fixed by setting a limit on the maximum number of concurrent communication threads.
- **Database operations failing due to reaching a timeout, while waiting on a database lock** – This was partially fixed by increasing the timeout.
- **Website being sometimes very slow and unresponsive** – This problem occurred when too many write operations kept the database locked for a long period of time. It was partially fixed by making the main daemon handle transactions in smaller bursts and sleeping for a moment after each.

As can be seen from tables 6.3 and 6.4, the memory usage was relatively small and scaled very well with increasing numbers of routers. The CPU load was spiking at 0.92 during execution of operations (like changing LAN to all devices at once), but kept around 0.00 once the *cs-controller* finished its work. Even during the *ping flood* scenario, the CPU load didn't get over 0.05. An interesting statistic is the amount of OpenVPN overhead during the ping test. While 8200 Kb/s was going through the tunnel, the total traffic going through its physical interface was 13600 Kb/s. When CS was restarted while 600 devices were connected, it took approximately 7 minutes until all devices came back online and synchronized (although 500 of them were online within 2-3 minutes). When validating 600 devices, it took 4 minutes until they were all online and synchronized.

Scenario	**Memory usage [MB] (100 routers)**	**Memory usage [MB] (350 routers)**	**Memory usage [MB] (600 routers)**
Routers are asking for validation.	292	258	339
Routers are connected but idle.	290	391	581
Clicked “validate” for all routers.	297	347	756
Set LAN to public mode for all routers.	304	403	601
Ping flood (tun: 8200 Kb/s, eth: 13600 Kb/s).	295	428	598

Table 6.3: Measurements of total used memory during various test scenarios.

Scenario	**CPU load (100 routers)**	**CPU load (350 routers)**	**CPU load (600 routers)**
Routers are asking for validation.	0.0	0.0	0.03
Routers are connected but idle.	0.0	0.0	0.01
Clicked “validate” for all routers.	0.92	2.43	3.46
Set LAN to public mode for all routers.	0.36	0.83	0.93
Ping flood (tun: 8200 Kb/s, eth: 13600 Kb/s).	0.02	0.03	0.05

Table 6.4: Measurements of averagel CPU load during various test scenarios.

Chapter 7

Conclusion

In this diploma thesis, various approaches and tools for managing large numbers of routers were studied. Remote configuration tools Ansible, Puppet and NETCONF library were examined and described in detail. Various tunneling software were studied and their features listed. Strengths and weaknesses of SmartCluster, an already existing application similar to the one that would be created, were pointed out and discussed.

A system for managing large numbers of VPN-connected routers was then designed and implemented. It has a star topology, with one central entity to which all routers are connecting. It uses OpenVPN as the bases for creating tunnels and builds on top of it. It supports dynamic clustering of routers together into separated groups and offers a way to add or delete firewall rules that would affect only the selected groups. It provides a range of statistics about traffic, and displays online status for each router. All communication is secured via the use of certificates, which are securely distributed to routers through a single trusted server. For each new router that is added into the system, an URL is generated for accessing the router's web interface through the tunnel, where the central entity works as a proxy server. Whenever a new router is added into the system, configuration of its LANs is retrieved and displayed. Each LAN can be reconfigured through the central entity. If a router is offline at the time when its configuration is changed, both the old configuration and the new one are temporarily available to the user, until the router comes back online and the changes are delivered. The user can manage what LANs should be accessible from the virtual network, and what LANs should not, by setting them into one of 4 modes. A client application for routers, called user module, was implemented, and needs to be installed on routers before they can start interacting with the system.

The whole system underwent a series of stress tests with 100, 350 and 600 (simulated) routers, with the conclusion that memory usage and CPU load scale quite well and a strong machine should probably easily handle even thousands of connected routers. The only issue being slow responsiveness of the application when an operation is executed for large numbers of routers at once. This is due to lack of optimization and should probably be easy to improve in the future.

There are many ways how the current application could further be improved. Currently there is support only for routers of Advantech manufacturer. However, the user module was successfully modified to run on PC (with some limitations), and with small effort, it should be possible to port it to any device that uses a Linux distribution as its operating system.

Bibliography

[1] *Ansible module development: getting started.* [Online; accessed 05.01.2019]. Retrieved from: https://docs.ansible.com/ansible/latest/dev_guide/developing_modules_general.html

[2] *Certificate authority and SSL.* [Online; accessed 05.01.2019]. Retrieved from: https://puppet.com/docs/puppet/6.0/ssl_certificates.html

[3] *Design and Implementation of SoftEther VPN.* [Online; accessed 05.01.2019]. Retrieved from: https://www.softether.org/4-docs/9-research/Design_and_Implementation_of_SoftEther_VPN

[4] *Differences between VPN Server and VPN Bridge.* [Online; accessed 05.01.2019]. Retrieved from: https://www.softether.org/4-docs/1-manual/5._SoftEther_VPN_Bridge_Manual/5.3_Differences_between_VPN_Server_and_VPN_Bridgey

[5] *Facter 101.* [Online; accessed 05.01.2019]. Retrieved from: https://puppet.com/blog/facter-part-1-facter-101

[6] *Frequently asked questions - Licensing.* [Online; accessed 05.01.2019]. Retrieved from: https://openvpn.net/vpn-server-resources/frequently-asked-questions-licensing/

[7] *Overview of Puppet's architecture.* [Online; accessed 05.01.2019]. Retrieved from: https://puppet.com/docs/puppet/5.5/architecture.html

[8] *Running Puppet on nodes.* [Online; accessed 05.01.2019]. Retrieved from: https://puppet.com/docs/pe/2017.3/run_puppet_on_nodes.html

[9] *SoftEther VPN Essential Architecture.* [Online; accessed 05.01.2019]. Retrieved from: https://www.softether.org/4-docs/1-manual/2._SoftEther_VPN_Essential_Architecture/2.2_User_Authentication

[10] *SoftEther VPN Project.* [Online; accessed 05.01.2019]. Retrieved from: https://www.softether.org/

[11] *Subsystems: Agent/master HTTPS communications (platform manual).* [Online; accessed 05.01.2019]. Retrieved from: https://puppet.com/docs/puppet/5.2/subsystem_agent_master_comm.html

[12] *Ultimate Powerful VPN Connectivity.* [Online; accessed 05.01.2019].
Retrieved from: https://www.softether.org/1-features/1._Ultimate_Powerful_VPN_Connectivity

[13] Anderson, L.; Madsen, T.: *Provider Provisioned Virtual Private Network (VPN) Terminology.* [Online; accessed 05.01.2019].
Retrieved from: https://tools.ietf.org/html/rfc4026

[14] *Ansible Installation Guide.* [Online; accessed 05.01.2019].
Retrieved from: https://docs.ansible.com/ansible/latest/installation_guide/intro_installation.html

[15] *Asynchronous Actions and Polling.* [Online; accessed 05.01.2019].
Retrieved from: https://docs.ansible.com/ansible/2.5/user_guide/playbooks_async.html

[16] *Building a simple module.* [Online; accessed 05.01.2019].
Retrieved from: https://docs.ansible.com/ansible/2.3/dev_guide/developing_modules_general.html

[17] *Httpapi Plugins.* [Online; accessed 05.01.2019].
Retrieved from: https://docs.ansible.com/ansible/latest/plugins/httpapi.html

[18] *Intro to Playbooks.* [Online; accessed 05.01.2019].
Retrieved from: https://docs.ansible.com/ansible/latest/user_guide/playbooks_intro.html

[19] *The benefits of agentless architecture.* [Online; accessed 05.01.2019].
Retrieved from: https://www.ansible.com/hubfs/pdfs/Benefits-of-Agentless-WhitePaper.pdf

[20] Björklund, M.: *YANG - A Data Modeling Language for the Network Configuration Protocol.* [Online; accessed 05.01.2019].
Retrieved from: https://tools.ietf.org/html/rfc6020

[21] Chisholm, S.; Trevino, H.: *NETCONF Event Notifications.* [Online; accessed 05.01.2019].
Retrieved from: https://tools.ietf.org/html/rfc5277

[22] Frankel, S.; Krishnan, S.: *IP Security (IPsec) and Internet Key Exchange (IKE).* [Online; accessed 05.01.2019].
Retrieved from: https://tools.ietf.org/html/rfc6071

[23] Geerling, J.: *Make your ansible playbook flexible, maintainable, and scalable.* [Online; accessed 05.01.2019].
Retrieved from: https://www.ansible.com/blog/make-your-ansible-playbooks-flexible-maintainable-and-scalable

[24] Kanies, L.: *Puppetl.* [Online; accessed 05.01.2019].
Retrieved from: https://www.aosabook.org/en/puppet.html

[25] Kent, S.: *IP Authentication Header*. [Online; accessed 05.01.2019].
Retrieved from: https://tools.ietf.org/html/rfc4302

[26] Kent, S.; Atkinson, R.: *IP Encapsulating Security Payload (ESP)*. [Online; accessed 05.01.2019].
Retrieved from: https://tools.ietf.org/html/rfc2406

[27] Kotuliak, I.; Rybár, P.; Trúchly, P.: *Performance Comparison of IPsec and TLS Based VPN Technologies*. [Online; accessed 05.01.2019].
Retrieved from: https://www.researchgate.net/publication/254015270_Performance_comparison_of_IPsec_and_TLS_based_VPN_technologies

[28] Kraft, M.; Hilgner, L.: *SmartCluster Application Note, (2015)*. [Online; accessed 05.01.2019].
Retrieved from: http://www.infopulsas.lt/files/eshop/358/1SmartCluster_Application_note_20151026.pdf

[29] Lewis, M.: *Comparing, Designing, and Deploying VPNs*. Cisco Press. 2006.
Retrieved from: https://www.researchgate.net/publication/240918687_Comparing_Designing_and_Deploying_VPNs

[30] Manral, V.; Hanna, S.: *Auto-Discovery VPN Problem Statement and Requirements*. [Online; accessed 05.01.2019].
Retrieved from: https://tools.ietf.org/html/rfc7018

[31] *NETCONF library for clients and servers*. [Online; accessed 05.01.2019].
Retrieved from: https://netopeer.liberouter.org/doc/libnetconf2/devel/group__client__session.html

[32] Reshma, A.: *What Is Ansible? – Configuration Management And Automation With Ansible*. [Online; accessed 05.01.2019].
Retrieved from: https://www.edureka.co/blog/what-is-ansible/

[33] Saurabh: *Puppet Tutorial – One Stop Solution For Configuration Management*. [Online; accessed 05.01.2019].
Retrieved from: https://www.edureka.co/blog/puppet-tutorial/

[34] Yonan, J.: *Reference manual for OpenVPN 2.4*. [Online; accessed 05.01.2019].
Retrieved from: https://openvpn.net/community-resources/reference-manual-for-openvpn-2-4/