Bachelor's thesis

NTNU
Norwegian University of Science and Technology

Magnus Aarøe
Pegah Banaei
Siti Aisya Binti Md Nazeri

Adopting a cloud-based network: An assessment of Cisco Meraki's ease of implementation

Bachelor's thesis in Digital Infrastructure and Cybersecurity
Supervisor: Besmir Tola
May 2023

Magnus Aarøe
Pegah Banaei
Siti Aisya Binti Md Nazeri

Adopting a cloud-based network: An assessment of Cisco Meraki's ease of implementation

Bachelor’s thesis in Digital Infrastructure and Cybersecurity
Supervisor: Besmir Tola
May 2023

Norwegian University of Science and Technology

Abstract

Software Defined Networking is a new and growing networking architecture in which the network devices' control and data planes are separated. This has made it possible to administer and manage the devices through a single centralized controller rather than doing so individually, greatly simplifying the process.

Due to this separation, the network is also made highly programmable, meaning that network administrators are free to use high-level programming languages to create programs and applications based on their needs.

In this thesis, we explore deploying a Software Defined network using Cisco Meraki, and compare it to that of a traditional network setup, simulated using Cisco Packet Tracer. We do so to gain an understanding of each solution's ease of implementation, and whether it could be utilized by those with limited networking experience. We also explore Cisco Meraki's potential for network automation and programmability.

Our results demonstrate that Cisco Meraki's solution is indeed easier to use overall when compared to a traditional network solution. However, it is not without its own set of challenges and possible downsides, such as its heavy reliance on having an internet connection to manage the network.

Moreover, we find that a Cisco Meraki network has a high automation potential, and that there are a variety of tools that can be used for this purpose. Examples of these are the Meraki dashboard API, the Meraki Python library, and a community for developers to share their solution with others and vise versa.

Sammendrag

Software Defined Networking er en ny og voksende nettverksarkitektur der kontroll- og dataplanet i nettverksenhetene er separert. Dette har gjort det mulig å administrere nettverksenhetene gjennom et sentralisert kontrollsystem framfor å forholde seg til hver enhet individuelt, som igjen har forenklet den generelle prosessen for nettverksadministrasjon.

På grunn av denne separasjonen blir nettverket lettere å programmere, som vil si at administratorene står fritt til å bruke høynivå programmeringsspråk til å lage programmer og nettverksapplikasjoner etter eget behov.

I denne oppgaven utforsker vi utrullingen av et Software Defined nettverk ved å bruke Cisco Meraki, og sammenligner det med et tradisjonelt nettverksoppsett simulert i Packet Tracer. Vi gjør dette for å få en bedre forståelse for hvor enkelt det er å implementere hvert nettverk, og om de kan tas i bruk av noen med begrenset erfaring innen nettverk. Vi utforsker i tillegg Cisco Meraki sitt automatiseringspotensiale.

Våre resultater viser at Cisco Meraki sin løsning er generelt mer brukervennlig sammenlignet med den tradisjonelle løsningen, men samtidig har den også noen utfordringer, blant annet et behov for internettforbindelse for å kunne administrere nettverket.

Videre finner vi at Cisco Meraki har et høyt automatiseringspotensiale, og at det finnes diverse verktøy som kan brukes for dette formålet. Eksempelvis har vi Meraki dashboard APIet, Meraki Python biblioteket, og et miljø for utviklere der de kan dele sine løsninger med andre og omvendt.

Contents

Abstract . . . v

Sammendrag . . . vi

Contents . . . vii

List of Figures . . . x

List of Tables . . . xi

List of Acronyms . . . xiii

Preface . . . xiv

1 Introduction . . . 1
1.1 Background . . . 1
1.2 Thesis Topic . . . 1
1.2.1 Research Questions . . . 2
1.3 Partner Organization . . . 2
1.4 Scope and Delimitation . . . 2
1.5 Project Participants . . . 3
1.6 Thesis Outline . . . 3

2 Theory . . . 5
2.1 Core Concepts and Definitions . . . 5
2.1.1 The OSI Model . . . 5
2.1.2 Network Devices . . . 6
2.1.3 Packet Flow . . . 6
2.1.4 Planes . . . 7
2.2 Traditional Network Architecture . . . 8
2.2.1 Network Addressing . . . 9
2.2.2 Dynamic Host Configuration Protocol . . . 10
2.2.3 Network Address Translation . . . 10
2.2.4 Local Area Network . . . 11
2.2.5 Virtual Local Area Network . . . 11
2.2.6 Wide Area Network . . . 11
2.2.7 WAN Technologies . . . 12
2.3 Network Management . . . 13
2.3.1 Secure Shell and Remote Access . . . 14
2.3.2 Simple Network Management Protocol . . . 14
2.3.3 Network Time Protocol and Syslog . . . 14
2.3.4 Management Challenges . . . 14
2.4 Network Security . . . 15
2.4.1 General Security . . . 16
2.4.2 Access Control and Firewall . . . 16
2.4.3 Intrusion Detection and Intrusion Prevention Systems . . . 17
2.4.4 Zero Trust Architecture . . . 17
2.5 Software Defined Networking . . . 17
2.5.1 Software Defined WAN . . . 18
2.5.2 SDN Management . . . 19

2.5.3 SDN Security 19

3 Methodology **21**
3.1 Equipment and Materials 21
3.1.1 Cisco Packet Tracer 21
3.1.2 Cisco Meraki Stack 21
3.1.3 Cisco Meraki Dashboard 23
3.2 Criteria of Analysis 24
3.2.1 Ease of Configuration and Setup 24
3.2.2 Ease of Management 26
3.2.3 Automation Potential 26
3.3 Use Cases 26
3.3.1 Use Case 1: Single-branch connectivity (LAN) 27
3.3.2 Single-branch Connectivity (LAN) with Packet Tracer 27
3.3.3 Single-branch Connectivity (LAN) with Cisco Meraki 37
3.3.4 Use Case 2: Branch-to-branch connectivity (WAN) 40
3.3.5 Branch-to-branch Connectivity (WAN) with Packet Tracer 41
3.3.6 Branch-to-branch Connectivity (WAN) with Cisco Meraki 45
3.4 Management and Monitoring Capabilities 46
3.4.1 Using Packet Tracer 46
3.4.2 Using Cisco Meraki 46
3.5 Automation 47
3.5.1 Automation with Packet Tracer 47
3.5.2 Automation with Cisco Meraki 48

4 Results **50**
4.1 Ease of Configuration and Setup 50
4.1.1 Rubric Results - Packet Tracer 50
4.1.2 Rubric Results - Cisco Meraki 51
4.2 Ease of Management and Monitoring 53
4.2.1 Using Packet Tracer 53
4.2.2 Using Meraki 58
4.3 Automation Potential 61
4.3.1 Automating Use Case 1 62
4.3.2 Retrieving Information 70
4.3.3 Dynamic Response 71
4.3.4 Modules, Expansions, and Templates 72

5 Discussion **73**
5.1 Discussion of The Two Network Solutions 73
5.1.1 Traditional Legacy Network - Packet Tracer 73
5.1.2 Software Defined Network - Cisco Meraki 74
5.2 Research Questions 75
5.2.1 Research question 1 75
5.2.2 Research Question 2 75

6 Conclusion **77**
6.1 Summary 77
6.2 Future Work 77

References **78**

A Appendix **81**
A.1 ”Internet” Configuration 81
A.2 Python Scripts 81
A.2.1 Use case 1 configuration script 81
A.2.2 Function imports for use case 1 configuration script 89
A.2.3 Automation Get-script 93
A.3 Best and Worst Cases for Ease of Configuration 97

List of Figures

1 Simplified form of a packet . . . 7
2 Planes in a network device . . . 8
3 Interconnected networks . . . 9
4 IPv4 Address Format. . . . 9
5 Division of an IP address' local portion . . . 10
6 Wide Area Network connection . . . 12
7 Simplified form of a packet with an IPsec header . . . 13
8 Separation of planes in Software Defined Networking . . . 18
9 The Meraki equipment stack . . . 22
10 The Meraki dashboard . . . 23
11 Packet Tracer - Single-branch LAN topology . . . 28
12 Employee WiFi AP Configuration . . . 30
13 DHCP IP allocation . . . 32
14 ISP ping response . . . 33
15 Establishing SSH connection to TRD-R . . . 34
16 Entering global configuration mode using SSH client . . . 34
17 Rejected SSH session . . . 35
18 GUEST-IN ACL verification . . . 36
19 TRD-R NTP status . . . 37
20 Meraki Single-branch LAN topology . . . 38
21 Changing hostname for a device . . . 38
22 The steps of VLAN creation . . . 39
23 ACL rules in the Meraki dashboard . . . 40
24 Branch-to-branch WAN topology . . . 41
25 GRE-tunnel route verification . . . 43
26 Active TRD-R security associations . . . 44
27 Static UDP port for VPN . . . 45
28 VLANs allowed to use the VPN . . . 45
29 Monitor column in dashboard's Network-wide menu . . . 46
30 The Meraki change logs . . . 47
31 Generating an API-key . . . 48
32 Meraki API request using Postman . . . 49
33 TRD-R interface status . . . 53
34 TRD-R DHCP pool status . . . 54
35 TRD-R DHCP pool binding . . . 54
36 TRD-R NAT status . . . 55
37 SNMP MIB browser settings . . . 55
38 SNMP MIB browser TRD-R system get-request . . . 56
39 SNMP MIB browser TRD-R system get-request . . . 56
40 Syslog service on NTP-Syslog server . . . 57
41 Netflow collector application on NTP-Syslog server . . . 58
42 Switch ports status page in dashboard . . . 59
43 One variant of the summary report page . . . 59
44 Response from the getDeviceNames function . . . 60
45 An example of a MiB get-command to the Meraki switch . . . 60
46 Uplink traffic information during normal operations . . . 61
47 Uplink traffic information during an outage . . . 61
48 First object in response from getSwitchPorts() . . . 71

List of Tables

1 Layers in the OSI reference model 5
2 Questions on ease of configuration 25
3 Trondheim VLAN subnet plan 27
4 LAN Internet IP allocations 28
5 Stooges' LAN addressing plan 29
6 VLAN to switchport mapping on TRD-SW 29
7 Meraki VLANs and addressing plan 38
8 Oslo VLAN subnet plan 40
9 WAN Internet IP allocations 42
10 Stooges' WAN addressing plan 42
11 Results of ease of configuration - traditional network in Packet Tracer 50
12 Results of ease of configuration - Cisco Meraki network 52
13 Best and worst case results for ease of configuration rubric 97

List of Acronyms

ACL Access Control List. 16

API Application Programming Interface. 18

CAM Content Addressable Memory. 6

CIA Confidentiality, Integrity, Availability. 15

CLI Command Line Interface. 2, 14, 15

DHCP Dynamic Host Configuration Protocol. 10

IDS Intrusion Detection System. 17, 20

IMP Interface Message Processor. 5

IP Internet Protocol. 6, 7

IPS Intrusion Prevention System. 17, 20

IPsec Internet Protocol Security. 13

ISO International Organization for Standardization. 5

ISP Internet Service Provider. 11

LAN Local Area Network. 11

MAC Media Access Control. 6

MIB Management Information Base. 13, 47

MPLS Multi-Protocol Layer Switching. 12

NAT Network Address Translation. 10

NGFW Next Generation Firewall. 22

NTP Network Time Protocol. 14

OID Object Identifier. 47

OSI Open Systems Intercommunication. 5, 6

PAT Port Address Translation. 32

PoE Power over Ethernet. 22

QoS Quality of Service. 22

RBAC Role-Based Access Control. 17

RSTP Rapid Spanning Tree Protocol. 22

SA Security Appliance. 21, 22

SD-WAN Software Defined Wide Area Network. 2, 18

SDN Software Defined Networking. 1, 17

SFP Small Form-factor Pluggable. 22

SMB Small to Medium sized Business. 1

SNMP Simple Network Management Protocol. 14, 47

SSH Secure Shell. 14

STP Spanning Tree Protocol. 22

TCP Transmission Control Protocol. 7

TCP/IP Transmission Control Protocol/Internet Protocol. 5

UTM Unified Threat Management. 22

VLAN Virtual Local Area Network. 11

VPN Vitrual Private Network. 13

VS Code Visual Studio Code. 48

WAN Wide Area Network. 11

WAP Wireless Access Point. 22

Preface

This report was written as our bachelor's thesis for the study of Digital Infrastructure and Cybersecurity at the Norwegian University of Science and Technology (NTNU). The aim of the report is to explore the complexities of a software-defined network from the perspective of users whose level of expertise is similar to ours. The adopted solution exploits a Cisco Meraki cloud-based infrastructure using an MX router, an MS switch, an MR wireless access point, and a web-based dashboard, provided to us by Cisco.

In exploring the ease of deployment, configuration, management, and automation aspects of a Cisco Meraki based network, we hope to provide relevant insights for anyone who might be interested in a similar solution for themselves, such as a small or medium sized business.

Our findings may also provide grounds for NTNU's Department of Computer Science (IDI) to consider expanding the current networking curriculum for Digital Infrastructure and Cybersecurity.

We would like to thank our mentor Prof. Besmir Tola, and project proposer Prof. Olav Skundberg who suggested the assignment and represents NTNU-IDI, as well as Arjan Toxopeus from Cisco Meraki, who contributed to our research and gave us insight into Meraki themselves. Lastly, we would like to thank NTNU for our years of studies and the opportunities given to us because of it.

1 Introduction

1.1 Background

As networks and the need for them grow, so does the complexity and difficulty of maintaining and managing them. Current networks often need per-device configurations, done with low-level languages that may also be vendor-specific, which makes the process of configuring and upgrading them very time and resource consuming. Automation of this process is also as good as non-existent, which adds to the overall complexity of today's ever-growing networks [1]. Another challenge is the expertise needed to configure and manage a network, which often necessitates hiring external technicians or relying entirely on a service provider.

Needless to say, then, that what we know of as the traditional network architecture is rather rigid, which has caused changes on the architectural level to be quite costly both in terms of time and resources (monetary and operational) [2]. This rigidity also makes networks less dynamic and ill-equipped for responding to the ever-changing network conditions of today, which, in turn, results in a need for constant manual changes and reconfigurations [3].

Software Defined Networking (SDN) has emerged as a solution to current network architectures' rigidity by decoupling the control layer of networking devices from their data layer, removing the need for per-device configurations, and simplifying the process by logically centralizing this control layer in a *controller* [1]. Two big benefits of this change are the introduced ease of making changes through a program rather than predefined, low-level commands or configuration files, and the ease of management of such a system since the centralized controller offers a global view of the entire network [3].

Cisco Meraki is one of the providers for a software defined network, complete with a specific set of devices made for their cloud-managed solution, and a centralised dashboard used for configuring, operating, and monitoring the network through a web browser. Their solution is marketed as highly scalable and suitable for a variety of networks, from simple home networks to campus or even enterprise level ones [4]. Cisco Meraki also cooperates with vendors like Telenor and Atea, giving customers the option to leave the management of their networks to the service provider they bought their equipment from if they do not wish to do so themselves.

With this as our background, we wish to explore and compare the capabilities of Cisco Meraki as an SDN solution with a traditional legacy network, specifically from the perspective of a small to medium sized business (SMB) with zero to limited knowledge about networks and their operations. Given the complexities inherent to a network, is an SDN truly easier to implement and use? Is this new technology a practical and beneficial solution for SMBs to adopt?

1.2 Thesis Topic

The thesis assignment was proposed due to the rise of software-defined networks in recent years, and due to NTNU-IDI's interest in incorporating the use of SDN related tools into the Digital Infrastructure and Cyber Security study programme. A test installation of Cisco Meraki was decided on, which would be showcasing the main capabilities of Cisco Meraki as an SDN solution, as well as comparing it to a legacy network setup similar to the ones available to students in the university's labs. Automation of network management using APIs and scripts was also suggested to be part of the investigation.

After discussions with Prof. Olav Skundberg, NTNU-IDI department representative, and Prof. Besmir Tola, our project supervisor, we decided to focus on exploring and

understanding both the benefits and the challenges introduced by a change in network infrastructure. With SDN being a relatively new technology which seems to be marketed as the solution to most traditional networking problems, we wanted to understand how challenging it could be for those with limited resources, expertise, and knowledge within networking, such as an SMB, to implement a network. In doing so, we could then evaluate if Cisco Meraki's SDN solution would be a viable option for them at all.

SDN also comes with the promise of high flexibility due to automation and network programmability, so we also wish to understand the extent of this promise by exploring the solution's automation potential and its associated benefits.

The research questions below were formulated to aid us in this research.

1.2.1 Research Questions

Research Question 1

How easy is it for an SMB with limited expertise to deploy, configure, and manage a Cisco Meraki based network compared to a similar network using traditional legacy devices?

Research Question 2

What is the automation potential of a Cisco Meraki-based network compared to that of a traditional legacy network?

1.3 Partner Organization

This thesis was completed using equipment and tools lent to us by NTNU, provided by Cisco Meraki. Arjan Toxopeus, our contact in Cisco Meraki, provided us with an inside-perspective of the current market situation for SDN solutions versus the traditional catalyst solutions, as well as extra equipment at his location which we used for our use-case's Oslo branch. He also gave us technical guidance in familiarizing ourselves with their product and contributed to us getting an idea of how SMBs realistically utilize the available SDN solutions, which aided us in forming use-cases for our research purposes. Despite Cisco Meraki's assistance in our research, they have not had an influence on the findings of this report.

NTNU's Department of Computer Science (NTNU-IDI) is also worthy of a mention as a partner organization, as their representative, Prof. Olav Skundberg, suggested the topic, and because the findings of this report will be used by them to consider adding use of Cisco Meraki's SDN tools to Digital Infrastructure and Cyber Security's practical curriculum.

1.4 Scope and Delimitation

This project's aim was to explore Cisco Meraki's SDN capabilities, showcase its automation potential, and assess how easy it would be to adopt it as a network operation and management solution. We were provided with three Meraki devices we used in our testing: an MX security and SD-WAN appliance, an MS switch, and an MR Wireless LAN device, which are further introduced in Section 3.1.2. In order to investigate Cisco Meraki's ease of use as an SDN solution, we decided to compare it to a similar traditional network setup, with low-level legacy management options such as the Command Line Interface (CLI), emulated inside the widely used network emulator Cisco Packet Tracer [5].

Although Cisco Meraki, and Software Defined Networking in general, allows users to get quite granular with their network configurations, it was not something we delved into in this project. Our focus was on less experienced users, such as those with only basic understanding of networks and networking concepts and less hands-on experience, and how easy it would be for these users to adopt and adapt to an SDN solution.

To compare these two types of networks, we began by first researching SDNs in general, their core concepts and ideas, and how they differ from traditional network architectures and technologies. We then created a use-case for our project to determine our network's needs and configured both an emulated (Packet Tracer) and a physical (Cisco Meraki) network for this use-case. We defined a set of criteria that properly outlined what ease means to us and used these criteria to measure the ease of configuring both networks according to our use-case. For comparing our networks' automation potential, we decided to compare automation methods for both types of networks, and further showcase some automation and scripting methods unique to Cisco Meraki as an SDN.

Outside of our scope are almost all of the ways traditional legacy network operators have adapted to the rigidity of such architectures, as well as the enterprise level capabilities of Cisco Meraki and Cisco DNA. Advanced configurations and scripting for network automation also falls outside the scope of this project.

1.5 Project Participants

This thesis was done as a final assignment for Pegah Banaei, Siti Aisya Binti Md Nazeri, and Magnus Aarøe's bachelors assignment in Digital Infrastructure and Cyber Security at NTNU. The topic was chosen out of a genuine interest for the field of networking, and because we felt it was relevant for the present and future of network management.

1.6 Thesis Outline

Chapter 1: Introduction

The first chapter provides background information about the project, its participants, as well as the thesis topic and scope of the report.

Chapter 2: Theory

In this chapter, we lay the theoretical foundation and explain some core concepts of networking, traditional network architectures, and Software Defined Networking.

Chapter 3: Methodology

This chapter explains the methods we chose for comparing and evaluating the two different network types based on our thesis topic. We further explain how we defined *ease* in this context, describe the use-cases we devised for testing purposes, and provide an overview of the equipment and materials we used.

Chapter 4: Results

In this chapter, we describe the outcome of the implementation of our two networks, present the results from our measurements and evaluation of their *ease*, and compare the two network solutions based on the criteria we defined in the previous chapter.

Chapter 5: Discussion

In this chapter, we discuss the process of implementing and configuring networks using both solutions, the challenges we faced during the process, and answer our research questions.

Chapter 6: Conclusion

This chapter provides a summary of our findings and conclusions, as well as work that can be done in the future.

2 Theory

This chapter explains the networking concepts and technologies necessary for understanding technologies and methods used in Chapter 3: Methodology. We first explain some of the fundamentals of networks and networking, then move on to what constitutes a traditional network architecture and explain some basic network functions (sec. 2.2), how it is managed and secured (sec. 2.3 and 2.4), before providing an overview of a new architecture known as Software Defined Networking (sec. 2.5).

2.1 Core Concepts and Definitions

A network, per definition, is an intercommunication between groups or systems [6]. The word is used to refer to a wide array of interconnected mechanisms, but the definition most relevant to us is that of computer networks and their interconnection. The purpose of this kind of inter-connectivity is to enable devices located across small and large geographical distances to communicate with each other.

We can trace the beginnings of today's Internet and inter-networking to ARPANET from the late 1960's, when Interface Message Processors (IMPs), predecessors of today's routers, were used as communication nodes between two sites over long distances, allowing the two connecting sites' host computers to communicate with one another [7].

A set of communication conventions, referred to as protocols, were later defined to allow interoperability regardless of the network hardware or physical connection [8]. Two important examples of this are the Internet Protocol Suite (TCP/IP) and the Open Systems Intercommunication (OSI) reference model, which defines how data sent over a network should be put into packets, addressed, routed, and received [9, 10].

2.1.1 The OSI Model

Layer	Function	Examples
7 - Application	The interface between user-applications and the network	HTTP, SNMP
6 - Presentation	Provides information on the data's format that is necessary for processing	TLS, SSL
5 - Session	Establishes, closes, and manages the connection between programs at each end of a connection	VPN
4 - Transport	Splits data into smaller packages at the source, and reassembles them at the destination	TCP, UDP
3 - Network	Defines the route data should take from source to destination	IPv4, IPv6, MPLS
2 - Data Link	Defines communications with the underlying physical network components	Ethernet, Wi-Fi
1 - Physical	Transmits data across hardware components used for connecting devices on the network	Physical Cables, NIC

Table 1: Layers in the OSI reference model

As data travels from one network host to another, it goes through several stages to get to its destination [7]. The International Organization for Standardization (ISO) divides

these stages of travel into seven separate layers [11], forming the Open Systems Interconnection (OSI) reference model. Table 1 provides an overview of each layer and its function, along with some example protocols or components used in each.

The layers are defined and separated based on their functions, protocols, and differences in data handling to allow changes to be made to one layer without a need to change the others [11]. This ensures that the layers are independent from those above or below them, allowing the implemented functions for each layer to be chosen on a case-to-case basis.

As data traverses through the network, it will go through each layer in the model in order [7]. From the source host, it will begin at the top of the stack, the application layer (L7), and move in decreasing order towards the physical layer (L1). The data is encapsulated with the necessary information in the correct format as it moves from one layer to the next. When it arrives at the destination host, it will go through the layers once more in increasing order. The data is then de-encapsulated, layer by layer, as it is processed and presented.

2.1.2 Network Devices

Switches

Switches are devices that are responsible for forwarding frames to other devices within their local network [10]. A frame is data that is encapsulated, by a layer-specific protocol, with a header and a trailer as it passes through the data link layer (L2). Inside a frame's header are the source and destination Media Access Control (MAC) addresses of the encapsulated data.

Switches operate primarily on the data link layer and have physical ports to which other devices can connect. Through communicating with their ports, switches will learn of any connected device's MAC address, which they then store in what is known as a MAC address table or a Content Addressable Memory (CAM) table [12].

Routers

When a frame is forwarded to the network layer (L3), it is de-encapsulated into what is known as a packet. Much like frames, these packets include source and destination addresses in the header of the data they encapsulate. The difference is that frames use hardware addresses, whereas packets use Internet Protocol (IP) addresses [8].

Routers are devices that operate on the network layer, forwarding packets across network boundaries based on packets' IP source and destination addresses. The forwarding decision is made based on the router's routing table, which includes information about known network addresses and their locations.

There are switches that also have network layer routing capabilities, called Layer 3 or multilayer switches [13], but they are not within the scope of this report.

2.1.3 Packet Flow

In the book *Internetworking With TCP/IP*, Douglas E. [8] defines packet flow as "[...] a sequence of packets sent from a given source to a given destination". These packets are the original data broken down into smaller segments for transportation, which are later reassembled at the destination [10]. Packet switching, in turn, refers to how network devices, such as routers and switches, process and forward packets individually, and that

packets in the same flow can take different routes to their destination [7]. Splitting data into smaller packets allows for greater flexibility in a network, as packets can be sent and received in any order, which increases the number of hosts that are able to simultaneously communicate and send data through a connection.

In its simplest form, a packet consists of two main parts: the header and the payload, as shown in Figure 1. The payload is the data the packet is carrying, and the header is where information such as the source and destination addresses are stored [8, 7].

Figure 1: Simplified form of a packet

The addresses used for locating and identifying hosts on a network are known as Internet Protocol (IP) addresses, which are numerical and unique within a network [8]. The protocol itself is a Layer 3 one, and has two versions: IPv4 and IPv6 [7]. As IPv6 falls outside our scope, we will only be referring to IPv4 when we mention IP addresses throughout the rest of this report.

As a packet traverses the network, each router it passes through will inspect its IP address and make a decision on where it should be forwarded to. This is a dynamic process and is the reason behind packets being able to take varying paths towards the same destination. However, packets can also get lost or corrupted and, consequently, dropped on the way [7].

Layer 4 protocols such as the Transmission Control Protocol (TCP) are used to ensure packets that arrive on destination are without errors by re-requesting any dropped packets, and reassembling them before sending the data to the upper layer [8].

2.1.4 Planes

Infrastructure devices in a network, such as routers and switches, all have multiple planes on which they operate. These planes are the control plane, the management plane, and the data plane (also known as the forwarding plane) [14, 1]. Figure 2 depicts these planes inside a network device.

Management Plane: This plane is the access and configuration point of the network. It is used for monitoring that the network is operating as expected, managing the network devices, and implementing policies and configurations.

Control Plane: This is the plane responsible for enforcing policies and configurations, as well as making forwarding and handling decisions for network packets using Layer 2 and Layer 3 mechanisms such as network topology and routing tables. The decisions made in this layer are passed onto the data plane.

Data Plane: This plane is the connection of the different ports on network devices, and is used to forward network traffic. Packets going through a router or switch move through this plane and are, amongst other things, forwarded, dropped, or changed based on the control plane's instructions.

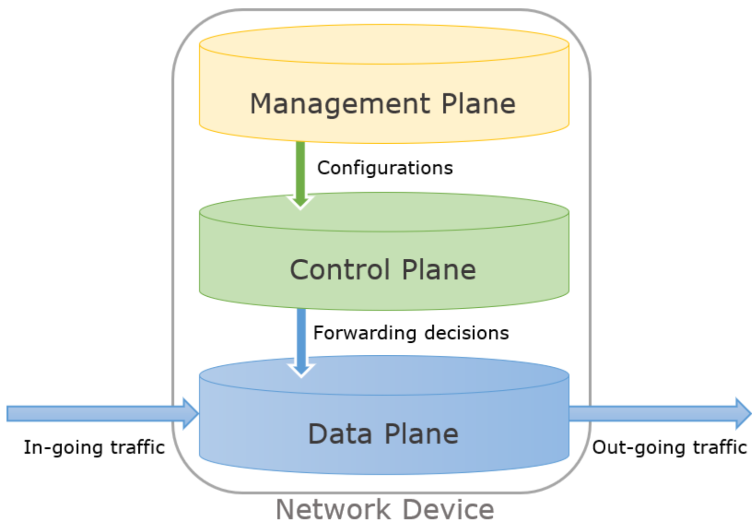


Figure 2: Planes in a network device

2.2 Traditional Network Architecture

When talking about traditional networks, we are referring to networks characterized by their hardware-centric approach, in which the control plane is distributed among the network devices [15]. This necessitates a per-device approach to configuration, which can be tedious and overly time-consuming. It might also result in an overly complex network that may be more prone to errors and misconfigurations [16].

At its simplest, a network needs a router, a switch, dedicated cables for connectivity, and some end-user devices that want to communicate using the network. The router and the switch both function as forwarding devices, but as previously mentioned, they operate on different layers. In order for the network devices to operate in tandem, they must be configured according to the network topology and setup.

Figure 3 presents a simple topology with different networks. On the edge of the network are the hosts and end-devices that wish to communicate with one another. These devices are physically connected to an access network, which is the network connecting them to the first router on their traffic path to the outside, also known as an edge router [10]. Beyond the edge router is the core network, which consists of a mesh of routers and switches that forwards traffic between other edge networks, forming the global overarching internet.

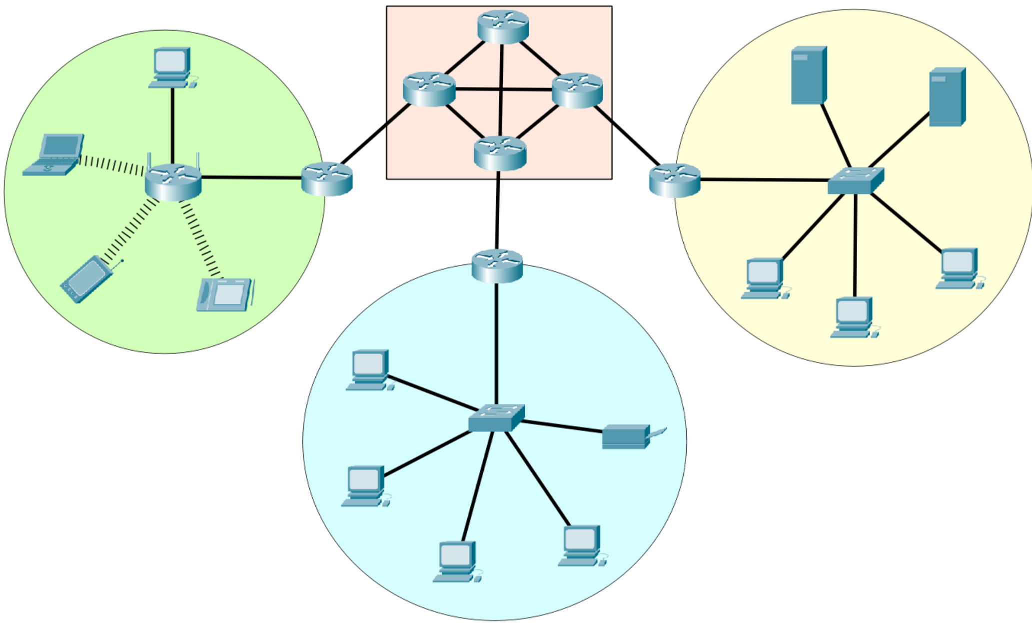


Figure 3: Interconnected networks

2.2.1 Network Addressing

In Section 2.1.2, we mentioned that switches use Layer 2 MAC addresses for forwarding traffic, while routers use Layer 3 IP addresses. MAC addresses are 48 bit addresses tied to network interface cards present in network devices and hosts [8] and are globally unique [7].

IPv4 addresses, in comparison, are 32-bit network addresses tied to an interface, which James Kurose defines as "the boundary between the host and the physical link [...]" [10], and consists of two parts: a prefix that identifies the network, and a suffix that identifies the host itself [8]. Devices with addresses that have the same prefix are able to communicate directly, whereas communication with others requires a router [7].

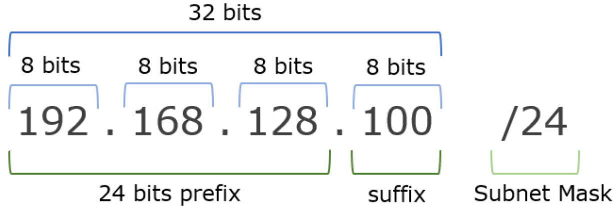


Figure 4: IPv4 Address Format.

As Figure 4 illustrates, IP addresses also include a /*N* slash notation, in which *N* is a number ranging from 1 to 32, known as the addresses' subnet mask [8]. This number indicates how many of the addresses' leftmost bits belong to the prefix, which also indirectly signifies how many of the rightmost bits belong to the suffix. The slash notation is

also known as the subnet mask because, inside a network, the suffix of addresses with the same prefix can be divided among several internal networks [17].

Consider a network that has been assigned the address 192.0.0.0 with a /16 prefix, meaning that all hosts within that network must have addresses in the form of 192.0.***.***. Externally, the network will be perceived as one singular network with 192.0. as its prefix, but internally, it can be divided into several networks, each with their own address prefix derived from the original.

This is done by dedicating a portion of the address suffix, referred to as the address' local portion [8], to specify which internal network the address belongs to, and the rest to identify the host within that network [17].

In our example, we can assume that we want three internal networks, and divide our initial address into the addresses 192.0.0.***, 192.0.1.***, and 192.0.2.***. By doing so, we have dedicated the first half of what was the local portion in the initial /16 address to specifying which internal network the address belongs to, while the rest is left for hosts in that network to occupy. Figure 5 shows this division using one of the addresses from our example.

Figure 5: Division of an IP address' local portion

2.2.2 Dynamic Host Configuration Protocol

The specifics of how organizations and companies obtain IP addresses for their networks falls outside the scope of this report. It is, however, worth explaining how these obtained addresses are distributed to individual devices within a network.

Dynamic Host Configuration Protocol (DHCP) is a Layer 7 protocol of the OSI model used for dynamically and automatically allocating IP addresses to devices when they connect to the network, choosing each address from a predefined address range [7, 10], referred to as a DHCP pool. Upon connection, new hosts will broadcast a DHCP request in the local physical subnet [18] and the DHCP server will respond by sending the necessary IP configurations to the host's MAC address. [8].

In case a dedicated DHCP server is not present in a subnet, a relay agent, such as a router [10], can be used for forwarding the DHCP messages between the server and the host [18, 7].

2.2.3 Network Address Translation

We can divide IP addresses into two groups: private addresses and public addresses. Private addresses are used internally for locating hosts within the same network, while public addresses are used for locating others across the public internet [10]. Every device that wishes to communicate with another using the internet, must be allocated a public IP addresses in addition to a private one.

Network Address Translation (NAT) is the technology used for associating and translating private addresses to public ones and vise versa [8]. NAT can be enabled on a

network's edge router, after which the router will replace (translate) the private source address inside packet headers with its own public address as it forwards them to outside networks. When a packet's destination sends a reply, the destination address will be the NAT router's public address, and, after the router receives the reply, it will translate the address back to the correct private address [10].

Using NAT not only hides the details of the private network from the rest of the internet, it also decreases the number of IP addresses needed for communicating through the internet because packets from the same network will have the same source address even though they may be coming from different hosts [7].

2.2.4 Local Area Network

Local Area Network (LAN) is a short-distance network that spans across a small area, such as an office building [8, 19], connecting devices in the same location using various technologies. The most prevalent LAN technologies are Ethernet [10] and Wi-Fi, which is a wireless implementation of Ethernet [7]. Figure 3 illustrates three different LANs using different LAN technologies.

Depending on the network's requirements, any number of Ethernet switches can be used for connecting any number of LANs. Given that switches are primarily Layer 2 devices, an Ethernet frame travelling through them will have hardware MAC addresses in its header for determining its source and destination. One or more Ethernet switches inside a LAN can connect to the LAN's edge router [10], allowing for communication with outside networks.

2.2.5 Virtual Local Area Network

While a LAN consists of devices connected to the same switch, a Virtual LAN (VLAN) allows for multiple isolated LANs to be attached to the same switch [10], as well as forming LANs from devices that are connected to different switches, by segmenting the network at the data link layer (L2) [7]. Traffic between VLANs is isolated [12] and we can assign a dedicated subnet to each of them. Devices in a VLAN will therefore assume they are the only ones connected to the switch, on their own network.

Although this type of traffic isolation provides a layer of security, it also poses a challenge for when different VLANs want to communicate with each other [7]. The simplest solution is to use a Layer 3 switch which also has routing capabilities, allowing it to forward traffic between VLANs without needing an additional device [12].

In cases of Layer 2 switches, we would need to introduce a router or another L2 switch to the setup, and define the connection between the devices as one belonging to multiple VLANs on the network, also known as trunking [12]. From the VLANs' perspective, it will seem as if they are each connected to their own switches, with an added device that connects them and allows them to communicate with each other [10].

2.2.6 Wide Area Network

A Wide Area Network (WAN), in contrast to a LAN, spans across a large geographical area, such as a city, country, or even continent [8, 19], and was initially invented due to a need to connect remote branches of the same organization together [7]. Figure 6 shows a simple WAN using the existing core network to connect two sites.

Traditional WANs are provided and generally managed by the Internet Service Provider (ISP), and a company who wants their own WAN must either lease a private dedicated

line from their ISP, or use the public packet switched network also provided by their ISP [20]. Routers play a big role in WAN architecture, as they are the primary devices used for forwarding traffic between the connected networks that form the WAN [8].

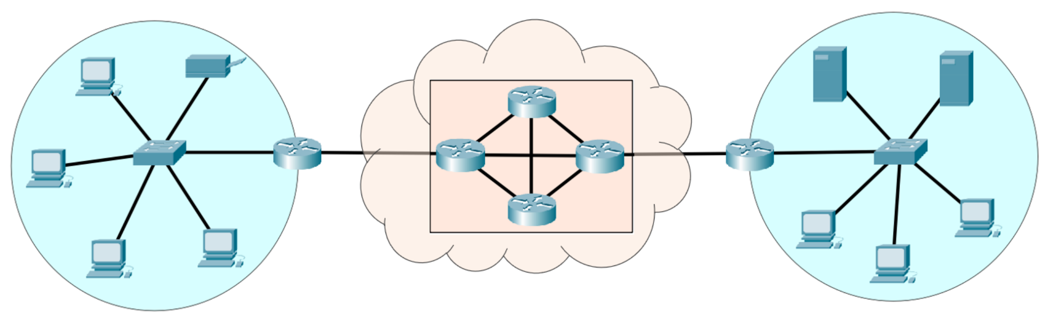


Figure 6: Wide Area Network connection

2.2.7 WAN Technologies

When WANs first appeared, they utilized point-to-point connections between the connecting networks. Today, most of them use packet switching over the ISP's network, which functions as a virtual point-to-point connection [7]. This means that in place of single and direct lines between connecting sites, which are costly despite their privacy, security, and quality benefits [20], the WAN traffic is forwarded using existing core network routers. The connection is thus referred to as *virtual point-to-point* because it still functions and seems as such to the connecting edge networks.

Private Lines

Private leased lines have the benefit of being highly secure, private, and can guarantee high quality bandwidth, causing it to be very expensive [20]. A completely private WAN infrastructure is even more costly as it needs to be installed and maintained by the organization that wants it, but there are those to whom the benefits of a private large-scale network outweighs the costs [10].

Multi-Protocol Layer Switching

Multi-Protocol Layer Switching (MPLS) is a Layer 3 network technique consisting of attaching labels to packets, and using only the contents of their labels to route packets through the network [7]. MPLS is specifically made to carry packets regardless of protocol [8], but specialized routers capable of reading MPLS headers are necessary for label-based forwarding to be possible [10].

In the context of WANs, MPLS is used by ISPs for establishing paths between multiple locations or branches over large distances [8]. This is because, with MPLS, it is possible to specify which path certain traffic should take towards a destination since paths are determined by the packets' labels instead of their IP addresses.

Virtual Private Network

A Vitrual Private Network (VPN) is an L2 or L3 method of securing a private connection between geographically dispersed sites to establish WAN connectivity over a public or unsecured network, providing another alternative to privately leased lines [7]. Much like with MPLS, the traffic is sent over existing network infrastructure. However, with VPN, the traffic is secured through encryption before it travels through a public connection such as the internet [10].

In addition to providing site-to-site connections, VPN clients (a program running on an end-device) can be used to give users who are connected to a different physical network remote access to the network. A VPN connection from one point to another is known as a VPN tunnel, and there are a few communication protocols that can be used for establishing one [7]. One popular example is IPsec, short for Internet Protocol Security.

IPsec itself is not a single protocol, but a collection of protocols that can be used at the network layer (L3) [8] for authentication and encryption, providing a secured network-to-network connection [7].

Packets being sent over a VPN tunnel with IPsec have their payloads encrypted, and are given an IPsec authentication header in addition to the normal IP header required for path determination and forwarding [10]. Furthermore, the choice of using IPsec is asymmetrical, meaning that either side of a connection can choose to use IPsec independently from the other [8]. Figure 7 depicts a simplified version of a packet with an IPsec header added to it.

Figure 7: Simplified form of a packet with an IPsec header

2.3 Network Management

Network management is the collective acts of configuring, monitoring, debugging, and maintaining a network [10, 8]. The person managing a network is often referred to as the network administrator or manager.

There are several components to network management. The authors of [10] define these in the Management Framework as such:

- Managing Server; the station from which the administrator initiates a change or monitors the network.
- Managed device; the network device that is being changed or monitored, for example a router.
- Management Information Base (MIB); data items which managed devices have to keep, their meaning, and which operations are allowed to be performed on them. Used for gathering specific network statistics.
- Management agent; the process running in the managed device that communicates with and executes commands coming from the management server.
- Management protocol; the protocol that defines the messages sent for managing the network.

2.3.1 Secure Shell and Remote Access

Secure Shell (SSH) is a protocol often used for establishing an encrypted connection to a network device for the purpose of management [13]. SSH uses a username and password for authenticating the administrator user, and gives access to the device's Command Line Interface (CLI). The protocol is used for remotely accessing devices without needing a physical link, and is considered secure because of the encrypted connection it provides.

2.3.2 Simple Network Management Protocol

Shortened to SNMP, Simple Network Management Protocol is a Layer 7 protocol defining the format for management messages between the management server and agents, as well as transport protocols for the messages [8]. Messages between server and agents are in the form of request/response, in which a get-request is for information retrieval, a set-request is for making a modification to the managed device, and a response is the reply back from the agent [10].

There have been three iterations of SNMP, with the latest one being SNMPv3 [7]. Previous versions of the protocol suffered from a lack of adequate security measures such as encryption, which resulted in SNMP being mainly used for accessing network statistics [8] and monitoring. SNMPv3, however, does provide strong encryption and authentication measures and is considered safe.

2.3.3 Network Time Protocol and Syslog

Network devices all have internal clocks for keeping the time, much like a phone or personal computer does. However, network devices' internal clocks are prone to drifting and becoming de-synchronized with other devices on the network over time [21]. The Network Time Protocol (NTP) is a solution for this issue, allowing devices on the same network to synchronize their clocks using a common server, referred to as the NTP server [7, 13].

NTP is essential for network management since de-synchronized clocks and inaccurate timestamps on logged events makes it severely difficult to understand when and in what order events took place as well as how they might correlate. This, in turn, makes creating a timeline of events, troubleshooting, and monitoring almost impossible [13].

In addition to their internal clocks, network devices also have mechanisms for logging events, which network administrators can use to understand which events took place, when, and what they were caused by [7]. The protocol most used for accessing such information is known as syslog, and networks will often use a syslog server to which network devices send their system messages for administrators to access later [13].

2.3.4 Management Challenges

As a network grows in complexity, it also becomes more challenging to manage [10, 22]. Furthermore, the dynamic nature of today's networks demands that administrators respond to changes just as quickly [2]. The issue, however, is that traditional networks are rather inflexible due to their distributed control plane, also referred to as vertical integration [1], meaning that they require individual configurations [16]. Despite this requirement, devices in a network cannot be configured in isolation if the goal is for them to work in tandem.

When configuring a device, administrators will often use low-level commands through a Command Line Interface (CLI) [3]. These configurations are stored in the form of configuration files, which can be retrieved from network devices and used (with some changes) to implement similar configurations to another device, or to restore the same device if necessary. The issue of inflexibility arises once more with regards to these configurations files because, as also identified in [1, 3, 22]:

- Configurations files for a network often are dependant on and reference each other, meaning that errors in one file will impact the rest of the network as well.
- Higher number of references between configuration files increases the chance of misconfigurations, as well as the number of steps necessary for making subsequent changes.
- Administrators often use various tools to automate maintaining, troubleshooting, and managing configuration files for their networks. Yet these tools alone are often not enough, and manual changes to configurations files are necessary regardless.

Combined with the aforementioned need for per-device configuration, the issues listed above make the complexity and difficulty of managing a traditional network more apparent.

Another reason behind the challenges network administrators face, is the possibility that network management was underestimated when interconnected network architectures were first designed and implemented. Many solutions used by administrators today, such as automation and troubleshooting tools, have therefore been made as patch and add-on solutions later on, when networks became more complex and unpredicted issues began appearing [3, 23].

2.4 Network Security

As the internet becomes a bigger and bigger part of our daily lives, the importance of securing networks and network traffic also grows rapidly. As we have explained in Section 2.1.3, data travels through networks in the form of segmented packets. Between the source and destination devices for each packet, there is an array of other network devices that packets must travel through, none of which the sender nor the receiver has an overview or control of [8]. User devices can be exposed to harmful programs (malware), an unwanted party could be spying on the packets being sent across the network (some of which may include sensitive information), and the network itself can be crippled by an attack on the infrastructure itself (Denial of Service) [10, 19].

This poses some important questions:

1. How can we ensure no one else but the intended recipient can access the data?
2. How can we ensure that the data received at the destination is the same one sent by the source?
3. How do we make sure that the network is working as intended, so that devices can communicate to begin with?

As we investigate solutions and answers to these questions in the context of networks, we will see that they relate closely to a security concept known as the CIA-triad, in which the three letters stand for Confidentiality, Integrity, and Availability [7].

2.4.1 General Security

One important step in securing a network is securing its physical components [19]. Access to network devices should only be given to authorized personnel, and the devices themselves should be kept in locations where the chances of them getting damaged or stolen are minimized. Device interfaces and features that are not in use should be shut down and disabled as well to prevent unwanted access [24].

Passwords used for accessing and configuring devices, both through the CLI and remote access, should also be encrypted and follow best practices specifically defined for password security. It is particularly important to secure administrators' passwords and accounts, as they have access privileges that could be catastrophic for the network if compromised.

Being able to accurately log and monitor network behavior is also crucial, as detection of abnormal or unexpected issues and responding appropriately in time is a big step in disaster prevention [19].

Simple security principles, however, are not enough on their own for guaranteeing security. There are additional steps network administrators should take and technologies they can utilize for further securing a network.

2.4.2 Access Control and Firewall

As the name suggest, Access Control are policies used for determining who is able to access which resources [19]. In a study on seven different networks [22] (four campus networks and three enterprise ones), it was found that all of them dedicated considerable portions of their router configurations files to defining Access Controls. By implementing Access Controls, network administrators are able to ensure only those with permission are able to communicate across or access certain resources on the network. A set of Access Control policies is called an Access Control List (ACL) [19], which, going forward, is the name we will be using when referring to the entire concept.

ACLs in networks are enforced through packet filtering [10] and in general take two forms: standard and extended - with the difference between the two being the level of granularity used for permitting or denying access. Standard ACLs only filters packets based on their IPv4 sources address, while extended ACLs can filter traffic based on their source and/or destination address, the protocol they are using, and their source and/or destination ports.

As an example, consider a VLAN with its own subnet (as mentioned in sec. 2.2.5) from which we want to limit traffic going outside the network. We can achieve this by implementing simple ACLs that drop all packets coming from any address belonging to that particular subnet, or we can use extended ACLs to fine-tune exactly what kind of traffic we want to allow or deny.

ACLs are also used for implementing Firewalls, which is a layer of protection and separation placed on the point of connection between a network and the outside world [8]. Firewalls are used for ensuring that internal resources requiring to be protected from the outside are inaccessible by those outside the network, as well as keeping certain, perhaps sensitive, traffic inside the network [19]. They come in different forms, such as directly implemented on specialized hardware, or software solutions that can be deployed on various hardware platforms. Modern routers often have some integrated Firewall functionalities [10].

2.4.3 Intrusion Detection and Intrusion Prevention Systems

Intrusion Detection and Intrusion Prevention Systems, shortened to IDS and IPS, are devices implemented behind a network's Firewall, providing higher levels of security by inspecting the behaviour and contents of packets [10]. The main difference between the two is how they handle suspicious packets: IDS will detect them and alert the network administrator, whereas IPS will perform some action on or because of them, for example dropping suspicious packets or dynamically changing firewall rules [19, 7].

The specifics of how intrusions are detected and which methods are used for doing so falls outside the scope of this report. However, we can say that, in general, they use one of two methods: signature detection [10] and misuse detection [19].

In signature detection, the IDS will look for patterns and characteristics it knows to be suspicious or malicious, by checking a database of known patterns and alerting the administrator when it finds a match. Misuse detection, referred by [10] and [7] as anomaly detection, consists of the IDS determining a baseline of what is normal behavior for the network, which it later uses to detect abnormal behavior [19].

2.4.4 Zero Trust Architecture

The security measures and technologies mentioned so far focus primarily on keeping outsiders from entering the network, but it is also important to prepare for scenarios in which the source of danger is within the network.

In a report for Forrester Inc. [25] John Kindervag suggests that "[..] all network traffic is untrusted.", and proposes the Zero Trust model for network architecture. In a Zero Trust architecture, internal users are treated the same as external users with regards to access rights. No one is granted access to resources they do not strictly need - a principle also known as Principle of Least Privilege. All traffic is also verified and logged to generate an audit trail and ensure users are only doing what they are supposed to, rather than simply trusting that they are.

The ultimate goal of this model is to introduce a new way of thinking when it comes to security, and moving away from the notion that threats mostly, or only, originate from outside the network. In addition, Zero Trust emphasises the importance of Role-Based Access Controls (RBAC) whereby users are grouped together and granted access to necessary resources based on their roles rather than individual identity.

2.5 Software Defined Networking

Software Defined Networking (SDN) is a relatively new and growing approach of network architecture, in which the control plane is decoupled (separated) from the data plane [1, 7], as illustrated in Figure 8. This solution transforms network devices into specialized hardware, used primarily for forwarding packets and frames, and gathers control functions in a logically centralized separate entity that communicates with each device. This entity is known as the controller.

Depending on the platform, the physical system for the controller can be either centralized or distributed. Although centralized controllers can be powerful and highly specialized, their biggest downside is that it introduces a single point of failure to the network [1]. A distributed controller, on the other hand, is divided between multiple servers [10], providing an overall better performance, greater scalability, and reliability.

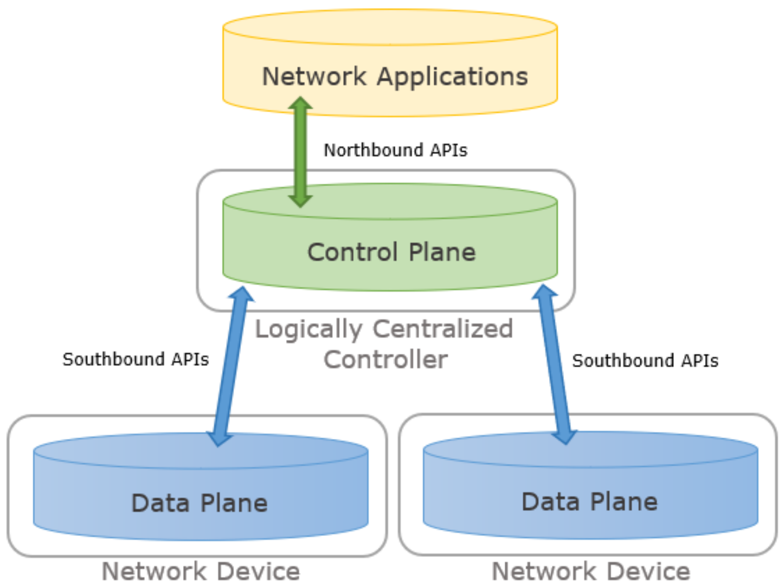


Figure 8: Separation of planes in Software Defined Networking

Other principles of SDN include flow-based forwarding decisions, and a programmable network. The former describes how forwarding devices can match an incoming packet's various header fields against a flow table, defined by the controller, in order to know how to handle it [10]. This means that packets are no longer forwarded based on their destination IP alone, but can also be forwarded based on either source or destination MAC addresses, protocols, or ports.

Network programmability refers to what is, arguably, the most valuable characteristic of SDN [1]: the ability to use software running logically on top of the controller (network applications) to program and control the data plane's behaviour. The applications offer a variety of services such as access control, firewall, monitoring, and others [26].

Communication between network applications and the controller is done via Northbound APIs. Network administrators are also able to access these APIs through scripts in various programming languages, such as Python, C++, and Java [1], instead of any specific network application. Similarly, the communication between the network devices and the controller is defined by Southbound APIs, for which OpenFlow [27] is the most commonly used standard [3].

2.5.1 Software Defined WAN

Software Defined Wide Area Network (SD-WAN) is a software based approach to WANs, allowing organizations to utilize and manage more than one physical network through the same centralized controller system.

We can view an SD-WAN connection between sites as having two main layers: the physical underlying network, and the virtual network overlaid on top of it [20]. The physical network consists of WAN technologies such as connective lines and MPLS (sec. 2.2.7) and is often managed and maintained by the ISP. The virtual layer is the one on which SD-WAN operates, establishing one or more virtual connections atop the underlying network.

SD-WAN, in many ways, resembles a traditional VPN connection between sites, as the overlaying virtual connections function in much of the same way as VPN tunnels do, and the data sent over them is encrypted using technologies such as IPsec. Their biggest difference lies in the unified management the SD-WAN controller provides, making the process of establishing and maintaining such connections between networks more simple and flexible. This had led companies like Google to adopt an SD-WAN solution for their data centers [28], lowering their operational costs whilst increasing their bandwidth utilization.

2.5.2 SDN Management

As discussed in Section 2.3, the issues administrators face in network management are largely due to the distributed control plane and the rather rigid [2] architecture of traditional networks. We also mentioned that administrators have adapted by creating ad-hoc solutions to help face these challenges.

In the SDN architecture, network management is improved by the separation of the control and data planes, because administrators no longer need to configure and communicate with each device individually. This separation means that the controller must, among other things, automatically discover the network devices in order to communicate with them, performing network discovery without the administrator's involvement [29]. Monitoring is also made easier, as the controller has a global overview of the network [3] and is in a constant communication loop with the underlying devices.

Additionally, the aforementioned network applications and APIs at administrators' disposal provide more dynamic and flexible management options, as they eliminate the need for using low-level languages. Instead, they provide an abstraction layer by using higher level programming languages [1], making it the controller's responsibility to then translate configurations (via the southbound interface) into the low-level languages the devices will understand. High-level languages in this context are also less error-prone, decreasing the chance of misconfigurations [30].

The programmability of the network also raises its automation potential, allowing administrators to more easily and quickly respond to changes, adjust network behavior as necessary [3], and scale the network more seamlessly [1].

2.5.3 SDN Security

Although SDN can improve network security with the many benefits it adds to network management, as it is with the introduction of any new technology or architecture, it also raises some security questions of its own [30]. The most important of which is whether or not the centralized controller could act as a single point of failure for the entire network, either by being a centralized system itself, or by being the main target of attacks given its importance [26].

The Northbound APIs and network applications running atop the controller introduces the possibility of a code vulnerability or bug causing a security breach into the controller and network. The Southbound APIs could also be targeted by attackers, both for gaining access to the controller and for listening in on or disrupting communication with the network devices.

A compromised controller is detrimental to an SDN environment because it would grant malicious actors control over the entire network. This could lead to eavesdropping and data theft [1], or the launch of Denial of Service attacks on the network [30]. Moreover, the spread of malicious software in the network would be faster and more catastrophic

than in a traditional network, because the centralized and highly programmable control plane would allow malware to attack an entire network from a single point.

Proposed solutions to these issues are not dissimilar to those proposed for traditional networks (sec. 2.4); Firewalls and IPS for traffic filtering and dropping potentially dangerous packets, IDS for anomaly and attack detection, Access Controls for user authentication and access authorization, and physical security for the servers used by the controller system [1, 30].

3 Methodology

The aim of our research is to compare the ease of use and competence required to set up and manage traditional legacy network solutions compared to Cisco Meraki's SDN solution. To achieve this, we designed two use-cases to simulate and set up single-branch and branch-to-branch connections using Packet Tracer and Cisco Meraki.

We evaluated the ease of implementation for each solution by completing a set of tasks outlined in the use-cases, which were used as concrete points of comparison between the two solutions for empirical analysis. Our final evaluation also takes into account the automation potential of both network types, and how it can be used to improve and simplify network management and security.

In this chapter, we begin by providing a description of the equipment used throughout our practical experiments (sec. 3.1). We then define the criteria we used to evaluate ease of setup and configuration (sec. 3.2.1), ease of management (sec. 3.2.2), as well as the criterion for evaluating automation potential (sec. 3.2.3).

Next, we present the two use-cases we developed to simulate the practical implementation of each solution, after which we describe the setup process for each use-case using traditional devices first and then Cisco Meraki's SDN solutions. Finally, we assess the automation potential of each solution, providing examples where relevant.

3.1 Equipment and Materials

3.1.1 Cisco Packet Tracer

Due to practical considerations which led to a lack of access to physical traditional legacy devices, we have chosen to use Cisco Packet Tracer v8.2.0 as an alternative to emulate the configuration of LANs and WANs according to our use-cases.

Cisco Packet Tracer is a network simulation tool that allows for designing, configuring, and troubleshooting network setups in a virtual environment. The tool provides a user-friendly graphical interface with drag and drop functionalities to add networking devices, create connections between them, and simulate network traffic.

However, it is also important to keep in mind that Packet Tracer is not able to simulate every feature and functionality of a physical device. This could result in discrepancies between the simulated modules and their physical counterparts in terms of performance and behavior, especially when it comes to how long it takes to synchronize configurations. Nonetheless, since the configuration process of the simulated network devices closely resembles that of real-life traditional network devices, we can effectively use it to simulate and analyze the ease of which traditional network devices can be configured.

3.1.2 Cisco Meraki Stack

Meraki is a family of products produced and distributed by Cisco that is, according to them, developed with ease of use, accessibility, and scalability in mind. It is an SDN solution built around a cloud platform, where networks and their devices are configured through the use of a browser-based dashboard, or by the use of an API.

The Meraki devices are all built with zero-touch provisioning as a central feature, where a device only needs to be registered to the organization and powered up in order to be configurable through the cloud. For this report, we will be using the three following devices, with their respective licenses, as provided by Cisco Meraki: MX67C-WW Security

Appliance (SA), MS120-8LP Switch, and MR36 Wireless Access Point (WAP). Figure 9 shows these devices and how they are interconnected.

Figure 9: The Meraki equipment stack

MX67C-WW

The MX67C security appliance/LTE router is a router which doubles as a security appliance that offers enterprise-class security features under the Unified Threat Management (UTM) term, which includes functions such as content filtering, Next Generation Firewall (NGFW), and intrusion detection [31]. Its range of features includes advanced threat protection, secure remote access, and zero-touch site-to-site VPN connectivity.

Key design properties are the connection points in the form of one RJ45 WAN connection port, four RJ45 ports for LAN purposes, and a cellular connection for WAN which can also be used as an LTE failover connection.

MS120-8LP

The MS120-8LP switch is a compact Layer 2 switch that supplies eight LAN ports, two Small Form-factor Pluggable (SFP) ports, and Power over Ethernet (PoE) [32]. The PoE capable ports are intended to power other Meraki devices such as access points, cameras, sensors, and other IoT devices. The device supports DHCP snooping and relay for detecting rouge DHCP servers, and forwarding of DHCP services.

Each port on the device can be configured individually, or combined, allowing for quick implementation of mapping, security, and loop protection options such as Rapid Spanning Tree Protocol (RSTP), Spanning Tree Protocol (STP), port isolation, port type, assigned VLANs, and Quality of Service (QoS).

MR36-AP

The MR36 is a wireless access point which offers wireless access service using Wi-Fi 6 technologies [33]. The MR36 is connected to the network and powered through a single RJ45 PoE port, and can offer up to 15 SSIDs simultaneously, each with individual settings. Wireless access is provided by 2.4 and 5 GHz radios, enabling fast and reliable network at different ranges.

The device offers security features to authenticate users for network connection, scanning of rogue SSIDs, spoof detection, detection of other malicious activity, and traffic shaping/firewall protection which can do Layer 7 inspection and content restriction.

3.1.3 Cisco Meraki Dashboard

The Meraki Dashboard is a graphical user interface accessible via a web-browser, through which an administrator can manage the organization, its networks, the devices within those networks, and analyze end-device traffic and connections. The dashboard is one of two ways that administrators can interact with the Meraki cloud, the other one being the Meraki Dashboard API.

As Figure 10 shows, the dashboard consists of a taskbar on the left-hand side and a search bar at the top of the screen. The taskbar consists of the different aspects of the managed networks, which are organized by the separation of roles and subjects of interest. Under each subject tab, there are relevant monitoring and configuration functions.

While it is possible to navigate through all available functions, configurations, and info-screens through the taskbar, it is also possible to use the search bar to look for specific functions or relevant Meraki documentation and threads.

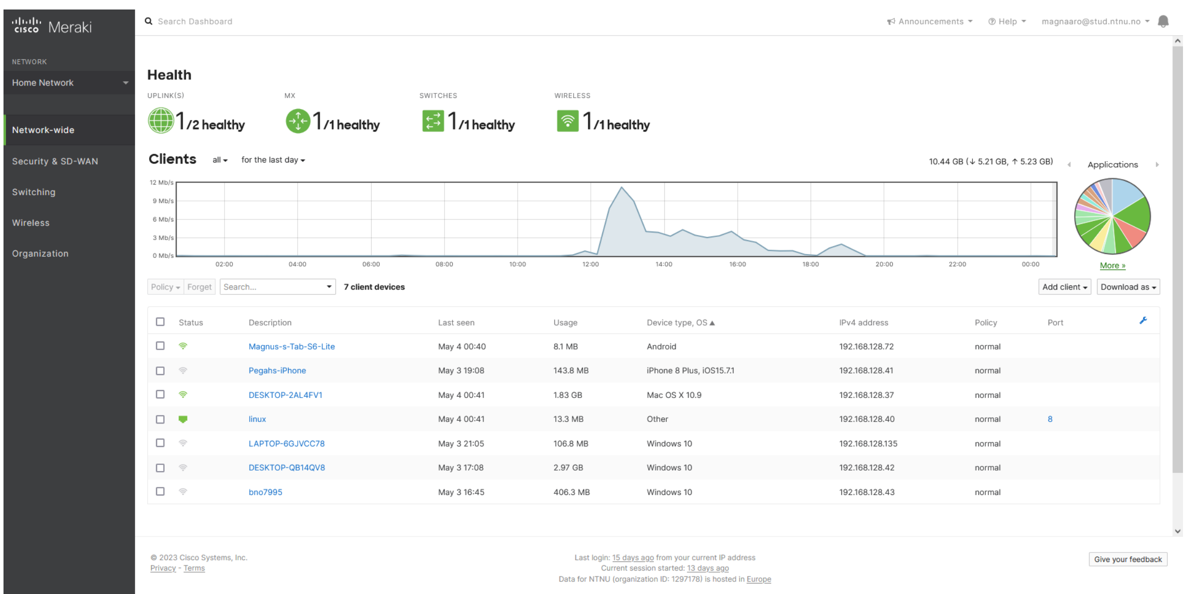


Figure 10: The Meraki dashboard

3.2 Criteria of Analysis

To evaluate and compare the ease of implementation of a Cisco Meraki-based network with a traditional legacy network for an SMB with limited expertise, we first decided on the three aspects we wanted to focus on when drawing a comparison between the two networks, and defined various measures for evaluating each of them. Next, we devised the two use-case scenarios described in Sections 3.3.1 and 3.3.4.

3.2.1 Ease of Configuration and Setup

For each use-case, we defined a set of features that needed to be configured based on the needs of each use-case, and, from this point forward, we will refer to the implementation of each feature as a *task*.

To measure the ease of each task, we had to define what *ease* meant in this context. We achieved this by deciding on several factors that determine the ease with which a task is done, and sorted them into the following categories of assessment: execution, transparency, success, duration, and errors. We then used our own prior experiences to devise a set of questions with yes-or-no answers related to each category, and created the rubric in Table 2.

Execution

Execution covers the process of performing the task. The questions are meant for gaining a better understanding of how the task was done, what was necessary for its completion, and whether any documentation or guidance was available. They focus on factors that could potentially reduce the ease of a given task, such as uncertainty on how it is meant to be done resulting in having to seek out solutions by others, unintuitiveness or lack of information on the interface used for executing the task (device CLI or Meraki dashboard), and lack of available official documentation or one without adequate information.

Transparency

Questions about Transparency are meant for understanding whether there was insight into the state of the device as a task was being performed. The aim is to see if any uncertainties arose due to a lack of feedback confirming a task's execution, because feedbacks helps understand what is happening which, in turn, makes performing a task somewhat less challenging.

Feedback here refers not only to written messages, but any indicators, such as visual cues, conveying that a change took place. A false positive refers to when there was an indication that a change has been made when in reality, it did not go through, potentially leading to other issues.

Success and Duration

Success is about the completion of a task and the time it took to do so. The goal of these questions are to understand whether the time it took to complete a task was proportional to its complexity, or if other issues occurred which prolonged the time it took to complete the task. Duration functions as a follow-up category, although it focuses more on the total amount of time spent on a task versus the time spent on things like troubleshooting.

Errors

Lastly, we took into account the possibility that various errors could occur when performing a task, and formulated questions to use in those cases. Errors focuses on the ease with which an error could be discovered, and whether there were mechanisms to stop a faulty configuration from affecting the rest of the network.

Category	Nr	Question	Yes	No
Execution	1.1	Was seeking out an external source necessary for completion?	0	1
	1.2	Was physical access to a device necessary for execution?	0	1
	1.3	Were you provided with any guidance/instructions where the device was configured?	1	0
	1.4	Was it clear where you needed to execute the task?	1	0
	1.5	Was there any official documentation available to you?	1	0
	1.6	*If yes to above:* Was the documentation's information sufficient for completing this task?	0	-1
Transparency	2.1	Were you given feedback throughout the of this task?	1	0
	2.2	*If yes to above:* Did the feedback provide information about the state of the device?	1	0
	2.3	Did you encounter any false positive feedback?	0	1
Success	3.1	Could you verify your success in completing the task?	1	0
	3.2	Was the task completed in a timely manner?	1	0
Duration	4.1	Was the majority of your time spent on actually executing the task?	1	0
Errors	5.1	Were you given feedback about the error or misconfiguration?	0	-1
	5.2	Was the feedback sufficient for identifying the source of the error?	0	-1
	5.3	Was the configuration executed with the error present?	-1	0

Table 2: Questions on ease of configuration

After each task's completion (which we define as when the feature was implemented), we used the rubric to make a qualitative assessment of the task's ease. We did so by using the binary nature of the yes-or-no answers to create a scoring system, and used the results (sec. 4.1) to determine each task's overall ease. This scoring system is not based on predefined metrics, and are only meant to assist us in conducting a high-level comparative assessment.

For the first five categories, a single point is added to the total based on whether or not a yes-answer counted as a positive contribution to the task's ease. A similar logic is applied to the no-answers as well, with the exception of the follow-up question in Execution, which removes a point from the total if answered no. The reason for this decision is our belief that inadequate documentation voids the documentation's overall contribution. Finally, given that questions in Errors were only meant to be answered in the case of an error occurring, we decided that, if answered, they should deduct up to three points from the total score.

Based on this system, the best case scenario results in a total of 11 points, whereas the worst case scenario results in 0 points. In case of errors, the score is calculated with the deducted points in mind, making the scores for the best and worst case scenarios 11 and -3 respectively. The general format of the total score can hence be described as $Total = N - m$, where $N = [0, 11]$ and $m = [-3, 0]$.

The best and worst case scenarios described above can be found in Appendix A.3.

3.2.2 Ease of Management

After the completion of all tasks, we compare the different methods available to each network solution for the purposes of monitoring, maintenance, and management. We take a simpler approach to this criterion, and focus on the following questions:

- What types of tools or protocols are available for viewing the state of the devices on the network?
- Was it easy or intuitive to retrieve and view information about the network devices using built-in user interfaces?
- Did the tools provide accurate insight into the network devices? Were there any false positives e.g. was there an undetected issue with the network?

Answers to these questions are not measured in the same way as those in Section 3.2.1, but are used to draw direct comparisons between the two network solutions. This is because the ability and methods available to manage the network partly depends on what features and functionalities the network has been configured with.

3.2.3 Automation Potential

Our third criterion of analysis is focused on our second research question, which explores and compares the automation potential of a Cisco Meraki based network to that of a traditional one. We use the following questions for evaluating this:

- Is it possible to automate a task with this solution?
- Which tools or resources are available for doing so?
- What is the extent of the automation? Which types of tasks can or cannot be automated?

3.3 Use Cases

The following two use-cases were used to evaluate the ease of setting up and configuring a LAN and WAN for a fictional SMB with limited IT knowledge and network expertise. It should be noted that each network's configuration varies based on each company's needs. For our use-cases, we have chosen to present the simplest version of a network and implemented configurations which provided basic functionality we deemed necessary for our case. By designing the network architecture for each use-case with traditional legacy devices and Cisco Meraki devices, we can provide practical examples of how each solution can be implemented in real-world scenarios.

This approach also allows us to compare the practical implications of both solutions, and assess their effectiveness in meeting the specific needs of each use-case. Ultimately, this approach should offer a comprehensive analysis of the strengths and weaknesses of each network solution based on our observations and experiences (discussed further in sec. 4.1 and 5.1).

3.3.1 Use Case 1: Single-branch connectivity (LAN)

This first use-case explores how to design and configure a simple LAN with simple internet connectivity for a small start-up consulting company named Stooges. For this use-case, we have assumed that the routes between the ISP router and Stooges' edge router are configured and handled by the ISP.

Stooges' office is located in Trondheim and consists of 8 employees. Their IT-administrators' prior network experience comes from CCNA training, and they have been tasked with designing and configuring their network.

The company has decided to use the 192.168.128.0 /24 private IP address block. To make network management easier, the network administrators devised an addressing plan which divides the chosen address block into smaller subnets for each VLAN as shown in Table 3.

Name	VLAN	IPv4	Default Gateway
MANAGEMENT	10	192.168.128.0 /27	192.168.128.1
IT-ADMIN	20	192.168.128.32 /27	192.168.128.33
EMPLOYEES	30	192.168.128.64 /26	192.168.128.65
GUESTS	40	192.168.128.128 /25	192.168.128.129

Table 3: Trondheim VLAN subnet plan

The company wishes to provide both wired and wireless connectivity for their employees. However, for security reasons, they would only like to provide controlled wireless connectivity for their occasional guests at the office such as potential clients and partners.

Their guests should not be able to communicate with other hosts within the company's private network, but should still be able to access the internet. Since Stooges is operating on their own with a single office, and it is assumed that all employees are physically present at the office to do their work, VPNs are not required, and the following network features are sufficient:

- VLANs to separate network traffic between IT-administrators, employees, and guests, providing more control over network resources for each group.
- DHCP to automatically allocate IPv4 addresses to hosts in each VLAN, simplifying network management and reducing configuration errors.
- Standard ACLs to secure the network devices from unauthorized remote access, reducing the risk of network security breaches.
- Extended ACLs to isolate the GUEST VLAN's traffic from the rest of the network to ensure network performance and security.
- NAT to allow hosts within the company's internal network to communicate with other devices and services on the internet.
- Basic logging and monitoring capabilities, intended to improve network availability and provide insight into the network.

3.3.2 Single-branch Connectivity (LAN) with Packet Tracer

To simulate the setup of Stooges' LAN using traditional legacy devices, we have chosen to use the 2901 router, 2960 switch, AP-PT, and Server-PT modules in Cisco Packet Tracer. These devices are interconnected as shown in Figure 11.

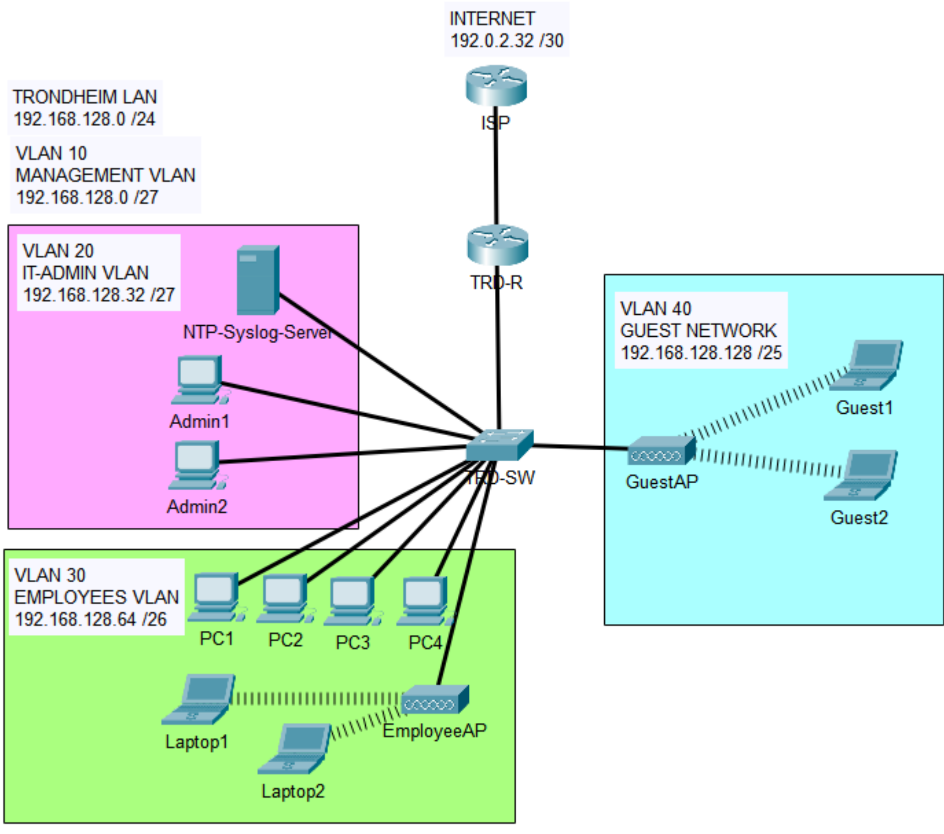


Figure 11: Packet Tracer - Single-branch LAN topology

Simulating the Internet

As mentioned in the use case description, in a typical LAN setup with internet access, the routing between the LAN's edge router and the internet would be handled by Stooges' ISP. However, in our simulated Packet Tracer environment, we do not have access to an actual ISP or internet connection.

To simulate internet access, we manually configured one of the 2901 router modules as an ISP router in order to emulate the connectivity to an ISP providing internet access to Stooges' network. The commands used to configure the routers and routes between them can be found in Appendix A.1. The ISP router module, labeled as "Internet" in Figure 11 was configured with the IP addresses shown in Table 4.

Name	**IPv4**
ISP-TRD WAN	**192.0.2.32 /30**
ISP g0/0	192.0.2.33
TRD-R g0/0	192.0.2.34

Table 4: LAN Internet IP allocations

Addressing Plan

The addressing plan for Stooges' simulated single-branch LAN is shown in Table 5, which lists all the subnets within the LAN and the IP addresses of each sub-interface on the Trondheim router.

Name	**IPv4**
Trondheim LAN	**192.168.128.0 /24**
TRD-R g0/1.10	192.168.128.1
TRD-R g0/1.20	192.168.128.33
TRD-R g0/1.30	192.168.128.65
TRD-R g0/1.40	192.168.128.129

Table 5: Stooges' LAN addressing plan

Stooges' Trondheim network uses a 2960 switch with twenty-four FastEthernet ports. Of those twenty-four, sixteen of the interfaces have been assigned a VLAN as described in Table 6, such that a host connected to a specific port will be connected to the port's corresponding VLAN.

The Employee and Guest access points are connected to switch interfaces associated with VLAN 30 and VLAN 40 respectively. This enables the separation of wireless guest traffic from wireless employee traffic. To prevent unauthorized access and configurations, the unused interfaces on all network devices are administratively shut down.

Name	**VLAN ID**	**Switch Interface**
IT-ADMIN	VLAN 20	FastEthernet 0/1-4
EMPLOYEES	VLAN 30	FastEthernet 0/5-15
GUESTS	VLAN 40	FastEthernet 0/16

Table 6: VLAN to switchport mapping on TRD-SW

Basic Device Configuration

The first task is to complete the basic device configurations for the router and switch. This consists of naming each device to distinguish it from others, securing CLIs, and enabling secure remote access.

By using the `hostname` command, we set the name of the router and switch as 'TRD-R' and 'TRD-SW'. To secure the CLI, we configured passwords for privileged mode access, secured the console line on both devices, and set a timeout to ensure the session is terminated after five minutes of inactivity using the following commands:

```
enable secret ba114
line con 0
  password cisco
  login
  exec-timeout 5
  exit
service password-encryption
security password min-length 5
```

This ensures that only authorized individuals are able to execute configuration and management commands on the device. On the router, we also ran the following command to block login attempts for five minutes (300 seconds) after three failed attempts within 60 seconds, to prevent brute-force password guessing attacks.

```
login block-for 300 attempts 3 within 60
```

After configuring unique hostnames, SSH can be set up on the router and switch to allow for remote access. To accomplish this, an IP domain name must be configured, and an RSA key pair must be generated.

In our configuration, we opted to create a 2048-bit key when prompted by the terminal for maximum security. Once the key was generated, an administrator user with the password "3ggsamm1ch" was established.

The final step is to activate the SSH protocol on the vty, i.e., virtual terminal, lines of the devices. The following commands were used to complete this configuration:

```
ip domain-name cisco.com
crypto key generate rsa
username admin secret 3ggsamm1ch
ip ssh version 2
line vty 0 15
  transport input ssh
  login local
  exec-timeout 5
  exit
```

The AP modules used within the network does not have its own CLI, but it does have a 'Config' tab which we used to set an SSID name and password for the EmployeeAP as shown in Figure 12. The GuestAP has been configured in a similar way, with the exception of the Authentication option being set to 'Disabled'.

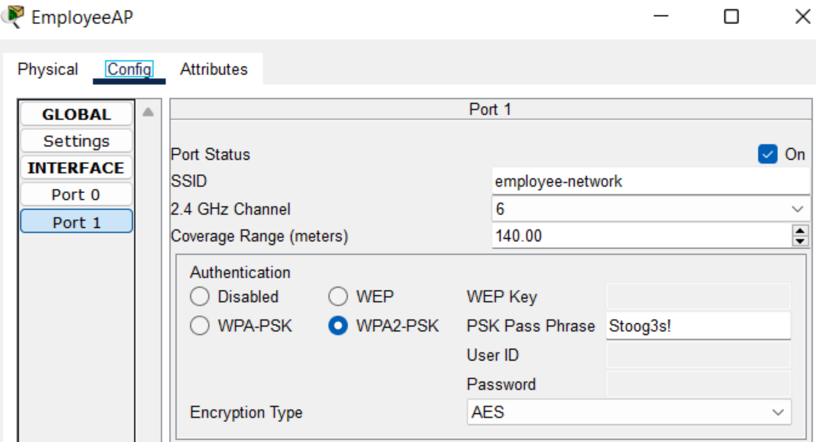


Figure 12: Employee WiFi AP Configuration

VLAN Configuration

VLAN configuration for this setup involved creating, naming, and assigning Stooges' different VLANs to switchport interfaces as outlined in Table 6.

The commands below illustrates how VLAN 20 was configured as an example. Similar commands were used to configure VLANs 10, 30, and 40, but with the appropriate values changed accordingly.

```
vlan 20
name IT-Admin
exit

int range fa0/1-5
switchport mode access
switchport access vlan 20
exit
```

To ensure that traffic from VLANs 10, 20, 30, and 40 can reach the router, the switch's G0/1 interface, which connects to the router, was configured as a Trunk using the commands below. This allows the interface to carry traffic from multiple VLANs simultaneously, and should also be done on the router's G0/1 interface.

```
int Gig 0/1
switchport mode trunk
switchport trunk native vlan 99
switchport trunk allowed vlan 10,20,30,40
end
```

Once the VLANs have been established on the switch, the next step is to allow for inter-VLAN routing. This involves the creation of sub-interfaces which will be assigned the IP address of each VLAN's default gateway and tagged with a VLAN ID.

Doing this allows the router to route traffic between VLANs, making it possible for hosts on the IT-admin VLAN to communicate with hosts on the Employee VLAN. The following commands were used to create a sub-interface for VLAN 20's default gateway. Similar commands were used for the remaining VLANs, but with the necessary values changed accordingly.

```
int G0/1.20
description DGW for VLAN 20 - TRD-IT-Admin
encapsulation dot1Q 20
ip add 192.168.128.33 255.255.255.224
exit
```

DHCP Configuration

With VLANs and inter-VLAN routing configured, the next step is to configure DHCP pools to automatically assign IP addresses to hosts depending on which VLAN they are connected to.

For this, we have chosen to configure the router as a DHCP server and made sure to exclude the first 5 IP addresses from VLAN 20 to avoid addressing conflict with potential hosts that have been statically assigned an IP address, such as the NTP-syslog server, using the following command:

```
ip dhcp excluded-address 192.168.128.33 192.168.128.38
```

To configure DHCP pools, we need to set a name, define the subnet that will be assigned IP addresses, and assign the subnet's default gateway as the pool's default router. Using VLAN 20's DHCP pool configuration as an example, we used to following commands:

```
ip dhcp pool TRD-IT-ADMIN
network 192.168.128.32 255.255.255.224
default-router 192.168.128.33
exit
```

Verification of proper DHCP configuration can be done by enabling DHCP on any host, connecting it to a switchport which has an assigned VLAN, and checking what its IP address and default gateway is. The following screenshot is an example of a host that is connected to the switch's fa0/2 port which belongs to VLAN 20. As shown in Figure 13, the host has received the IP address of 192.168.128.40, which is from the TRD-IT-Admin's DHCP pool.

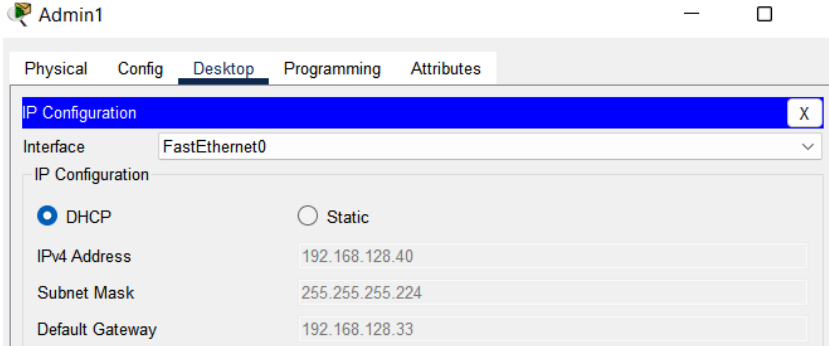


Figure 13: DHCP IP allocation

NAT Configuration

Once basic device configurations, VLAN, and DHCP has been successfully implemented, the subnets within the LAN should have internal connectivity and be able to successfully ping each other. However, since Stooges' LAN is using private addresses, hosts on their LAN are unable to communicate with public hosts on the internet such as the Internet router.

To provide them with internet access, we used the following commands to configure Port Address Translation (PAT), also known as NAT overload, on Stooges' edge router, TRD-R, to translate private addresses within their LAN's subnet to a public address, allowing for internet access:

```
ip nat inside source list 1 interface Serial0/0/0 overload
ip access-list 1 permit 192.168.128.0 0.0.0.255
int g0/1.20
ip nat inside
    exit
int g0/0
    ip nat outside
    exit
```

Once NAT is set up, the hosts within Stooges' internal network should be able to communicate with public hosts outside of Stooges' LAN.

We can verify this by checking whether it can successfully ping the ISP router at 192.0.2.33. As shown in Figure 14, PC1 which is connected to Stooges' Employee VLAN receives responses from the ISP router.

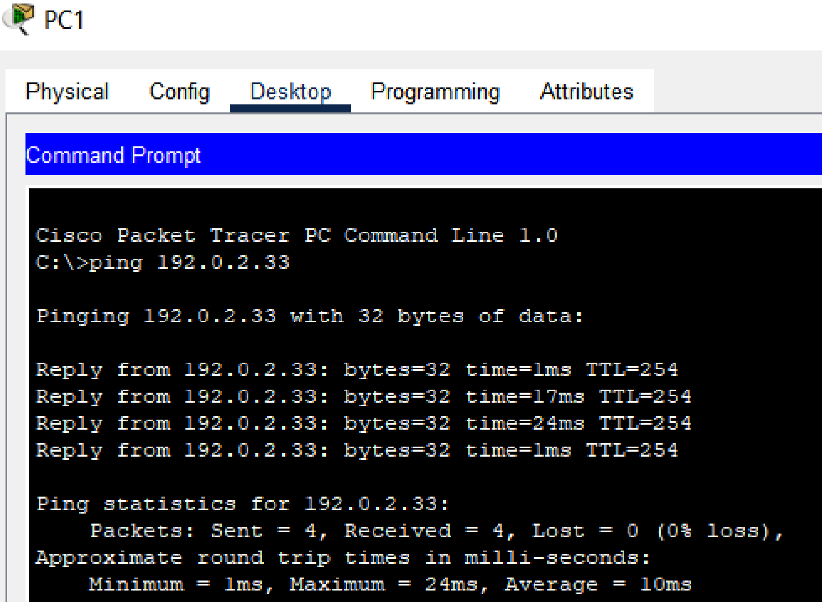


Figure 14: ISP ping response

ACLs

To control the traffic flow within Stooges' network, we configured ACLs for Stooges' single-branch LAN setup. Since we previously configured the network devices to allow for remote access, we now need to implement ACLs to ensure that only hosts within the IT-admin subnet are able to access the network devices remotely through SSH.

One way to do this is to only allow traffic to the Management VLAN from the IT-Admin VLAN. Since this policy would be defined based on the source's IP address, we can create a standard ACL which allows traffic from the IT-Admin VLAN by running the following commands on TRD-R:

```
ip access-list standard MANAGEMENT-INBOUND
    remark allow ip traffic from IT-VLAN
    permit 192.168.128.32 0.0.0.31
```

We would then make sure to apply the ACL to the router and switch's vty lines.

```
line vty 0 15
    ip access-class MANAGEMENT-INBOUND in
    end
```

To test the implementation of our standard ACL, we can attempt to SSH into the router through Admin1's PC as shown in Figure 15. Once the IP address of the router and user is inputted, a session is established and the admin user's password is required to go any further.

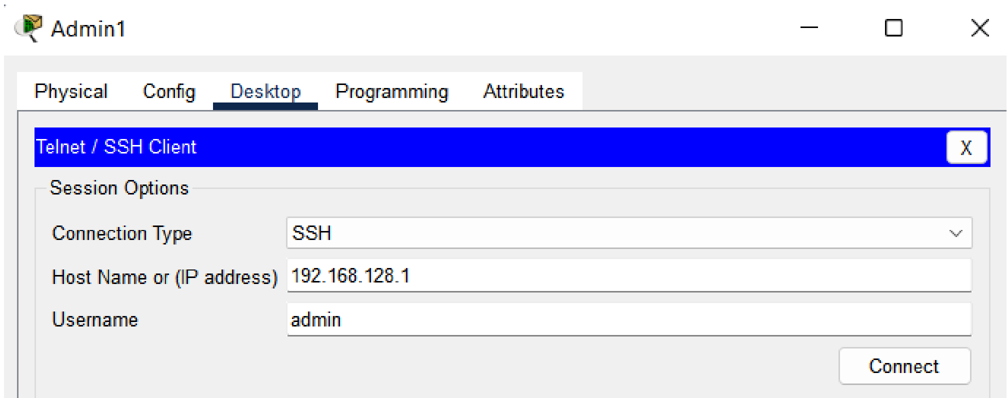


Figure 15: Establishing SSH connection to TRD-R

Inputting the user's password successfully provides the user with remote access to the TRD-R router, and the network administrator is able to run configurations remotely through the SSH client as shown in Figure 16.

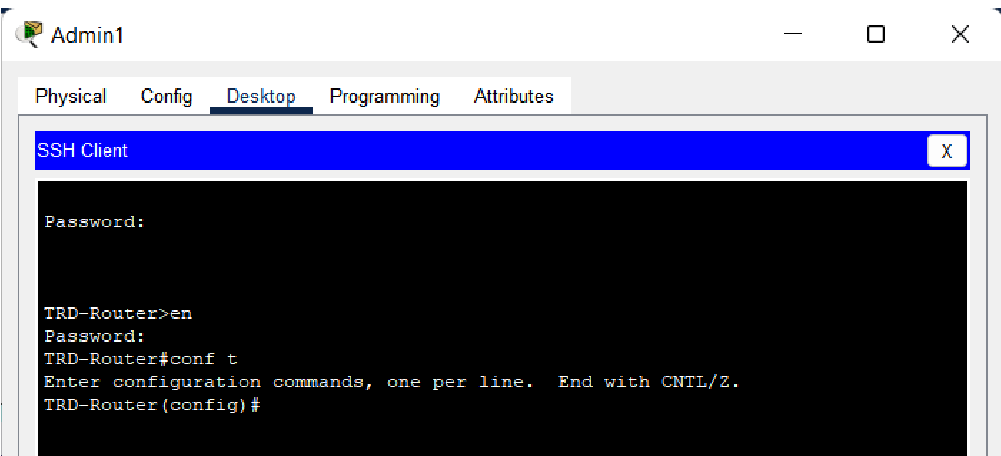


Figure 16: Entering global configuration mode using SSH client

To test if the ACL blocks traffic from any other VLAN than IT-Admin, we can try SSH into the router from PC1, which is connected to the Employee VLAN. As shown in Figure 17, no session is established, preventing remote access from PC1 into TRD-R.

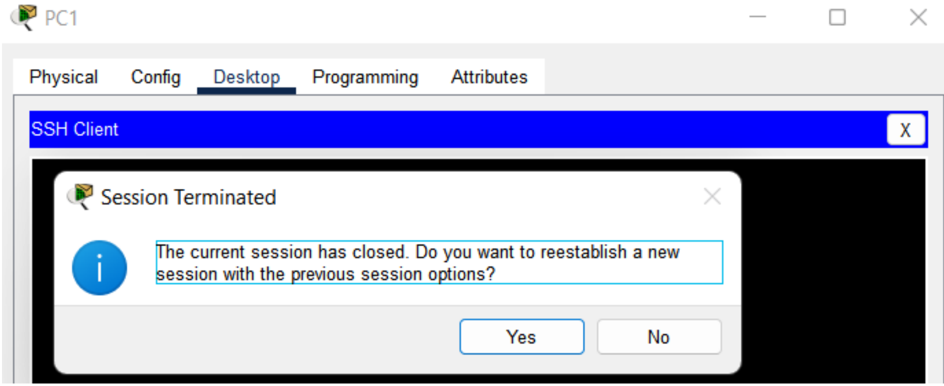


Figure 17: Rejected SSH session

Since we would also like to prevent the Guest network from communicating with Stooges' internal network, we need to set up an extended ACL. Unlike a standard ACL, an extended ACL allows us to specify both the source and destination of the traffic.

In this case, we want to block Guest traffic from accessing Stooges' internal network, while still allowing Guest traffic access to addresses outside of Stooges' LAN. To accomplish this, we defined an extended ACL called "GUEST-IN" and applied it as an inbound ACL on the g0/1.40 sub-interface since it handles the routing for the Guest VLAN.

By applying the ACL on this sub-interface, we ensure that any traffic coming into the Guest VLAN is checked against the ACL before it is allowed to enter the internal network. The commands used to define and apply the ACL are shown below:

```
ip access-list extended GUEST-IN
    remark Permit DHCP
    permit udp any eq bootpc any eq bootps
    remark Permit traffic to the ISP or internet
    permit ip 192.168.128.128 0.0.0.127 192.0.2.0 0.0.0.255
    remark Deny any other traffic to company private networks
    deny ip 192.168.128.128 0.0.0.127 192.168.128.0 0.0.0.255
    exit

int g0/1.40
    ip access-class GUEST-IN in
```

To verify the extended ACL works as intended, we can attempt to ping the IPv4 address of a host within Stooges' LAN, such as PC1 in the Employee VLAN from a host connected to the Guest VLAN such as Guest1.

As shown in the Figure 18, we receive the response "Destination host unreachable." when trying to ping PC1's IPv4 address at 192.168.128.67. However, when we use Guest1 to ping the ISP router at 192.0.2.33, we still receive a response.

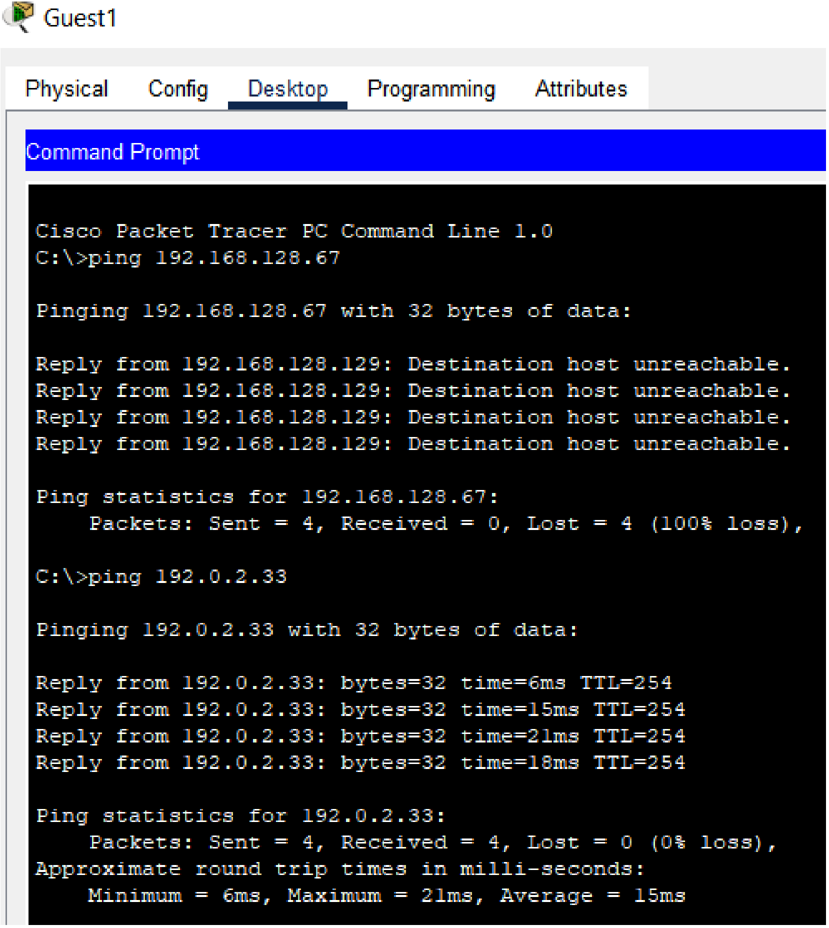


Figure 18: GUEST-IN ACL verification

Monitoring Configurations

The final configuration for this use case is to set up NTP, syslog, SNMP and Netflow to allow for network monitoring and troubleshooting. We added a web-server module (labelled as NTP-syslog server in Figure 11) to the network and assigned it a static IP address of 192.168.128.34. Using the following NTP command on each network device, we synchronized their clocks to the time on the NTP server.

```
ntp server 192.168.128.34
```

To confirm that NTP has been successfully configured, we ran `show ntp status` to view the status of the network device's synchronization with the NTP server. Once properly synchronized, the output should resemble that which is shown in Figure 19.

```
TRD-R#sh ntp status
Clock is synchronized, stratum 2, reference is 192.168.128.34
nominal freq is 250.0000 Hz, actual freq is 249.9990 Hz, precision is 2**24
reference time is E7D2D9E7.000003AD (16:27:19.941 UTC Mon May 1 2023)
clock offset is 0.00 msec, root delay is 0.00  msec
root dispersion is 19.62 msec, peer dispersion is 0.47 msec.
loopfilter state is 'CTRL' (Normal Controlled Loop), drift is - 0.000001193
s/s system poll interval is 6, last update was 2 sec ago.
```

Figure 19: TRD-R NTP status

To set up for syslog, we ran the following commands on each network device to add timestamps to generated log messages and set the IP address of the device to which the logs will be sent to. In this case, the logs are to be sent to the NTP-syslog server module with the static IP address of 192.168.128.34.

```
service timestamps log datetime msec
logging 192.168.128.34
```

As mentioned in Section 2.3.2, SNMP can be used to send both get- and set-requests to an SNMP agent. Set-requests would allow the network administrators to remotely configure devices with SNMP enabled through the MIB browsers on their end-devices. However, seeing as we have already set up SSH for secure remote connections to the network devices, and it is easier to configure devices using the CLI, we have decided to prevent the use of set-requests within our network by omitting the configuration of an SNMP read and write community string on Stooge's network devices. Doing this also ensures that only users on the IT-ADMIN VLAN can make changes to Stooge's network devices, since remote access via SSH has been restricted using the MANAGEMENT-INBOUND ACL we configured in Section 3.3.2.

As we would still like to use SNMP get-requests to retrieve information about the state of a device, such as its performance, status, and other details, we used the following command to set the read only community string on Stooges' network devices.

```
snmp community TRD ro
```

In addition to SNMP, NTP and syslog, we implemented Netflow for the monitoring IP traffic flows throughout Stooge's network by running the commands below on the router to specify where the IP traffic flow data should be sent to using port 2055 before running `ip flow ingress` on each of its interfaces:

```
ip flow-export destination 192.168.128.34 2055
ip flow-export version 9
```

3.3.3 Single-branch Connectivity (LAN) with Cisco Meraki

The first step to implementing the Stooges' LAN using Cisco Meraki was to create user accounts necessary for accessing the Meraki dashboard and managing the network and its devices. The Meraki devices, as described in Section 3.1.2, were plugged in a power source and interconnected with each other using Ethernet cables. Figure 20 displays the network topology that was created.

Figure 20: Meraki Single-branch LAN topology

To make configurations to the devices, it was also necessary to connect them to the Meraki cloud via connecting them to the internet. Access to the internet was provided by connecting the MX router to a wall outlet belonging to the the underlying network. The router then provided internet connectivity for the other two devices.

Name	VLAN ID	IPv4	Default Gateway
Management	10	192.168.128.0/27	192.168.128.1
IT-Admins	20	192.168.128.32/27	192.168.128.33
Employees	30	192.168.128.64/26	192.168.128.65
Guests	40	192.168.128/25	192.168.128.129

Table 7: Meraki VLANs and addressing plan

Basic Device Configuration

In order for the devices to be configurable and provide connectivity to other end-devices, they first need to be logically placed on the same network. Therefore, we began by creating a network within the Meraki dashboard, named ”Home Network”, and added the three Meraki devices to it. Once they were successfully connected to the internet, the devices automatically fetched a set of firmware upgrades from the Meraki cloud and recognized their interconnectivity within the network we created.

Using the general interface of the dashboard, we then changed each device's hostname accordingly, as depicted in Figure 21. Next, we provided the address of their physical locations to all three devices. Lastly, we changed the name of the WAP's default SSID, and configured basic security by setting and enabling a password for it.

Figure 21: Changing hostname for a device

VLAN Configuration

To establish a VLAN configuration on the network, it was necessary to enable VLANs on the router, and assign a name, VLAN ID, subnet, and gateway to each of them as shown in Table 7. To do this, we navigated to the page for ”Addressing and VLANs” in the dashboard. Here, we first enabled the use of VLANs and then used the addressing plan

in Table 7 to create the VLANS one by one. Figure 22 shows the two steps of creating a VLAN using the Meraki dashboard.

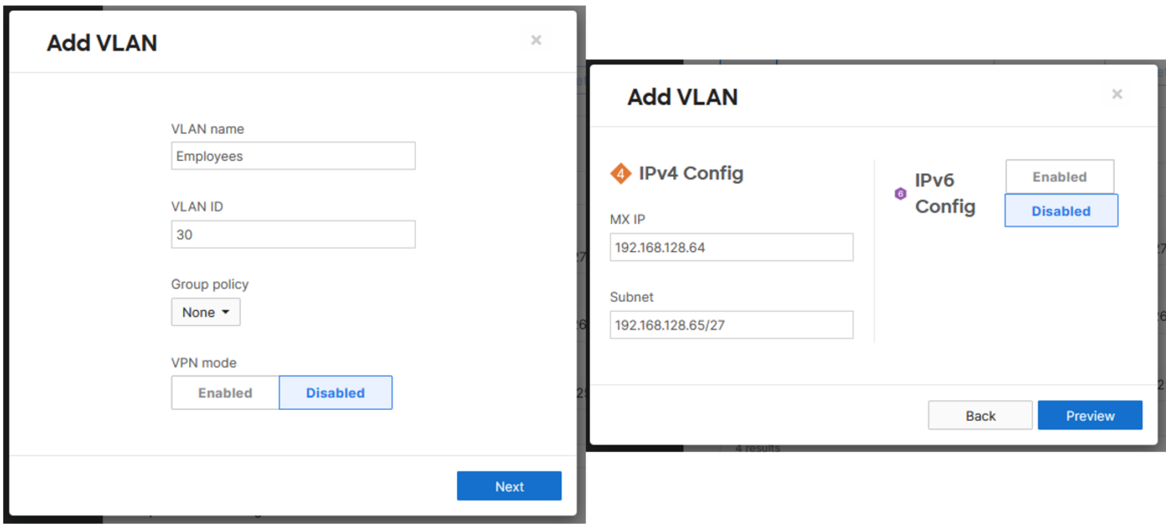


Figure 22: The steps of VLAN creation

To allow for inter-VLAN communication in the network, we also configured the router ports under the same section. The port connecting the MS switch to the router was labeled as a trunk connection that allowed all VLANs. A trunk label was also applied to the port connection between the switch and the WAP. Lastly, we specified that the router should drop all untagged packets.

We assigned the switch and access point addresses in the management VLAN, by going into the page for each individual device, and editing its IP settings to accept an address via ”DHCP” from VLAN 10.

DHCP Configuration

By default, both the MX router and WAP provided DHCP and NAT functionalities to wired and wireless connections respectively. The WAP, however, distributed addresses in the 10.0.0.0/8 range, which is not in the addressing plan used for this use-case.

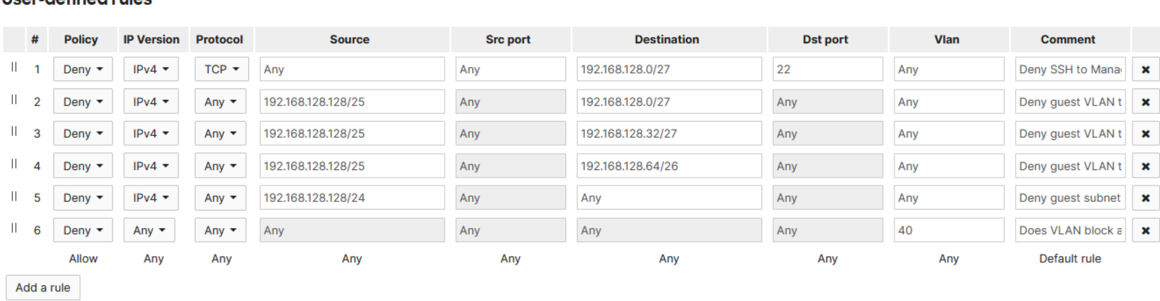
To change this and extend the correct VLAN and its subnet to the wireless clients (VLAN 40), we navigated through the wireless menu of the dashboard to find the settings for the active SSIDs. By specifying that the device should use an external DHCP server, we activated bridge mode and assigned the VLAN to which the addresses would belong. This action made the WAP function as a DHCP relay agent from this point forward.

ACL

In order to restrict traffic according to the specifications of the use-case, we implemented multiple ACLs on the network. This was done network-wide under the ”Switching” menu of the dashboard, as shown in Figure 23.

Since the Meraki dashboard does not differentiate between standard and extended ACLs, both types were configured at the same time. For the standard ACL we specified

only the source address and chose whether it should be permitted or denied, while for our extended ACLs we also specified the destination address, traffic type, and ports.



User-defined rules

#	Policy	IP Version	Protocol	Source	Src port	Destination	Dst port	Vlan	Comment
1	Deny	IPv4	TCP	Any	Any	192.168.128.0/27	22	Any	Deny SSH to Mana
2	Deny	IPv4	Any	192.168.128.128/25	Any	192.168.128.0/27	Any	Any	Deny guest VLAN t
3	Deny	IPv4	Any	192.168.128.128/25	Any	192.168.128.32/27	Any	Any	Deny guest VLAN t
4	Deny	IPv4	Any	192.168.128.128/25	Any	192.168.128.64/26	Any	Any	Deny guest VLAN t
5	Deny	IPv4	Any	192.168.128.128/24	Any	Any	Any	Any	Deny guest subnet
6	Deny	Any	Any	Any	Any	Any	Any	40	Does VLAN block a
	Allow	Any	Any	Any	Any	Any	Any	Any	Default rule

Add a rule

Figure 23: ACL rules in the Meraki dashboard

SNMP access

To provide a monitoring alternative for the Meraki devices, we enabled SNMP access to the Meraki devices. To use SNMP for monitoring, it is necessary to enable SNMP in the ”Organization” ”settings” via the taskbar, where we enable ”Version 2C” in the fold-down menu. Under the ”Network-wide” tab, we navigated to the ”Reporting” section, where we set ”SNMP access” to ”V1/V2c (community string), using ”communitySNMP” as the community string. We also needed to alter the firewall rules for SNMP from ”None” to the IP-ranges 192.168.128.0/27 and 192.168.128.32/27 to allow access from clients in the network.

3.3.4 Use Case 2: Branch-to-branch connectivity (WAN)

In our second use-case, some time has passed and our small start-up company has recently opened a new branch in Oslo with six additional employees. One of the IT-administrators from the Trondheim branch has been sent down to the new office to configure the new branch’s network devices. For this new branch, Stooges has decided to use the 172.168.128.0 /24 address block.

As in their Trondheim branch, Stooges should set up VLANs in the Oslo branch to separate between employee and guest traffic. The employees in Oslo should be able to connect to their network either wired or wirelessly, and guests in Oslo should be isolated from the rest of the network. However, unlike the Trondheim branch, the Oslo branch will not need their own IT-ADMIN VLAN, as Stooges has decided to manage the network devices in Oslo through remote access (SSH) from the Trondheim branch.

The subnet plan for the Oslo branch is as shown in Table 8.

Name	VLAN	IPv4	Default Gateway
MANAGEMENT	10	172.168.128.0 /27	172.168.128.1
EMPLOYEES	30	172.168.128.64 /26	172.168.128.65
GUESTS	40	172.168.128.128 /25	172.168.128.129

Table 8: Oslo VLAN subnet plan

After establishing single-branch connectivity within each branch, the network administrators are tasked with configuring a secure WAN connection between the Oslo and Trondheim branches. As mentioned in Section 2.2.7, since the internet is a public network that does not allow for private IP addresses, a site-to-site VPN connection needs to be established for WAN connectivity between Stooges' two internal networks. To achieve this, the network administrators will need to configure the following network features:

- GRE tunnel between Trondheim and Oslo's border routers to allow the private networks to communicate as though they were directly connected by establishing WAN connectivity via virtual links over the public internet
- IPsec to protect the GRE tunnel and ensure the traffic sent through the tunnel is secured.

3.3.5 Branch-to-branch Connectivity (WAN) with Packet Tracer

To simulate Stooges' WAN setup using traditional legacy devices, we will be using the same router, switch and AP modules which were used to set up Stooges' single-branch LAN in use-case 1 (sec. 3.3.2). However, the router modules for this use case has been modified to have a HWIC-2T WAN card installed in the right-most WIC slot. This was done by turning off the router, and adding the card from the 'MODULE' menu on the left-hand side into the router's WIC slot.

The two LANs are directly connected to the Internet router using Serial DTE connections to connect the TRD-R, ISP, and OSL-R router modules via their serial ports as shown in the topology illustration in Figure 24.

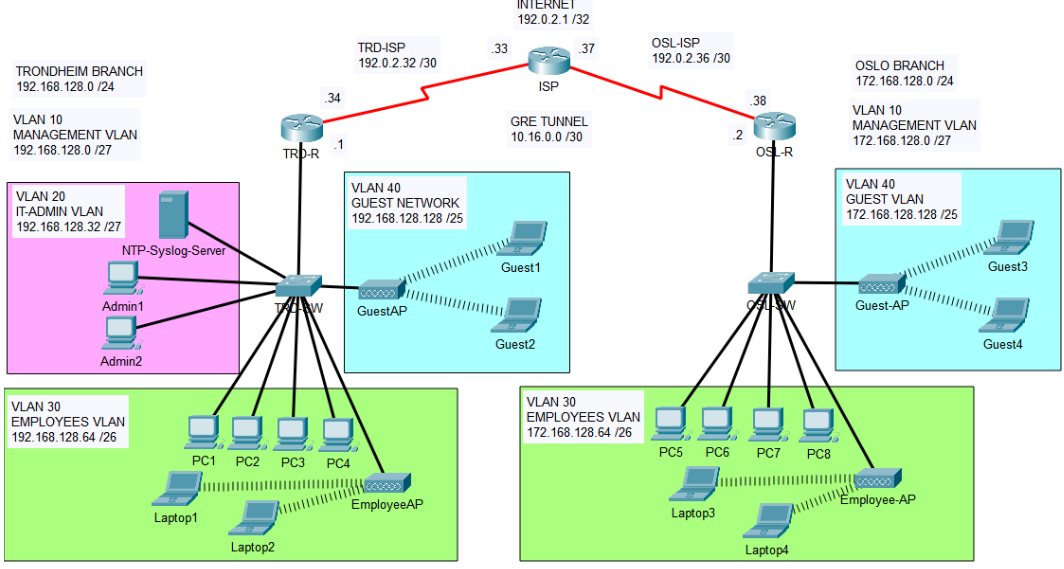


Figure 24: Branch-to-branch WAN topology

Simulating the Internet

Once again, we have assumed that the routing between Stooges' LAN border routers and the ISP router is handled by their ISP. To simulate "internet access", we manually

configured the router module named 'Internet' with the IP addresses shown in Table 9, using the commands found in the Appendix A.1. If a host is able to ping the router's loopback address at 192.0.2.1, we say it has access to the internet.

Name	IPv4
ISP Loopback	**192.0.2.1 /32**
ISP-TRD WAN	**192.0.2.32 /30**
ISP s0/0/0	192.0.2.33
TRD-R s0/0/0	192.0.2.34
ISP-OSL WAN	**192.0.2.36 /30**
ISP s0/0/1	192.0.2.37
OSL-R s0/0/0	192.0.2.38

Table 9: WAN Internet IP allocations

The addressing table for Stooges' branch-to-branch WAN topology is depicted in Table 10, consisting of a list of all their subnets and the IP addresses of each interface.

Name	IPv4
Trondheim LAN	**192.168.128.0 /24**
TRD-R g0/1.10	192.168.128.1
TRD-R g0/1.20	192.168.128.33
TRD-R g0/1.30	192.168.128.65
TRD-R g0/1.40	192.168.128.129
OSLO LAN	**192.168.128.0 /24**
OSL-R g0/1.10	172.168.128.1
OSL-R g0/1.30	172.168.128.65
OSL-R g0/1.40	172.168.128.129
TRD-OSL Tunnel	**10.16.0.0 /30**
TRD-R Tunnel1	10.16.0.1
OSL-R Tunnel1	10.16.0.2

Table 10: Stooges' WAN addressing plan

GRE-Tunnel Configuration

To set up a GRE tunnel between TRD-R and OSL-R, we need to establish the tunnel's interface, assign it the IP addresses given in Table 10, and set the tunnel's source as the router's local interface with the public IP address as well as set the tunnel's destination to the other router's public IP address.

On the TRD-R router, we used the following commands:

```
interface Tunnel1
    ip address 10.16.0.1 255.255.255.252
    tunnel source Serial0/0/0
    tunnel destination 192.0.2.38
    exit
```

In addition to establishing the tunnel, we need to configure the routes on both routers to inform them of the remote subnets behind each router and how to reach them via the GRE-tunnel. The simplest method is to set static routes on each router.

Once the same process is completed on the OSL-R router, we can verify successful configuration of the GRE-tunnel by attempting performing tracert on the IP address of a host within Stooges' Oslo branch from a host within Stooges' Trondheim branch. If the IP address of the tunnel's interface appears as shown in Figure 25, it means traffic was successfully routed through our GRE-tunnel.

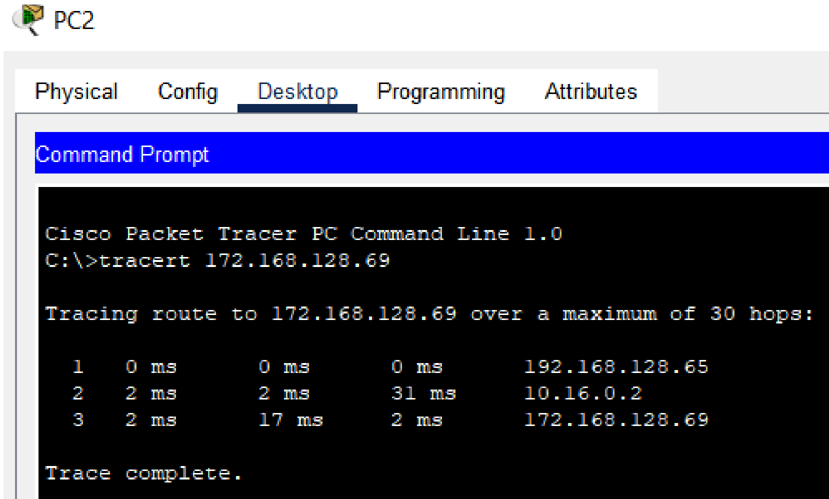


Figure 25: GRE-tunnel route verification

IPSec Configuration

With the establishment of our GRE-tunnel, Stooges' employees in Trondheim can now communicate with employees in Oslo. However, the traffic travelling through the tunnel was not encrypted, leaving it susceptible to eavesdropping attacks.

Setting up IPSec for the GRE-Tunnel comes with several steps. First, we needed to make sure we had the license which allowed us to run crypto ISAKMP commands. To do this, we ran the following boot sequence on TRD-R and OSL-R:

```
license boot module c2900 technology-package securityk9
```

The next step was to create and configure an IPSec ISAKMP policy and pre-shared key on both routers using these commands:

```
crypto isakmp policy 5
    encr aes
    authentication pre-share
    group 5
    lifetime 600
    exit
crypto isakmp key tTJsQA8f address 192.0.2.38
```

Since we only need to encrypt the GRE traffic that is being transmitted between TRD-R and OSL-R's tunnel source, we created an extended access list to define which traffic to

permit through the tunnel and encrypt. On TRD-R, this extended ACL was named 'OSL-VPN', and permitted traffic destined for Stooge's Oslo branch. Once we have defined what traffic we will be encrypting, we needed to define how we will be encrypting the traffic using a transform set which defines how we want to encrypt data with IPsec. For this case, we configured the following transform set on both routers:

```
crypto ipsec transform-set IPSEC-GRE esp-aes 256 esp-sha-hmac
```

The final step was to enable IPsec encryption by tying together the transform set, the extended ACL, and the IPsec peer, before applying it to the relevant interface. As an example, the following commands were used to achieve this on TRD-R:

```
crypto map IPSEC-TO-OSL 10 ipsec-isakmp
    set peer 192.0.2.38
    set transform-set IPSEC-GRE
    match address OSL-VPN
    exit
int s0/0/0
    crypto map IPSEC-TO-OSL
```

To ensure that IPSec has been configured correctly, we ran the `show crypto ipsec sa` command to display active IPsec Security Associations (SAs) on TRD-R or OSL-R. This command provides information such as the source and destination IP addresses, the type of encryption used, and the lifetime of the SA, as illustrated in Figure 26.

```
interface: Serial0/0/0
    Crypto map tag: IPSEC-TO-OSL, local addr 192.0.2.34

   protected vrf: (none)
   local  ident (addr/mask/prot/port): (192.0.2.34/255.255.255.255/47/0)
   remote  ident (addr/mask/prot/port): (192.0.2.38/255.255.255.255/47/0)
   current_peer 192.0.2.38 port 500
    PERMIT, flags={origin_is_acl,}
   #pkts encaps: 174, #pkts encrypt: 174, #pkts digest: 0
   #pkts decaps: 178, #pkts decrypt: 178, #pkts verify: 0

   local crypto endpt.: 192.0.2.34, remote crypto endpt.:192.0.2.38
   path mtu 1500, ip mtu 1500, ip mtu idb Serial0/0/0
   current outbound spi: 0x31EB1F31(837492529)

   inbound esp sas:
    spi: 0xEAF680D8(3942023384)
      transform: esp-aes 256 esp-sha-hmac ,
      in use settings ={Tunnel, }
      conn id: 2004, flow_id: FPGA:1, crypto map: IPSEC-TO-OSL
      sa timing: remaining key lifetime (k/sec): (4525504/597)
      IV size: 16 bytes
      replay detection support: N
      Status: ACTIVE

   outbound esp sas:
    spi: 0x31EB1F31(837492529)
      transform: esp-aes 256 esp-sha-hmac ,
      in use settings ={Tunnel, }
      conn id: 2005, flow_id: FPGA:1, crypto map: IPSEC-TO-OSL
      sa timing: remaining key lifetime (k/sec): (4525504/597)
      IV size: 16 bytes
      replay detection support: N
      Status: ACTIVE
```

Figure 26: Active TRD-R security associations

3.3.6 Branch-to-branch Connectivity (WAN) with Cisco Meraki

To establish a branch-to-branch connection, we used the network implemented for our first use-case (sec. 3.3.3) as the Trondheim branch, and the spare MX router made available to us as the Oslo branch. This extra MX was not physically available to us, but we added it (and its liscense) to our organization, and created a separate network for it called ”Oslo Branch”.

Our implementation of the VPN between the two MX routers consisted of configuring a site-to-site VPN using the Meraki dashboard. The ”Home Network” was set as a hub, while the ”Oslo branch” was configured as a spoke, and the connection established between them was automatically encrypted using IPsec and AES encryption. Due to issues with unfriendly NAT on the ”Home Network”, we implemented a static UDP port to route VPN traffic through, as shown in Figure 27.

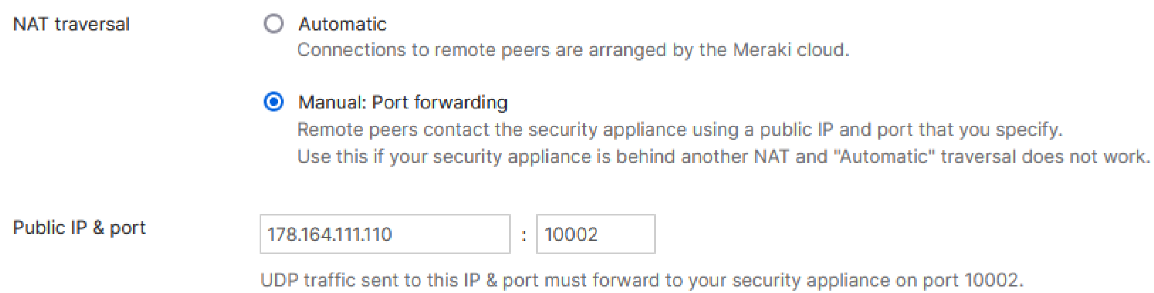


Figure 27: Static UDP port for VPN

Inside ”VPN Settings”, Home Network’s VLANs were given permission to communicate with those on Oslo Branch by enabling VPN mode for all of them. Figure 28 shows these VLANs as the VPN’s participants. We did not configure any VLANs on the Oslo Branch as it was not necessary for establishing the WAN connection between the sites.

VPN participants

Search by Network, VLAN name,

Network ▲	VLAN name	VLAN ID	Subnet
Home Network - appliance	Management	10	192.168.128.0/27
Home Network - appliance	IT-Admins	20	192.168.128.32/27
Home Network - appliance	Employees	30	192.168.128.64/26
Home Network - appliance	Guest	40	192.168.128.128/25
Oslo Branch	Single LAN Settings	0	192.168.0.0/24

5 results

Figure 28: VLANs allowed to use the VPN

3.4 Management and Monitoring Capabilities

3.4.1 Using Packet Tracer

Packet Tracer, representing the capabilities of traditional network architectures, offers several methods to monitor and manage network devices. The CLI on each network device, which can be accessed either directly through a console connection or remotely through SSH, serves as the simplest way to manage and retrieve information about the device and network's current configurations.

An alternative to monitoring and managing network devices on a traditional network is through a MIB browser, which requires the configuration of SNMP read only (ro) or read and write (rw) community strings. In Packet Tracer, the MIB browser is accessible through the 'Desktop' tab on most end-devices. Although we did not configure a read and write community string in our use case, doing so makes it possible to send `snmp-set` commands to configure certain aspects of an SNMP agent using the MIB browser.

To log information about system changes or events within a traditional network, network administrators would need to configure and set up syslog and NTP as we have done in Section 3.3.2, to forward timestamped event logs for monitoring and troubleshooting to a dedicated syslog server.

In addition to collecting event logs, it is also possible to collect information about the IP traffic flow within a traditional network using Netflow. The information can then be forwarded to a web server for monitoring once configured on the router as we have also done in Section 3.3.2.

3.4.2 Using Cisco Meraki

Through the Meraki dashboard, a set of analytics and data automatically collected on the networks are accessible under the "Monitor" column for each section's menu, as displayed in Figure 29. We primarily used this menu to navigate through the dashboard and find the correct pages for monitoring and managing different parts of the Meraki based network.

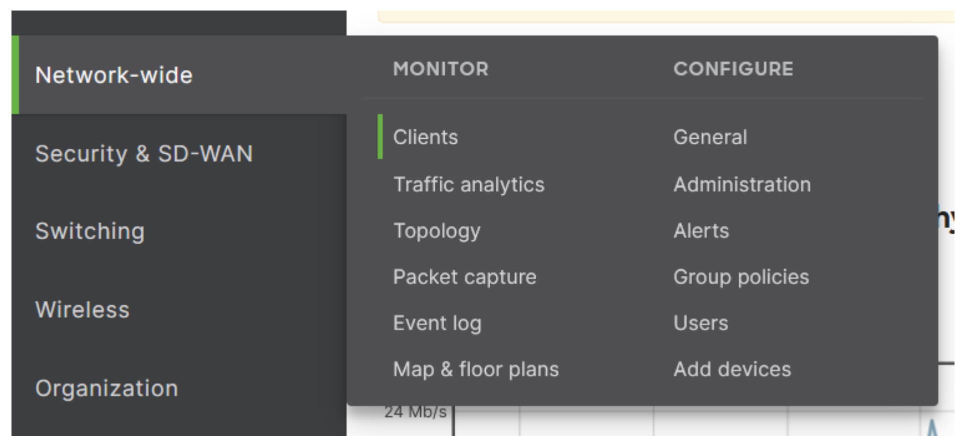
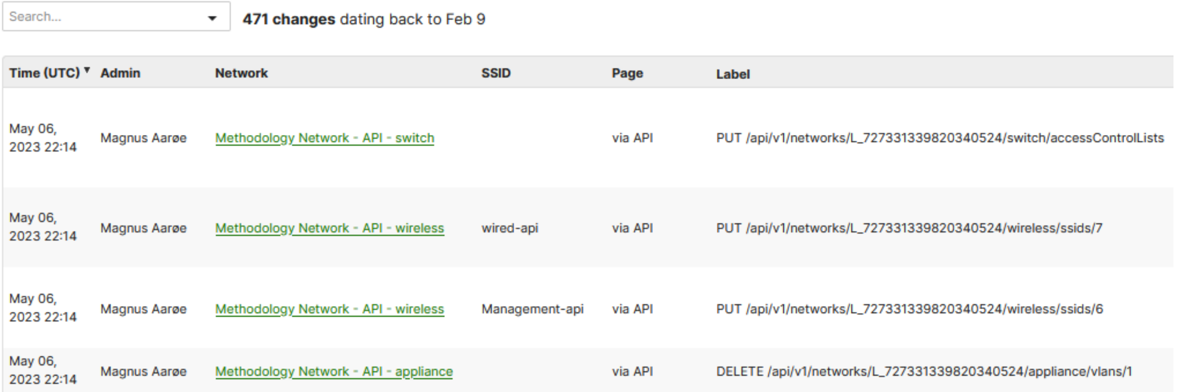


Figure 29: Monitor column in dashboard's Network-wide menu

To review configurations and changes made by us to the network, we used the dashboard's change logs shown in Figure 30. These logs can be accessed using the Meraki

dashboard API as well. Information available on the dashboard can also be downloaded in the form of CSV, XML, or Excel files, depending on the content.



NTNU change log

Search... ▾ **471 changes** dating back to Feb 9

Time (UTC) ▾	Admin	Network	SSID	Page	Label
May 06, 2023 22:14	Magnus Aarøe	Methodology Network - API - switch		via API	PUT /api/v1/networks/L_727331339820340524/switch/accessControlLists
May 06, 2023 22:14	Magnus Aarøe	Methodology Network - API - wireless	wired-api	via API	PUT /api/v1/networks/L_727331339820340524/wireless/ssids/7
May 06, 2023 22:14	Magnus Aarøe	Methodology Network - API - wireless	Management-api	via API	PUT /api/v1/networks/L_727331339820340524/wireless/ssids/6
May 06, 2023 22:14	Magnus Aarøe	Methodology Network - API - appliance		via API	DELETE /api/v1/networks/L_727331339820340524/appliance/vlans/1

Figure 30: The Meraki change logs

SNMP is also available as an option for retrieving information about the network, as was configured in Section 3.3.3. To use this method of retrieving information, we installed a MIB browser for Windows, the iReasoning MiB Browser (version 14 build 4722), downloaded from their website [34]. After extracting the downloaded folder, we go into the ”bin” folder and execute the ”browser.bat” file to run the MiB browser. To retrieve information, we input the IP address of desired device, and specify the SNMP version and community string accessed in the ”Advanced...” button. We then select what type information we want to retrieve using the MiB-tree, set the SNMP operation as ”Get”, and press ”GO”. The devices support many Object Identifiers (OIDs) in the subset of *SNMPv2-MIB .1.3.6.1.2.1.1* and *IF-MIB .1.3.6.1.2.1*, in addition to offering a MIB file under ”Organization” SNMP settings, which are used for polling the dashboard (cloud) for Meraki specific information, instead of the devices.

3.5 Automation

3.5.1 Automation with Packet Tracer

Traditional networking solutions do not offer much in terms of automation by default. In our Packet Tracer setup, we saved some time by copy-pasting pre-written commands into network devices when configuring them.

For example, when configuring the OSL-R for our second use case, we copied TRD-R’s running configuration into an empty file, changed certain parameters such as IP addresses and subnets, and pasted the altered configuration file in OSL-R. We did this in small sections to make sure a task was successfully configured before pasting commands for the next task.

3.5.2 Automation with Cisco Meraki

Meraki, on the other hand, provides a set of tools and mechanisms for automating network configuration and management, which we explore the potential of in more detail in Section 4.3. These resources include API access, templates, community platforms, and modules among other things.

One of the most important tools Meraki provides is their open dashboard API, for which they also offer a Python SDK module via GitHub [35]. This API can be used to communicate with the dashboard directly, omitting the need for a web-browser. To access and use it, we enabled the API functionality under ”Dashboard API Access” in ”Organization” settings, and generated an API-key tied to one of our admin accounts. Each account is given a maximum of two unique keys to use for this purpose, as depicted in Figure 31.

API access

API keys

Key	Created at	Last used	
************************************91c2	Apr 10 2023 14:49 UTC	May 06 2023 12:27 UTC	Revoke
************************************5c71	Apr 08 2023 14:59 UTC	Never	Revoke

Figure 31: Generating an API-key

The Meraki Python SDK (version 1.27.0) was then downloaded using PyPI, Visual Studio Code (VS Code) was chosen as the editor, and the scripts were run using the Python interpreter version 3.11.3. To connect to the dashboard API and make changes to the network, a session had to first be established using the API-key.

Below is an example of how a session with the dashboard was established, using the appropriate API-key and some modules provided by the Meraki Python SDK. The variable *dashboard* is then referenced by all functions as changes are made to or information is retrieved from the dashboard.

```

MERAKI_DASHBOARD_API_KEY = "YOUR-API-KEY-HERE"
# Establish session with dashboard API
dashboard = meraki.DashboardAPI(
    api_key=MERAKI_DASHBOARD_API_KEY,
    suppress_logging=True
)

```

An alternative to using the dashboard API is to send HTTP requests with the Postman application [36]. An example of this is shown in Figure 32, where a list of devices in the organization was requested using the following URL with the API-key provided in the request’s header: `https://api.meraki.com/api/v1/organizations/:organizationId/devices`.

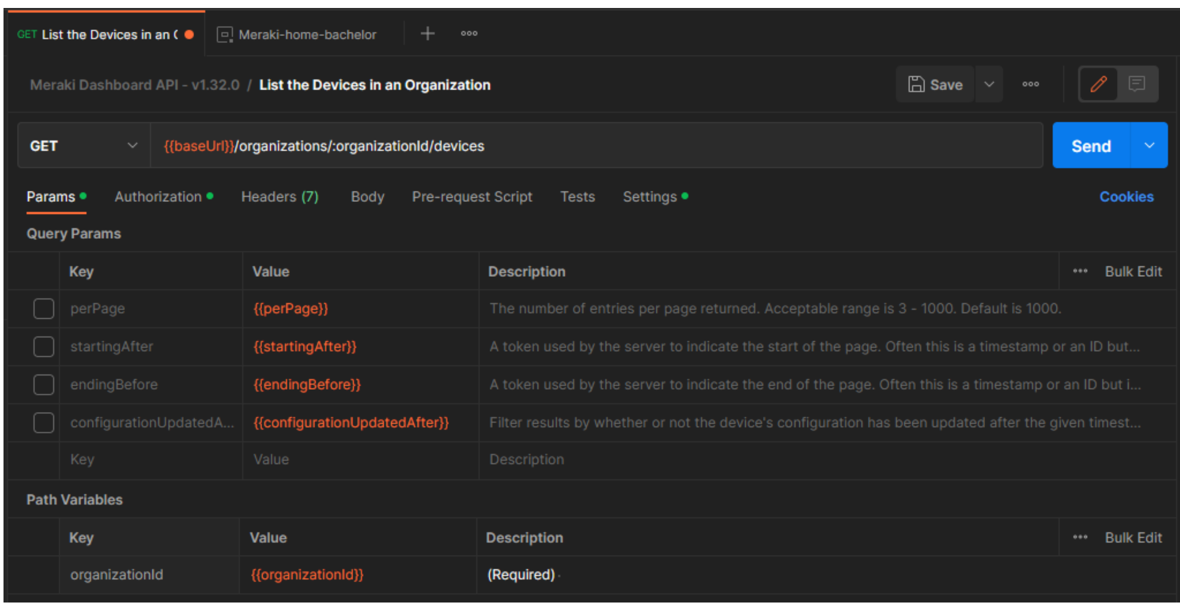


Figure 32: Meraki API request using Postman

4 Results

In this chapter, we present the results of our configuration process as described in Chapter 3, sectioned in order of the criteria of analysis defined in Section 3.2. We have also chosen to separate these results according to each network solution within each criteria to maintain our focus on the criteria itself and present our findings as it specifically related to each.

4.1 Ease of Configuration and Setup

As Section 3.2 states, we devised two use-cases for our testing purposes, and decided on a set of features that would need to be implemented for both of them (described in sec. 3.3.1 and 3.3.4) .

We defined the process of implementing each feature as a *task*, and used the assessment method explained in Section 3.2.1 and Table 2 to calculate a total score for each task, which aided us in evaluating and comparing the ease with which they were completed. In this section, we present the results of the aforementioned rubric and provide more information about our experience throughout the process.

4.1.1 Rubric Results - Packet Tracer

Table 11 displays a list of the tasks that were configured in Sections 3.3.2 and 3.3.5 in our traditional network setup using Packet Tracer. Based on the wide range of the total scores calculated for these tasks, we can see that the tasks varied greatly in their ease, which we interpreted as the overall process of deploying and configuring the network being challenging, though not overly difficult.

Task \ Question Nr	1.1	1.2	1.3	1.4	1.5	1.6	2.1	2.2	2.3	3.1	3.2	4.1	5.1	5.2	5.3	**Total**
Basic config	✗	✓	✗	✓	✓	✓	✓	✓	✗	✓	✓	✓	✓	✓	✗	9
VLAN(switch)	✗	✗	✗	✓	✓	✓	✓	✗	✗	✓	✓	✓	✓	✓	✓	8
VLAN(router)	✗	✗	✗	✓	✓	✓	✓	✓	✗	✓	✓	✓	✓	✓	✗	10
DHCP	✗	✗	✗	✓	✓	✓	✗	-	✗	✓	✓	✗	-	-	-	7
ACLs	✓	✗	✗	✗	✓	✓	✗	-	✗	✓	✓	✓	✓	✗	✗	5
NTP	✗	✗	✗	✓	✓	-	✗	-	✓	✓	✓	✗	-	-	-	6
SNMP	✓	✗	✗	✓	✗	-	✗	-	✗	✓	✓	✓	-	-	-	6
Syslog	✗	✗	✗	✓	✓	-	✓	✓	✗	✓	✓	✓	-	-	-	10
Netflow	✓	✓	✗	✗	✗	-	✗	-	✗	✓	✓	✓	-	-	-	4
NAT	✓	✗	✗	✗	✗	-	✗	-	✗	✓	✓	✓	-	-	-	5
GRE tunnel	✓	✗	✗	✓	✗	-	✓	✓	✗	✓	✓	✗	-	-	-	7
IPsec	✓	✗	✗	✗	✗	-	✓	✓	✗	✓	✓	✓	✓	✓	✗	7

Table 11: Results of ease of configuration - traditional network in Packet Tracer

We can see that for cases where official documentation was lacking or insufficient, we were able to seek guidance from an external source[1]. This mainly applied for the execution of ACLs, SNMP, Netflow, NAT, GRE tunneling, and IPSec. Once remote access

[1]The external source in these cases refers to various teaching materials provided to us in previous courses such as lab assignments and/or obligatory exercises.

via SSH was configured on the network devices, physical access[2] was no longer necessary for configuration. For the execution of most tasks, we felt it was clear where each task needed to be executed.

When using the CLI, we were not provided with any guidance on the interface itself. Although, the commands `?` and `help` could be (and were, in some cases) used for seeing a list of the valid commands we could use at any given time, or checking which arguments a specific command accepted, the answer to question 1.3 was "no" for all tasks because this guidance was not readily given, but needed to be sought out.

Questions 1.5 and 1.6 explicitly refers to the use of official documentation, and we can see that where this type of documentation was lacking, an external source was sought out to fills its place. In addition, our results shows that the available official documentation was sufficient for performing the related task. Question 1.6 is left unanswered in cases where we did not use and thus could not judge the documentation that was available.

In terms of transparency, we experienced an overall lack of feedback when performing most tasks. Feedback that was given was often for a specific step that was taken, or because something was wrong. In some cases, the feedback also provided some information about the state of the device, which was helpful in confirming that the task was being done properly.

We only encountered a false positive once, which occurred during the implementation of NTP, where the OSL-R router believed it was synchronized when it in reality, it had the wrong date. It should be noted that this issue could be due to a limitation of our emulated environment, but we did not have a viable way of confirming it.

All tasks were successfully completed and done in a timely manner, meaning that no task took longer to complete than we expected it would in a real network setup. For tasks such as implementing ACLs, this estimated time also takes into account the time that was spent on defining and preparing the policies before actually implementing them through the CLI.

In a few cases, the estimated duration of time spent on a task was not dedicated fully to the execution of the task itself, but to where and how it could or should be done. The distinction here, however, is relatively vague, and to avoid that, we considered timing each task from beginning to end. We later decided against this, because the emulated nature of our network in Packet Tracer would not provide us with valid enough results to later compare to our network in Cisco Meraki.

Questions regarding errors were not applicable in most cases, and, when they were, they deducted a single point from only two of the tasks. In the case of configuring VLANs on the switch, a faulty configuration where the name of the VLAN included a space was carried out regardless of the error, and another name was automatically chosen for the VLAN that was created. The other instance occurred when implementing ACLs, where the error did not provide enough information about what had caused it, making it more difficult to find and fix the issue.

4.1.2 Rubric Results - Cisco Meraki

Table 12 displays the results from tasks performed under Sections 3.3.3 and 3.3.6. The first thing to note when comparing this table to Table 11, is the total number of tasks. Many of the functionalities we implemented in our simulated traditional network such as NTP, NAT, and IPsec, were present by default in the Cisco Meraki network, leaving us with fewer tasks to perform in order to meet each use case's requirements.

[2] In Packet Tracer's case this translates to selecting individual devices to configure, rather than using SSH access via a PC.

We can also see that the total scores for these tasks are closer to each other in range than those calculated in Table 11, which we can interpret as the tasks being overall easier to perform.

Task \ Question Nr	1.1	1.2	1.3	1.4	1.5	1.6	2.1	2.2	2.3	3.1	3.2	4.1	5.1	5.2	5.3	**Total**
Basic config	✓	✗	✗	✓	✓	✗	✓	✓	✗	✓	✗	✗	-	-	-	6
VLAN	✗	✗	✓	✓	✓	-	✓	✓	✗	✓	✓	✓	-	-	-	11
DHCP Relay	✗	✗	✓	✗	✓	-	✓	✗	✗	✓	✓	✓	-	-	-	9
ACLs	✗	✗	✓	✓	✓	-	✓	✗	✗	✓	✓	✓	-	-	-	9
SNMP	✗	✗	✓	✗	✓	✓	✓	✗	✗	✓	✓	✓	-	-	-	9
site-to-site VPN	✗	✗	✓	✓	✓	✓	✓	✗	✓	✓	✗	✗	-	-	-	7

Table 12: Results of ease of configuration - Cisco Meraki network

We rarely needed to seek an external resource in order to perform a task, and physical access to the devices was not necessary due to the dashboard's availability via a web-browser. In our initial attempt in connecting the devices to the internet and each other, we experienced a brown-field issue (discussed further in sec. 5.1.2) that resulted in a long process of troubleshooting. This was the only instance in which we sought information from sources other than Meraki, and was also the only one in which the official documentation proved to be insufficient for finding the source of the problem.

The different pages of the Meraki dashboard, that we navigated between in order to perform various tasks, often included general information about the settings on the page and their function, such as what enabling something would mean and how the network would be affected by it.

In most cases, it was also easy to understand where we needed to go to configure something, as the dashboard provides named and categorized menus, as well as a search bar that can be utilized for searching through both the dashboard and Meraki documentation. Although we did not use the official documentation that was available in all cases, for most of the times we did, we found them to be sufficient for performing the task at hand.

Whenever a change was made on the dashboard, feedback was given via a confirmation message on screen. These messages, however, only confirmed that the setting was saved and rarely provided further insight into the state of the device or network. We used the devices' LED lights as visual indicators that a change had been applied, but it was not clear when exactly the device would be notified of a change by the controller and when said change would actually be taking place.

All tasks were successfully completed, though not all were done in what we consider a timely manner. In addition to the brown-field issue we experienced during the initial setup, establishing a site-to-site VPN to the MX router in our Oslo Branch network also took longer than realistically expected. These two tasks were also the ones where the majority of the time spent was on understanding and troubleshooting the problem, not its actual implementation.

The issues related to establishing a site-to-site VPN happened due to a NAT-unfriendliness that occurred because of the presence of an upstream external router (which was the one actually responsible for address translation rather than our MX router). This was also the only instance of a false positive that we experienced; while the VPN connection was established and all indicators showed green, no traffic was able to traverse between the

two sites. Section 3.3.6, Figure 27 illustrates the solution we used for this problem.

We were not presented with any explicit errors when performing any of the tasks, which, in this particular case, contributed negatively to some of the tasks' ease. For example, with the NAT-unfriendliness issue described earlier, a lack of errors made it difficult to understand why the problem had occurred and where, which resulted in a late discovery and solution. In general, we discovered that the Meraki dashboard would rarely provide us with errors, and that we had to search for them ourselves if we noticed something was wrong with the network.

4.2 Ease of Management and Monitoring

4.2.1 Using Packet Tracer

In Section 3.4.1, we discussed the primary methods of managing traditional networks and demonstrated how to configure them. Although the CLI is mainly used to manage the network and the devices within it through configuration commands, it also comes with several `show` commands which serve as a method for retrieving a range of information about the device, such as its interface states, current configurations, and network traffic.

By running the `show ?` command, the CLI will output a list of commands to monitor the state of the network and its devices, some of which were run in Section 3.3.2 and 3.3.5 to verify successful configurations. Some other commands which weren't previously demonstrated are as follows.

The `show ip interface brief` command, which provides a summary of the state of the interfaces on a network device as shown in Figure 33.

```
TRD-R#show ip interface brief
Interface              IP-Address      OK? Method Status
Protocol
GigabitEthernet0/0     unassigned      YES NVRAM  administratively down down
GigabitEthernet0/1     unassigned      YES NVRAM  up                    up
GigabitEthernet0/1.10  192.168.128.1   YES manual up                    up
GigabitEthernet0/1.20  192.168.128.33  YES manual up                    up
GigabitEthernet0/1.30  192.168.128.65  YES manual up                    up
GigabitEthernet0/1.40  192.168.128.129 YES manual up                    up
Serial0/0/0            192.0.2.34      YES NVRAM  up                    up
Serial0/0/1            unassigned      YES NVRAM  administratively down down
Tunnel1                10.16.0.1       YES manual up                    up
Vlan1                  unassigned      YES unset  administratively down down
```

Figure 33: TRD-R interface status

Running the `show ip dhcp pool` command reveals the status of existing DHCP pools, while running `show ip dhcp binding` provides a mapping of leased IP addresses to device MAC addresses. The outputs of these commands are shown in Figures 34 and 35.

```
TRD-R#sh ip dhcp pool

Pool TRD-MANAGEMENT :
 Utilization mark (high/low)    : 100 / 0
 Subnet size (first/next)       : 0 / 0
 Total addresses                : 30
 Leased addresses               : 0
 Excluded addresses             : 5
 Pending event                  : none

 1 subnet is currently in the pool
 Current index        IP address range                    Leased/Excluded/Total
 192.168.128.1        192.168.128.1    - 192.168.128.30    0    / 5     / 30

Pool TRD-IT-ADMIN :
 Utilization mark (high/low)    : 100 / 0
 Subnet size (first/next)       : 0 / 0
 Total addresses                : 30
 Leased addresses               : 2
 Excluded addresses             : 5
 Pending event                  : none

 1 subnet is currently in the pool
 Current index        IP address range                    Leased/Excluded/Total
 192.168.128.33       192.168.128.33   - 192.168.128.62    2    / 5     / 30

Pool TRD-EMPLOYEES :
 Utilization mark (high/low)    : 100 / 0
 Subnet size (first/next)       : 0 / 0
 Total addresses                : 62
 Leased addresses               : 6
 Excluded addresses             : 5
 Pending event                  : none

 1 subnet is currently in the pool
 Current index        IP address range                    Leased/Excluded/Total
 192.168.128.65       192.168.128.65   - 192.168.128.126   6    / 5     / 62

Pool TRD-GUESTS :
 Utilization mark (high/low)    : 100 / 0
 Subnet size (first/next)       : 0 / 0
 Total addresses                : 126
 Leased addresses               : 2
 Excluded addresses             : 5
 Pending event                  : none

 1 subnet is currently in the pool
 Current index        IP address range                    Leased/Excluded/Total
 192.168.128.129      192.168.128.129  - 192.168.128.254   2    / 5     / 126
```

Figure 34: TRD-R DHCP pool status

```
TRD-R#sh ip dhcp binding
IP address       Client-ID/              Lease expiration        Type
                 Hardware address
192.168.128.39   0000.0C09.8037           --                     Automatic
192.168.128.69   0004.9A2E.A629           --                     Automatic
192.168.128.66   0090.2B30.B79C           --                     Automatic
192.168.128.67   0090.2B3A.BD55           --                     Automatic
192.168.128.71   0009.7CEB.2BB9           --                     Automatic
192.168.128.70   0030.A3AB.6B4C           --                     Automatic
192.168.128.130  0001.6397.4BC3           --                     Automatic
192.168.128.131  00E0.A3C2.193A           --                     Automatic
```

Figure 35: TRD-R DHCP pool binding

The `show ip nat translations` command outputs a similar table, shown in Figure 36,

mapping public IP addresses and ports to translated private IP addresses.

```
TRD-Router#sh ip nat statistics
Total translations: 0 (0 static, 0 dynamic, 0 extended)
Outside Interfaces: Serial0/0/0
Inside Interfaces: GigabitEthernet0/1.20 , GigabitEthernet0/1.30 ,
GigabitEthernet0/1.40
Hits: 36  Misses: 939
Expired translations: 36
Dynamic mappings:
TRD-Router#sh ip nat translations
Pro  Inside global     Inside local       Outside local      Outside global
icmp 192.0.2.34:106    192.168.128.40:106 192.0.2.33:106     192.0.2.33:106
icmp 192.0.2.34:107    192.168.128.40:107 192.0.2.1:107      192.0.2.1:107
icmp 192.0.2.34:108    192.168.128.40:108 192.0.2.1:108      192.0.2.1:108
icmp 192.0.2.34:109    192.168.128.40:109 192.0.2.1:109      192.0.2.1:109
icmp 192.0.2.34:110    192.168.128.40:110 192.0.2.1:110      192.0.2.1:110
```

Figure 36: TRD-R NAT status

From these examples, we demonstrated how running these `show` commands provided one or more easy-to-read tables of information, organized into columns and rows. Retrieving information about a network device using this method is arguably the easiest and most straightforward, but the network administrator needs to know which command to use to retrieve the desired information. Additionally, due to the distributed control plane of traditional networks, network administrators are limited to monitoring to each device individually, making the solution less scalable.

The alternative to viewing information about the network's devices through the CLI is to do so using SNMP get-requests. However, this method is slightly more complicated to use, and requires an understanding of how the MIB Tree and OID Tree work.

To retrieve information about our network devices within Packet Tracer, we used the MIB Browser on the Admin1 PC. We filled out the IP address of the network device we wanted to retrieve information from, the SNMP community strings, and selected the SNMP version we had set up on the device as shown in Figure 37.

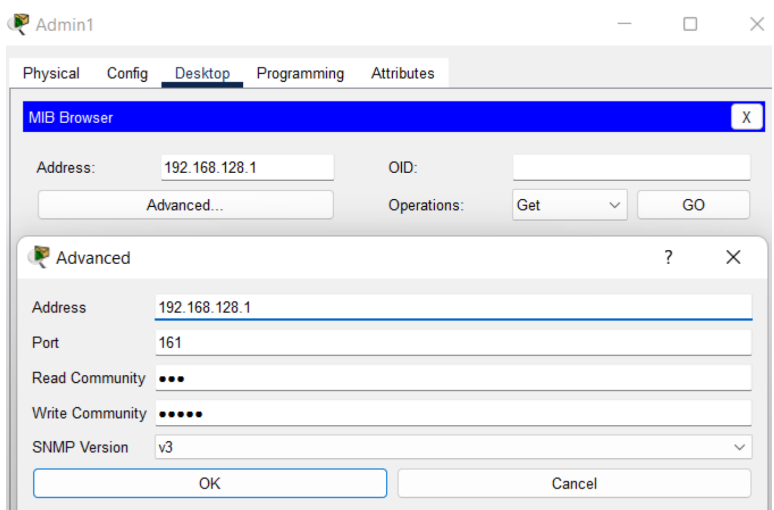


Figure 37: SNMP MIB browser settings

As an example, we have chosen to retrieve information about the router's system. By

either navigating through the MIB Tree or setting the OID of .system, we can select the 'Get-Bulk' option and press 'GO' to send an SNMP get-request to our target SNMP agent. The resulting response is shown as an easily readable table, displayed in Figure 38.

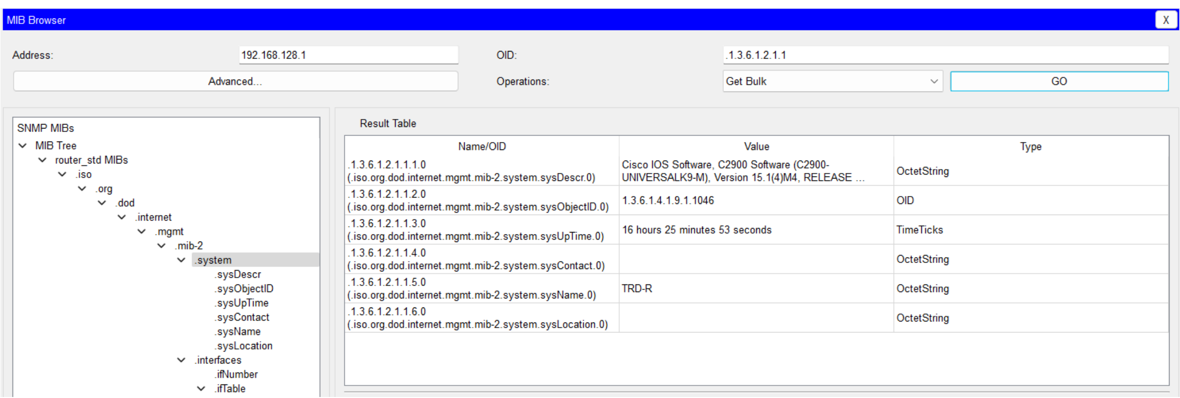


Figure 38: SNMP MIB browser TRD-R system get-request

For a slightly more complicated example, in Figure 39, we sent a get-request using the .ifPhysAddress OID. This produced a result table listing the physical address of each interface on TRD-R.

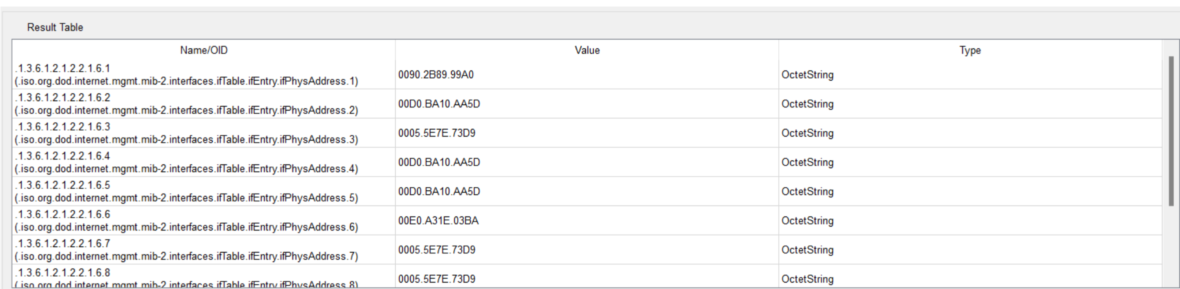
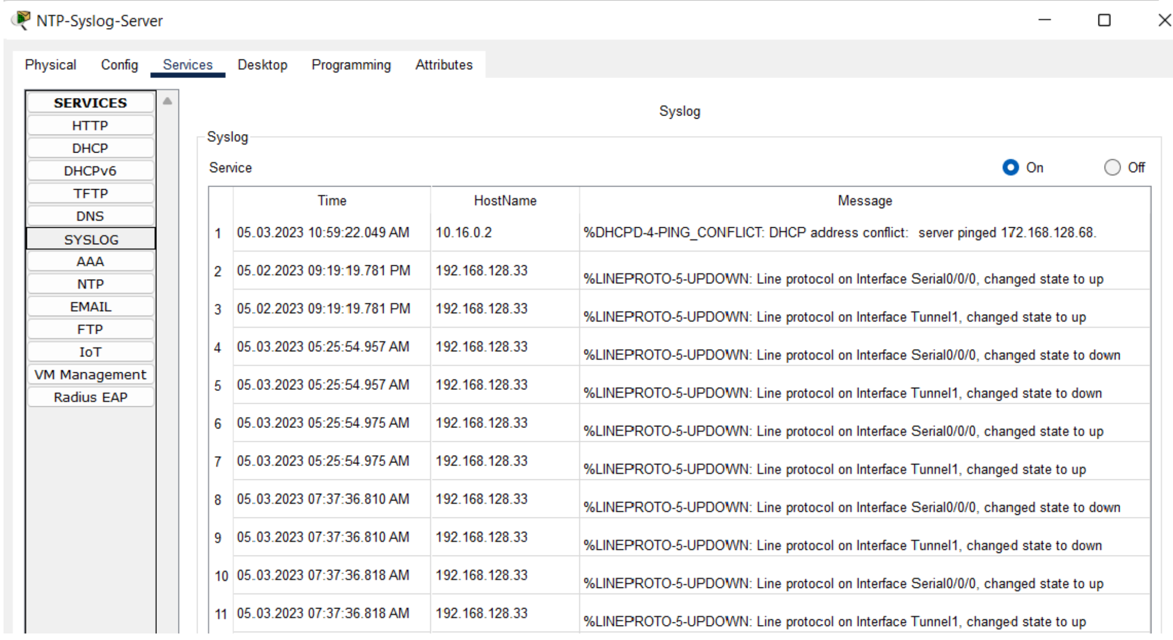


Figure 39: SNMP MIB browser TRD-R system get-request

However, as Figure 39 shows, the Name/OID column does not include the name of each interface, making it slightly challenging to know which interface the values are associated with. To retrieve that information, we would need to send another get-request using the .ifDescr OID. The information from the resulting table should provide information to map the name of each interface to their physical addresses. This goes to show that retrieving more detailed information using SNMP get-requests can get rather complicated. Network administrators would need to be familiar with which OID refers to which interface in order to properly interpret the results, and mapping the information from each get-request could get tedious.

While retrieving information using the CLI and SNMP requires initiative from the network administrator, the syslog monitoring protocol automatically generates event logs which can be accessed on the dedicated Syslog server by navigating to its 'Services' window and selecting the 'SYSLOG' tab. This provides the network administrator with a centralized compilation of timestamped events as shown in Figure 40.



	Time	HostName	Message
1	05.03.2023 10:59:22.049 AM	10.16.0.2	%DHCPD-4-PING_CONFLICT: DHCP address conflict: server pinged 172.168.128.68.
2	05.02.2023 09:19:19.781 PM	192.168.128.33	%LINEPROTO-5-UPDOWN: Line protocol on Interface Serial0/0/0, changed state to up
3	05.02.2023 09:19:19.781 PM	192.168.128.33	%LINEPROTO-5-UPDOWN: Line protocol on Interface Tunnel1, changed state to up
4	05.03.2023 05:25:54.957 AM	192.168.128.33	%LINEPROTO-5-UPDOWN: Line protocol on Interface Serial0/0/0, changed state to down
5	05.03.2023 05:25:54.957 AM	192.168.128.33	%LINEPROTO-5-UPDOWN: Line protocol on Interface Tunnel1, changed state to down
6	05.03.2023 05:25:54.975 AM	192.168.128.33	%LINEPROTO-5-UPDOWN: Line protocol on Interface Serial0/0/0, changed state to up
7	05.03.2023 05:25:54.975 AM	192.168.128.33	%LINEPROTO-5-UPDOWN: Line protocol on Interface Tunnel1, changed state to up
8	05.03.2023 07:37:36.810 AM	192.168.128.33	%LINEPROTO-5-UPDOWN: Line protocol on Interface Serial0/0/0, changed state to down
9	05.03.2023 07:37:36.810 AM	192.168.128.33	%LINEPROTO-5-UPDOWN: Line protocol on Interface Tunnel1, changed state to down
10	05.03.2023 07:37:36.818 AM	192.168.128.33	%LINEPROTO-5-UPDOWN: Line protocol on Interface Serial0/0/0, changed state to up
11	05.03.2023 07:37:36.818 AM	192.168.128.33	%LINEPROTO-5-UPDOWN: Line protocol on Interface Tunnel1, changed state to up

Figure 40: Syslog service on NTP-Syslog server

Assuming the network devices' clocks are successfully synchronized with the NTP server, the logs' timestamps makes it very easy for network administrators to create a timeline of events for auditing, monitoring, and troubleshooting. The main challenge we can foresee with using NTP and syslog as a network monitoring service is that the network administrators would need to be acquainted with the syslog message format to properly understand what the event logs mean for the system. This is, however, something we feel is relatively easy to learn since it uses a standard message format.

Another monitoring protocol used for monitoring traditional networks is called Netflow, which focuses more on providing insight into how traffic flows throughout the network. The application organizes traffic with the same source and destination IP address, ports, and protocol interface into one flow, as shown in Figure 41.

In addition to statistics and metadata about the flow itself, Netflow also calculates and presents each flows' traffic contribution. This provides network administrators with the ability to monitor and analyze network traffic patterns in real time, allowing them to gain insight into the traffic traversing the network, which can help with troubleshooting network performance issues.

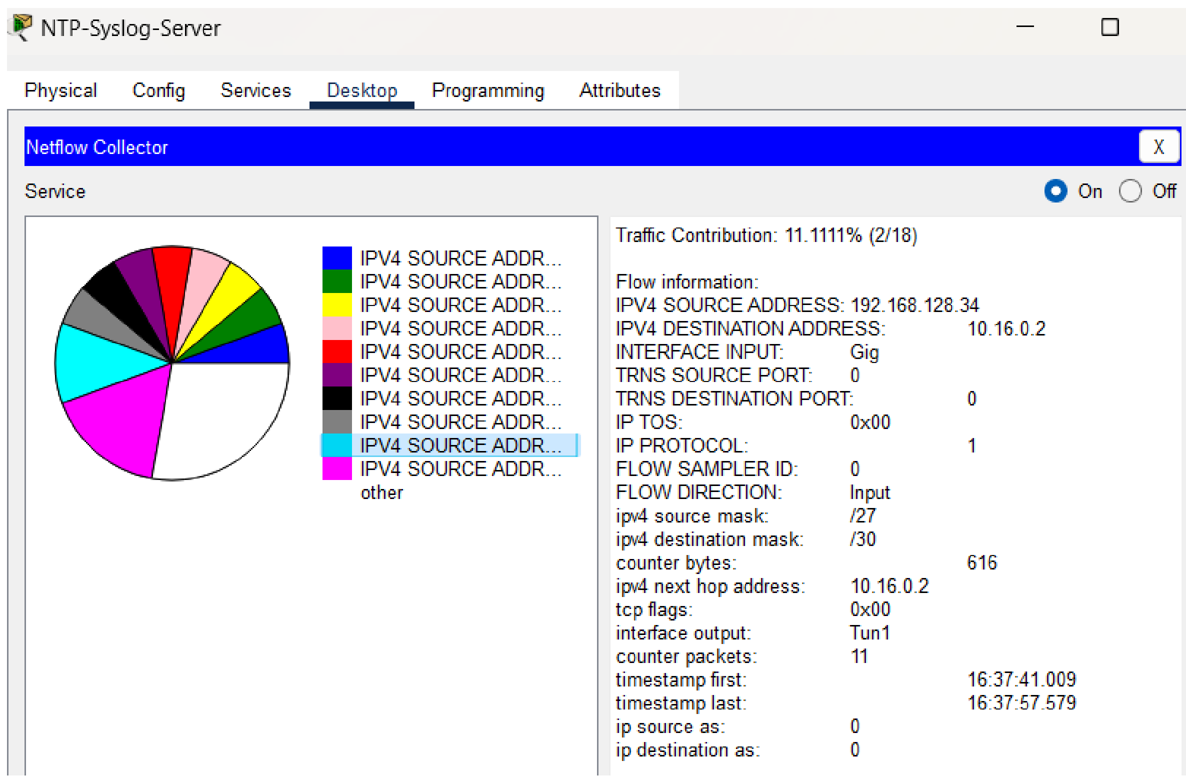


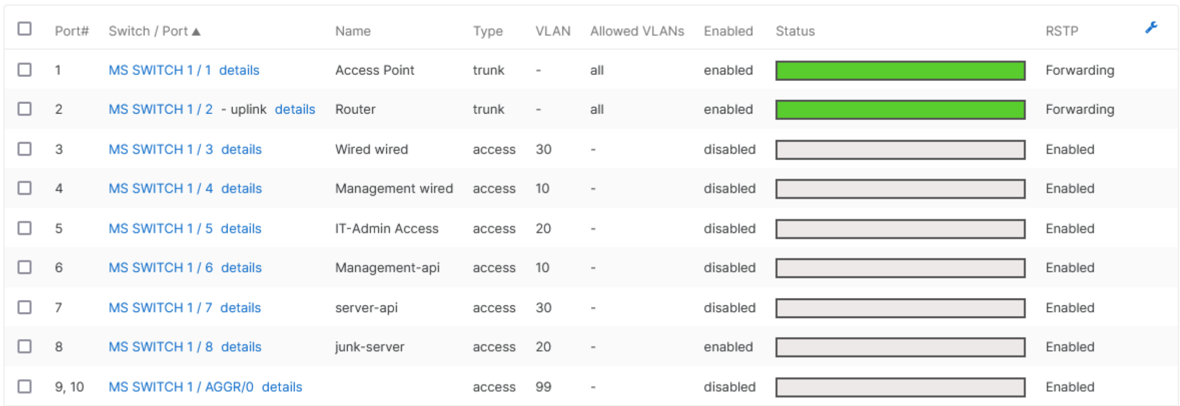
Figure 41: Netflow collector application on NTP-Syslog server

Since Packet Tracer is a simulated environment, we were limited in terms of how we could utilize the data collected by Netflow. We were able to set up the generation of Netflow records as traffic enters the router's sub-interfaces, and view the information using the 'Netflow Collector' application on the server's 'Desktop' tab. However, without any additional tools or programs to analyze the collected data, as we would have had in a real traditional network, it becomes very difficult and time consuming to accurately interpret the presented data of each traffic flow to identify any anomalies.

4.2.2 Using Meraki

As explained in Section 3.4.2, the simplest way to gather information about the state of the system is to use the built-in dashboard feature of Cisco Meraki. The dashboard offers several different status-pages spread around the interface under related sections, which provides a wide range of information about the network.

For example, the status page for active switch ports is located in the "Monitor" section under the "Switching" tab of the taskbar. Most of these pages list graphs, other graphics, or tables with information. Many of these tables can be altered using the configuration menu, visible at the top-right corner in Figure 42, and can usually be downloaded as well.



Port#	Switch / Port ▲	Name	Type	VLAN	Allowed VLANs	Enabled	Status	RSTP
1	MS SWITCH 1 / 1 details	Access Point	trunk	-	all	enabled		Forwarding
2	MS SWITCH 1 / 2 - uplink details	Router	trunk	-	all	enabled		Forwarding
3	MS SWITCH 1 / 3 details	Wired wired	access	30	-	disabled		Enabled
4	MS SWITCH 1 / 4 details	Management wired	access	10	-	disabled		Enabled
5	MS SWITCH 1 / 5 details	IT-Admin Access	access	20	-	disabled		Enabled
6	MS SWITCH 1 / 6 details	Management-api	access	10	-	disabled		Enabled
7	MS SWITCH 1 / 7 details	server-api	access	30	-	disabled		Enabled
8	MS SWITCH 1 / 8 details	junk-server	access	20	-	enabled		Enabled
9, 10	MS SWITCH 1 / AGGR/0 details		access	99	-	disabled		Enabled

Figure 42: Switch ports status page in dashboard

The dashboard also has a summary report page for the whole organization containing a compact overview of some key information about the network and its clients as seen in Figure 43. These reports can be scheduled to be sent via email on a regular basis in addition to other email report options such as when the WAN connection of the router fails [37]. However, there is no centralized page, report, or location on the dashboard that presents all the available information on the network, its configurations, connections, clients, or devices; nor is there a option for creating such a page, meaning that navigating the dashboard's various pages is nearly unavoidable.

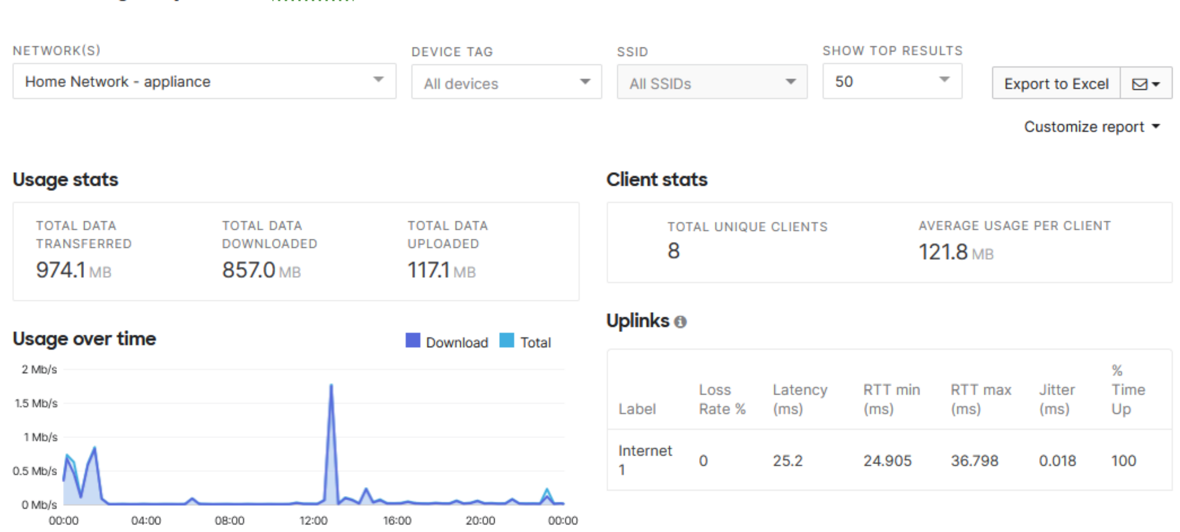


Figure 43: One variant of the summary report page

In addition to the dashboard, there is also the dashboard API, as mentioned in Section 3.4.2. The API allows administrator users to gather information using get-calls instead of

accessing the dashboard to view the information. This allows the forwarding of information either into third-party software, or for further use in API configuration calls. Below is a code example for fetching the names of the devices on the network, and adding them to a list. Figure 44 shows the function's response.

```python

## Get device names
def getDeviceNames(session, device_list):
    device_names = []
    for device in device_list:
        device_get = session.devices.getDevice(device)
        device_names.append(device_get["name"])
    return device_names

```

```
['MX ROUTER', 'MS SWITCH 1', 'MR WIFI AP 1']
```

Figure 44: Response from the getDeviceNames function

One of the advantages of using the API method is that it can be used to change configurations using the same system. The automation of the system can very much be improved upon through the integration of get-calls being fed into create and update calls, allowing more dynamic implementations of network configuration. This will be further discussed in Section 4.3.

It is also possible to use a more traditional monitoring option in the form of SNMP, as configured in Section 3.3.3 and mentioned in Section 3.4.2. Devices can be polled either through the Meraki cloud using the supplied MIB file from their settings, or by polling the devices directly. Figure 45 shows an example of a MIB get-command sent to the MS-switch's local IP address.

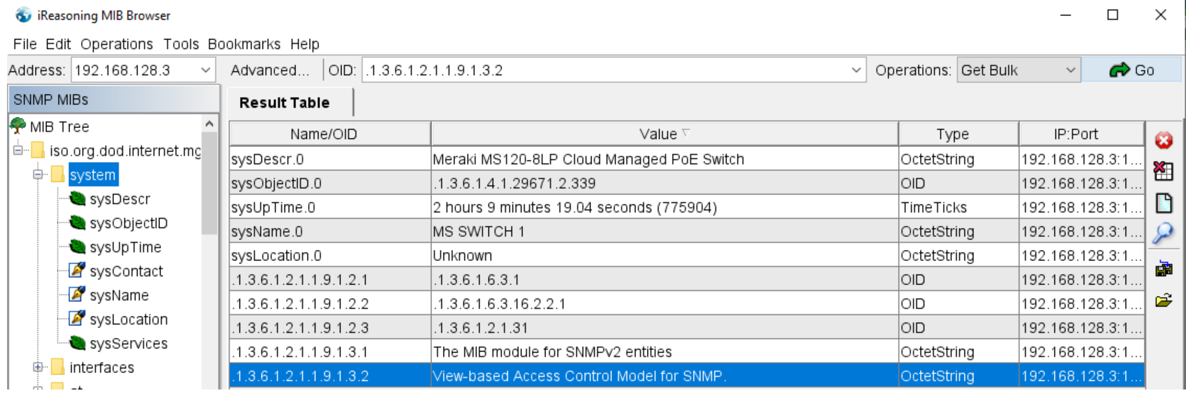


Figure 45: An example of a MiB get-command to the Meraki switch

The SNMP method differs from the other methods in that it can request information directly from the devices regarding their state. This is important since the dashboard is dependant on the administrator having an internet connection and is not usable without it. Additionally, in our experience, there was often a slight delay between what was presented in the dashboard and the actual state of the devices or network, meaning that SNMP could be used for getting accurate information from the devices.

Also in our experience, when a device lost connection to the internet, the device's LED light would represent the issue almost instantly, but it would take five to seven minutes for the dashboard to show the change. During our testing, we found the MX router's uplink traffic information to be the most reliable way of knowing whether the network was working properly. This information is gathered and presented on the dashboard in real-time, and, by tracking it, we were able to spot connectivity issues with the network before the dashboard would display it.

Figure 46 depicts uplink traffic during normal operations, in which the graph will be shifting as it updates every few seconds. The graph will become static as the updates cease during an outage, and eventually the connection will be lost, as depicted in Figure 47.

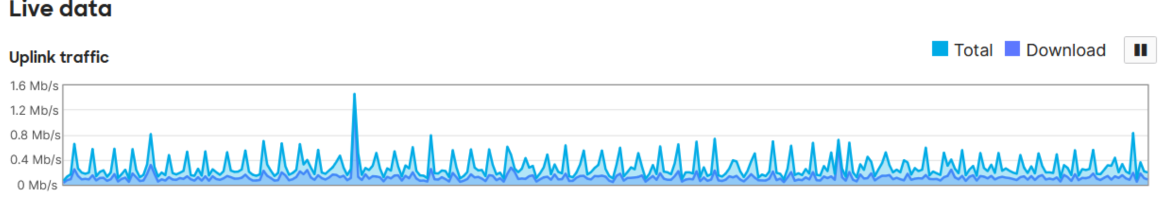


Figure 46: Uplink traffic information during normal operations

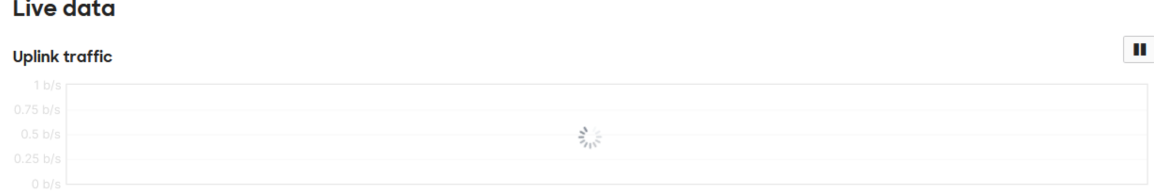


Figure 47: Uplink traffic information during an outage

4.3 Automation Potential

We stated in Section 3.5.1 that we did not have many options with regards to automation for our traditional legacy network. We were able to compile our configurations in one place for later use, which saved us some time when a similar configuration needed to be made on a different device, but that was the extent to which we were able to utilize anything resembling automation.

Outside of an emulated environment like ours, it would be possible to utilize simple Bash scripts or third-party tools to automate certain parts of our configurations, but this was not possible for us to test in Packet Tracer. The remainder of this section will hence focus mainly on the automation potential of our Cisco Meraki network.

As we previously presented in Section 3.5.2, we enabled access to the dashboard API and generated the API-key necessary for using it. For the initial API testing process, we used the Postman application to run some HTTP GET-requests, first to confirm that the API was working and that we could establish a connection, and then for testing the Meraki

collection available for Postman [38]. Although Postman was a practical solution to begin with, we decided to use Meraki's Python library [35] instead, both at our supervisor's suggestion and because we realized it would provide us with higher levels of flexibility.

4.3.1 Automating Use Case 1

For this section, we will review snippets of the main Python script (available in its entirety in Appendix A.2) used to automate the configurations we made when implementing our first use case in Meraki (sec. 3.3.3). Values that are hard-coded in the script, such as network name and device serial numbers, can be altered as necessary and the script can be used repeatedly. In a more elaborate version, these values would simply be imported from configuration files, and be more easily substituted by the need of the user.

We used functions from the imported Meraki SDK module, which are used in separate files from the main script presented here. These functions can be viewed in Appendix A.2.2. For the sake of simplicity, we will not show them in this section.

The first section in the following code snippet is for importing the modules used in the script. The *meraki* module is the library mentioned previously, which gives us access to the simplified API calls designed by Meraki. The *pprint* module is used as it gives a neater output of lists in the terminal compared to regular print, but is not necessary for the the script to function. The *script_functions* folder is where we stored all the functions used in this script, which is done for ease of use and organizing. The functions themselves will not be included in this review, but can be found in the Appendix A.2.2.

```
1
2   # Modules
3   import meraki
4   from pprint import pprint
5   from script_functions import acl, devices, mx_ports, \
6   network, organization, ssid, switch_ports, vlan, snmp
7
```

The variables defined in this section will be relevant for the rest of the script. Other function-specific variables will be created before the function callback, since this was deemed more practical for presentation purposes. The API-key is necessary for establishing a connection with the dashboard API, and at least one of the remaining variables (organization, network, and devices) are required as arguments by all functions in the script.

For security reasons, sensitive values such as the API-key and the real device serial numbers are replaced with simple text.

```
1
2   ### Session Variables
3   # API-KEY
4   MERAKI_DASHBOARD_API_KEY = "API-Key-from-dashboard"
5
6   # Organization name
7   ORGANIZATION_NAME = "NTNU"
8   ORGANIZATION_ID = ""
9
10  # Network Name and id
11  NETWORK_NAME = "Home Network"
12  NETWORK_ID = ""
13
14  # Device Serials:  Router, Switch, AP
15  DEVICES_NAMED = [
16      {"device_serial":"MX-device-serial", "device_name":"MX-Router"},
```

```
        {"device_serial":"MS-device-serial", "device_name": "MS Switch 1"},
        {"device_serial":"MR-device-serial", "device_name":"MR WIFI AP 1"}
        ]

    DEVICES = [
        DEVICES_NAMED[0]["device_serial"],
        DEVICES_NAMED[1]["device_serial"],
        DEVICES_NAMED[2]["device_serial"]
    ]

```

The snippet below is used to establish a connection with the API, which we assign to the variable `dashboard`. This is what we will use as the argument for all `session` parameters in the function callbacks throughout the script.

```

    # Start session
    dashboard = meraki.DashboardAPI(api_key=MERAKI_DASHBOARD_API_KEY, suppress_logging=True)

```

Here, we have put in a set of variables used to activate select parts of the script. The purpose is to allow partial implementation of the script based on the user's needs. It also serves as a brief description for which general configurations are performed in the script, and where.

```

    # script parts activation - For ease of use
    network_exist = False
    network_create = True
    device_assign = True
    device_name_assign = True
    device_ip_assign = True
    router_config = True
    switch_port_config = True
    router_port_config = True
    vlan_config = True
    ssid_config = True
    acl_update = True
    snmp_config = True

```

In the snippet below, the `ORGANIZATION_NAME` variable is used to retrieve the corresponding ID, which is used when creating the new network, and for enabling organization wide SNMP access at the end of the script. The network ID is defined here only in case the network we are trying to interact with already exists. The `getOrganizationId` function borrows a loop function from [39].

```
    ### Variables

    ## Organization variables retrival
    ORGANIZATION_ID = str(organization.getOrganizationId(dashboard,ORGANIZATION_NAME))

    ## Network variable retrieval
    if (network_exist == True):
        NETWORK_ID = network.getNetworkId(dashboard, ORGANIZATION_ID, NETWORK_NAME)


```

In this part of the script, we define the variables used to create a network, and call the create function. The product types need to be defined in cases where the network

contains more than one type of device. The `CreateNewNetwork` function returns the network as an object, from which we assign the ID to `NETWORK_ID`.

```

    ## New Network variables
    # Product types are what type of devices should be in the new network
    PROD_TYPES = ["appliance", "switch", "wireless"]

    ### create network
    if network_create == True:

        new_net = network.CreateNewNetwork(
            session=dashboard,
            org_id=ORGANIZATION_ID,
            net_name=NETWORK_NAME,
            prod_types=PROD_TYPES,

        NETWORK_ID = new_net['id']

```

In the following snippet, the ”assignDevices” function claims devices into the created network, using the `DEVICES` variable that was defined in the beginning of the script. The ”nameDevices” function assigns names to each device from the `DEVICES_NAMED`, which contains the serial number and name for each device. The devices' names are connected to their serial number, and will persist until changed even if moved to a different network.

```

    ###  Assign devices to network
    if (device_assign == True):
        assigned_devices = devices.assignDevices(dashboard, NETWORK_ID, DEVICES)
        pprint(assigned_devices)


    ### Name devices
    if (device_name_assign == True):
        device_named = devices.nameDevices(dashboard, DEVICES_NAMED)
        pprint(device_named)

```

In Meraki, there is a single-LAN setting by default, with the subnet 192.168.128.0/24. When we enable VLANs, this single-LAN is turned into VLAN 1 for native. Since this VLAN is not in our addressing plan, it can either be modified or deleted in order to utilize the subnet address elsewhere. Due to overlapping address rooms, we use the `updateVlan1` function to alter its subnet, before deleting it after the other VLANs have been configured.

For the `createVlans` function, we inserted a list of all the VLANs we wish to configure. For each VLAN, a subnet is configured, and we choose whether DHCP is mandatory.

```

    ### New network content
    ## VLANs, subnets, and DHCP
    if (vlan_config == True):

        new_vlans_dict=[
            {
            'vlan_id': '10',
            'vlan_name': 'Management-api',
            'vlan_subnet': '192.168.128.0/27',
            'gateway_ip': '192.168.128.1',
            'mandatory_dhcp': {"enabled": False}
            },
            {
```

```
            'vlan_id': '20',
            'vlan_name': 'IT-admins-api',
            'vlan_subnet': '192.168.128.32/27',
            'gateway_ip': '192.168.128.33',
            'mandatory_dhcp': {"enabled": False}
            },
            {
            'vlan_id': '30',
            'vlan_name': 'Employees-api',
            'vlan_subnet': '192.168.128.64/26',
            'gateway_ip': '192.168.128.65',
            'mandatory_dhcp': {"enabled": False}
            },
            {
            'vlan_id': '40',
            'vlan_name': 'Guest-api',
            'vlan_subnet': '192.168.128.128/25',
            'gateway_ip': '192.168.128.129',
            'mandatory_dhcp': {"enabled": True}
            }
        ]


        # turn on VLAN
        vlan_enabled = vlan.enableVlan(dashboard, NETWORK_ID, True)

        # To handle vlan 1 special case
        vlan.UpdateVlan1(dashboard, NETWORK_ID)

        # new vlans
        new_vlans_response = vlan.createVlans(dashboard, NETWORK_ID, new_vlans_dict)
        pprint(new_vlans_response)

        #delete vlan 1
        vlan.deleteVlan(dashboard, NETWORK_ID, "1")

        pprint(vlan.getVlanStatus(dashboard, NETWORK_ID))

```

In the following snippet, we define a management VLAN which is assigned to the switch using the function `ipSwitchVlanManagement` and the defined variable. The second function `ipDeviceAll` is called to assign a VLAN to all network devices from which they can get an IP-address automatically. The difference between the two is that the `ipSwtichVlanManagement` assigns a VLAN to all the switches in the network, which is used to specify which VLAN the STP protocol messages are to be sent to. This is only necessary since we removed native VLAN and have chosen to drop untagged traffic.

```

    ### Configure ports and management IPs
    ## Assign VLAN and IP to management devices
    if (device_ip_assign == True):
        device_management_vlan = {
            "wan1": {
                "UsingStaticIp": False,
                "vlan": 10
            }
        }

        # switch management IP
        devices.ipSwitchVlanManagement(dashboard, NETWORK_ID, 10)

        devices.ipDeviceAll(dashboard, DEVICES_NAMED, device_management_vlan)
```

```
16
```

For the following snippet, we configured all the router ports with appropriate values. In most cases, the ports should be disabled until they are in use and should only be accessed by IT-admins.

```

    ## Router ports
    if (router_port_config == True):

        ## VLAN variables for ports
        router_port_values=[
            {
            "port_id" : "2",
            "port_enable" : False,
            "port_type" : "access",
            "vlan_id" : 20,
            "allowed_vlan_list" : "20"
            },
            {
            "port_id" : "3",
            "port_enable" : False,
            "port_type" : "access",
            "vlan_id" : 20,
            "allowed_vlan_list" : "20"
            },
            {
            "port_id" : "4",
            "port_enable" : False,
            "port_type" : "access",
            "vlan_id" : 20,
            "allowed_vlan_list" : "20"
            },
            {"port_id" : "5",
            "port_enable" : True,
            "port_type" : "trunk",
            "vlan_id" : "",
            "allowed_vlan_list" : "all"
            }
        ]

        router_port_updated = mx_ports.updateMxPortArray(dashboard,NETWORK_ID, router_port_values)
        pprint(router_port_updated)

```

We do a similar configuration for the switch, where we assign trunk ports to the router and the WAP, while all other ports are set to access mode. We also disable PoE for most ports, except for the WAP. In the `updateSwitchPort` function, we provide the target switch and the list of ports to update.

```

    ## Switch Ports
    if (switch_port_config == True):

        # Switch port variables
        update_ports_list=[
            {
            'port_id': '1',
            'port_name': 'WAP-api',
            'port_enabled': True,
            'poe_enabled': True,
            'port_type': "trunk",
```

```
13                  'allowed_vlan': "all"
14                  },
15                  {
16                  'port_id': '2',
17                  'port_name': 'Router-api',
18                  'port_enabled': True,
19                  'poe_enabled': False,
20                  'port_type': "trunk",
21                  'allowed_vlan': "all"
22                  },
23                  {
24                  'port_id': '3',
25                  'port_name': 'Employee-api',
26                  'port_enabled': True,
27                  'poe_enabled': False,
28                  'port_type': "access",
29                  'vlan_id': 30
30                  },
31                  {
32                  'port_id': '4',
33                  'port_name': 'Employee-api',
34                  'port_enabled': True,
35                  'poe_enabled': False,
36                  'port_type': "access",
37                  'vlan_id': 30
38                  },
39                  {
40                  'port_id': '5',
41                  'port_name': 'Employee-api',
42                  'port_enabled': True,
43                  'poe_enabled': False,
44                  'port_type': "access",
45                  'vlan_id': 30
46                  },
47                  {
48                  'port_id': '6',
49                  'port_name': 'Employee-api',
50                  'port_enabled': True,
51                  'poe_enabled': False,
52                  'port_type': "access",
53                  'vlan_id': 30
54                  },
55                  {
56                  'port_id': '7',
57                  'port_name': 'Management-sensor-api',
58                  'port_enabled': False,
59                  'poe_enabled': True,
60                  'port_type': "access",
61                  'vlan_id': 10
62                  },
63                  {
64                  'port_id': '8',
65                  'port_name': 'Server-api',
66                  'port_enabled': True,
67                  'poe_enabled': False,
68                  'port_type': "access",
69                  'vlan_id': 20
70                  },
71                  {
72                  'port_id': '9',
73                  'port_name': 'unavailable-api',
74                  'port_enabled': False,
75                  'poe_enabled': False,
76                  'port_type': "access",
```

```
            'vlan_id': 30
            },
            {
            'port_id': '10',
            'port_name': 'unavailable-api',
            'port_enabled': False,
            'poe_enabled': False,
            'port_type': "access",
            'vlan_id': 30
            }
        ]

    # Switch port function call
    update_switch_ports = switch_ports.UpdateSwitchPort(
        dashboard,
        DEVICES_NAMED[1]["device_serial"],
        update_ports_list
    )
    pprint(update_switch_ports)

```

In the following snippet, we define variables for the SSIDs in the network, before using them in the `UpdateSsidList` function. Each of the user-VLANs we configured earlier are assigned their own SSID.

```

    ### SSID
    if ssid_config == True:

        # SSID variables
        ssid_list=[
            {
            'ssid_number': '1',
            'ssid_name': 'IT-admins-api',
            'ssid_enabled': True,
            'ssid_use_vlan': True,
            'ssid_default_vlan_id': int(20),
            'ssid_psk': "adminPassword",
            },
            {
            'ssid_number': '2',
            'ssid_name': 'Employee-api',
            'ssid_enabled': True,
            'ssid_use_vlan': True,
            'ssid_default_vlan_id': int(30),
            'ssid_psk': "employeePassword",
            },
            {
            'ssid_number': '3',
            'ssid_name': 'Guest-api',
            'ssid_enabled': True,
            'ssid_use_vlan': True,
            'ssid_default_vlan_id': int(40),
            'ssid_psk': "guestPassword",
            }
        ]

        # SSID function call
        new_ssid = ssid.updateSsidList(
            dashboard,
            NETWORK_ID,
            ssid_list)
        pprint(new_ssid)
```

```
39
```

In this snippet, we define the ACL rules in a list variable, before applying them using `updateAcl`. The function used will replace all existing rules with the ones supplied here. The rules are implemented in the order they are organized, from top to bottom.

```
    ###  ACL creation, Standard and Extended
    if acl_update == True:
        ## ACL rules
        NETWORK_ACL_RULES = [
            {
                'comment': 'Deny SSH to Management',
                'dstCidr': '192.168.128.0/27',
                'dstPort': 22,
                'ipVersion': 'ipv4',
                'policy': 'deny',
                'protocol': 'tcp',
                'srcCidr': 'any',
                'srcPort': 'any',
                'vlan': 'any'
            },
            {
                'comment': 'Deny guest VLAN to Management',
                'dstCidr': '192.168.128.0/27',
                'dstPort': 'any',
                'ipVersion': 'ipv4',
                'policy': 'deny',
                'protocol': 'any',
                'srcCidr': '192.168.128.128/25',
                'srcPort': 'any',
                'vlan': 'any'
            },
            {
                'comment': 'Deny guest VLAN to IT-Admin',
                'dstCidr': '192.168.128.32/27',
                'dstPort': 'any',
                'ipVersion': 'ipv4',
                'policy': 'deny',
                'protocol': 'any',
                'srcCidr': '192.168.128.128/25',
                'srcPort': 'any',
                'vlan': 'any'},
            {
                'comment': 'Deny guest VLAN to Employees',
                'dstCidr': '192.168.128.64/26',
                'dstPort': 'any',
                'ipVersion': 'ipv4',
                'policy': 'deny',
                'protocol': 'any',
                'srcCidr': '192.168.128.128/25',
                'srcPort': 'any',
                'vlan': 'any'
            }
        ]

        # ACL function call
        new_acls = acl.updateAcl(dashboard, NETWORK_ID, NETWORK_ACL_RULES)
        pprint(new_acls)

```

For this last snippet, we enable SNMP for the organization and network. The function `configureSNMP` performs the following three tasks: allowing SNMP access in the organiza-

tion, defining the version and community string for the network, and placing an exception in the firewall for IP-addresses in the management VLAN and IT-Admins VLAN.

```

    ### SNMP configurations
    if snmp_config == True:

        # SNMP variables
        snmp_config_list = {"version_access":"community", "community_string":"communitySNMP"}
        snmp_allow_list = ["192.168.128.71", "192.168.128.0/25", "192.168.128.32/25"]

        # SNMP function call
        snmp_configured = snmp.configureSNMP(
            dashboard,
            ORGANIZATION_ID,
            True,
            NETWORK_ID,
            snmp_config_list["version_access"],
            snmp_config_list["community_string"],
            snmp_allow_list)
        pprint(snmp_configured)

```

4.3.2 Retrieving Information

In the previous section, we demonstrated how to create and set configurations using the dashboard API. All the information presented in the dashboard GUI can also be retrieved using the API.

In Section 4.2.2, we presented an example of how the state of our switch is shown in the dashboard in Figure 42. This information can also be retrieved using the code in the following snippet once the device's serial number is provided. The result of this can be seen in Figure 48. Further examples of similar get-functions can be found in Appendix A.2.3.

```

    def getSwitchPort(session, device_serial):
        # Gets switch ports
        ports_got = session.switch.getDeviceSwitchPorts(device_serial)
        return ports_got


    if(switches_ports==True):
        pprint(
            getSwitchPort(dashboard, DEVICES_NAMED[1]["device_serial"])
        )

```

```
[{'accessPolicyType': 'Open',
  'allowedVlans': 'all',
  'daiTrusted': False,
  'enabled': True,
  'isolationEnabled': False,
  'linkNegotiation': 'Auto negotiate',
  'linkNegotiationCapabilities': ['Auto negotiate',
                                  '1 Gigabit full duplex (forced)',
                                  '100 Megabit (auto)',
                                  '100 Megabit half duplex (forced)',
                                  '100 Megabit full duplex (forced)',
                                  '10 Megabit (auto)',
                                  '10 Megabit half duplex (forced)',
                                  '10 Megabit full duplex (forced)'],
  'name': 'Access Point',
  'poeEnabled': True,
  'portId': '1',
  'portScheduleId': None,
  'rstpEnabled': True,
  'stpGuard': 'disabled',
  'tags': [],
  'type': 'trunk',
  'udld': 'Alert only',
  'vlan': None,
  'voiceVlan': None},
```

Figure 48: First object in response from getSwitchPorts()

4.3.3 Dynamic Response

The script and functions we have presented so far are relatively simple in how they work and how they are utilized. However, for this report, we have only looked at network creation and initial configuration. The actions taken and the need to automate in these scenarios are both practical and important as the scale increases, but, most of the time, it is network operations and performance that will be the primary tasks for network administrators. By combining the get and set requests, we can start automating network behavior.

Consider the usage of bandwidth during a regular workday; at midday there will likely be more strain on the bandwidth capacity than there is early in the morning or late at night. In order to ensure that the user experience is kept satisfactory for most clients, it might be necessary to limit the available bandwidth per client.

With automation, this can be done by retrieving user traffic, and checking whether the performance is above a certain threshold. Should it fail this check, configurations can be automatically implemented to reduce available bandwidth per client. As network usage decreases, the bandwidth throttling can steadily be reduced.

A different example would be to regularly scan ports on the network, and automatically close unused ports. This could help maintain access security by restricting access until requested by users. It would also be possible to dynamically gather data and create a network baseline, against which all network behavior will be compared, and define a set of actions to be taken when a deviation from the baseline occurs.

4.3.4 Modules, Expansions, and Templates

Meraki offers more than the API-access itself. As we saw in our script, we did not have to write our own functions completely, but could instead simplify our efforts by using the Meraki Python library. This library provides yet another layer of abstraction for the network's programmability, making it available to more users by lowering the barrier for entry. In developing our script for this report, once we completed a lightning course for learning Python, we could begin experimenting using the Meraki library and getting an overview of what was available to us in terms of automation and programmability. Documentation for the API was also easy to find in the dashboard under the help menu, in addition to in-editor assistance in the form of auto-complete for functions.

As mentioned earlier, it is also possible to make HTTP-requests directly to the API, in addition to a wide array of other tools and pre-made solutions available. Two examples of this are the Cisco Developer Code Exchange [40], and the Meraki Developer Hub Marketplace [41].

The Code Exchange is a set of curated repositories with projects related to the Meraki products which includes solutions from both Meraki and other third-party developers, such as automation scripts [42] and CLI-tools [43] made for the Meraki dashboard. The repositories listed in code-exchange are licensed either under the Open Source Initiative [44] or Cisco-sample Code [45], which makes them free to use within organizations.

The Meraki Developer Hub Marketplace on the other hand, is a commercial platform, meant to connect third-party developers who wish to sell their services to users of the Meraki products. Software such as the vehicle-plate recognizer [46], which can be linked to Meraki cameras, or the SNMP graphical monitor tools [47] are examples of this.

In addition to automation in terms of API-calls and scripts, the Meraki dashboard has implemented a set of features to reduce the need for manual management. The template [48] system and group policies [49] offer easy deployment and methods of reuse and standardization for certain settings.

The template is mainly a network specific solution, where you bind one or more networks to a configuration template according to its users' requirements. The template can be especially beneficial when several networks are to be made identical, or need to adhere to a set of specific rules.

The group policies are pre-made rules which can be applied to specific entities such as clients, VLANs, or client-device-type, and can be used for traffic shaping and network access. Both the template and the group policies can be used in the dashboard itself, or be combined with various automation methods.

5 Discussion

In this chapter, we discuss our findings from Chapter 4 and provide a comparative evaluation of the two networks solutions we implemented for the purpose of this report. In doing so, we will also answer our research questions from Section 1.2.1:

Research Question 1 - How easy is it for an SMB with limited expertise to deploy, configure, and manage a Cisco Meraki based network compared to a similar network using traditional legacy devices?

Research Question 2 - What is the automation potential of a Cisco Meraki based network compared to that of a traditional legacy network?

5.1 Discussion of The Two Network Solutions

5.1.1 Traditional Legacy Network - Packet Tracer

Based on our results presented in Table 11, the implementation of a network was generally more challenging when using traditional network devices. The main reason for this is that implementing a network feature usually consisted of several steps that would need to be performed in a specific order. However, this also allowed for more granular control over the network as we will discuss later.

The deployment process of a traditional network consists of physically interconnecting the network devices before configuring them. In a real-life deployment, at least one network administrator would need to be physically present on-site to install and mount the devices, ensure proper powering, interconnect them using the correct cables, configure them for the first time, and enable remote access through SSH for the option of remote configurations later. Once remote access has been enabled and secured on each network device, as long as the devices are properly connected and available, and the network administrator is on the same network, they should be able to configure the network devices without needing physical access.

Due to the distributed control planes of traditional network devices, the network administrator would need to manually configure each device individually, which becomes more cumbersome, time-consuming, and prone to simple mistakes, especially when more network devices are involved. Troubleshooting the resulting misconfigurations can also be quite challenging, especially in a traditional network setup, as the network administrator would need to first find where the point of failure is by, for example, checking for a loss of connectivity between devices using means such as checking physical connections, checking for connectivity using ICMP pings or monitoring syslog events for status updates.

When configuring our traditional network in Packet Tracer, our main challenge was knowing which commands to run on the CLI to configure a task, as well as where in the CLI the command needed to be run. This is mainly due to the fact that the devices were configured using low-level Cisco IOS commands, which has a steep learning curve, requiring familiarity with its commands and syntax to properly utilise the different options and features when configuring the network and its functionality.

The lack of any guidance on the interface itself also added to the complexity of the process, resulting in the need to seek help from official documentation, external documentation, and a frequent use of the `?` command to navigate the syntax and acceptable arguments for specific commands. With that being said, the level of complexity and detail required in commands used to complete each task is what gives network administrators more granular control when configuring their network and the policies used within them.

In terms of monitoring and managing a traditional network, we found the configuration of NTP, syslog, SNMP and Netflow to be relatively straightforward, although the interpretation of the information which was retrieved or generated by these protocols proved to be a lot less intuitive compared to what was presented in Meraki's dashboard. The ease of which we could retrieve detailed information about the network devices and their current configuration states using the `show` commands, however, was an aspect we felt Meraki lacked.

5.1.2 Software Defined Network - Cisco Meraki

Our initial impression of Cisco Meraki was somewhat negatively affected by the brown-field issue we experienced during our first attempt at connecting the devices. The issue consisted of the devices having already been registered and given their own separate networks in the Meraki dashboard. These were steps we were meant to take as a part of our initial configuration of the network, and should not have been done automatically. Due to this problem, the devices were unable to connect to the Meraki cloud in order to fetch their firmware configurations, nor could they be configured via the dashboard.

The time we spent troubleshooting this issue was prolonged due to a lack of error feedback that we could use to find the source of the issue, causing us to try anything that we thought would possibly fix it. We tried, for example, to configure the MX router and the MS switch using a physical connection (Ethernet cable), and were able to open their static settings page by going to their IP addresses in a web-browser. These only offered a limited set of options though, such as changing the device's private IP address, and were insufficient for fixing the problem in our case.

We do acknowledge that this issue may have been entirely unique to us, but we believe it to be worthy of mention here because the experience gave us valuable insight into a situation that we would not have been able to emulate ourselves.

During this time, we quickly noticed the extent of a Meraki-based network's dependency on a stable internet connection, which is perhaps the main downside of the solution. Without it, neither the dashboard nor the API can be used to manage the devices because they cannot communicate with the Meraki cloud. Diagnostic information is also unavailable through the dashboard or dashboard API, although it is possible to use SNMP locally, or view the network logs for events that took place before the connection was lost, which might help to a degree.

The dashboard itself is also a vital component, because there is little changes one can do to the network without using the dashboard; even access to the the API must first be enabled using the dashboard's general interface. For us, the importance of the dashboard also raised the question of why administrator accounts are not, by default, required to enable two-factor authentication and are only recommended to do so, given that they are able to access and manage every aspect of the network once logged into the dashboard.

Once the brown-field issue was resolved, we found the experience of using Meraki to be simple and efficient, which is reflected in Table 12. The dashboard's general interface is easy to navigate, its various settings are conveniently listed under appropriate categories, and the information it provides on each page is, for the most part, easily understandable. Some level of exploration is required from the user, however, as not all information or settings are as readily accessible as others.

For example, operations such as DHCP, NAT and the configuration of the VPN tunnel were automatic and heavily abstracted, making specific details about their configurations difficult to find, resulting in a more challenging troubleshooting process. Finding detailed

information about the network's state could also be problematic at times, because all the information is not compiled in a single place and users must look into different pages to find the exact thing they are looking for.

This is not a big issue for SMBs with less experience, because they are not expected to require granular control or want to have an overview of all their network's details, but Meraki's solution is also aimed towards more seasoned users, which is where these issues might make themselves more apparent.

Furthermore, though no prior technical expertise is required to deploy a simple network using Meraki, in our experience, some level of knowledge about network concepts and functionalities is useful for understanding how to best utilize the solution. This is especially true for network automation and the dashboard API, as they unlock a wide range of possibilities for network configuration and management, but users are required to actually know how they can use them in order to maximize their benefits.

5.2 Research Questions

5.2.1 Research question 1

Compared to a traditional legacy network, Cisco Meraki's network solution is one made for being more flexible, user friendly, and easy to use by those with limited experience. A Meraki-based network requires less steps overall for establishing connectivity, as it is a plug-and-play solution in essence, and includes many core functionalities (such as DHCP and NAT) that would need to be manually configured in a traditional network.

The high levels of abstraction created by the Meraki dashboard interface and the centralized controller system further simplifies making changes to the network, as the need for low-level commands and per-device configurations are almost entirely removed. The dashboard also provides relevant guidance and information on nearly all its pages, which contributes to lowering the barrier for entry and allows users to continuously learn about various network functionalities as they explore them.

Management is also made more accessible because, in contrast to a traditional network, a Meraki-based network does not require any additional configuration for monitoring the network, as information about the network and its devices is fetched automatically and within regular intervals. This information is then made available on the dashboard through various charts, graphs, and tables that are easy to read and understand. However, there is also the option of extracting the same information for use in third-party solutions, should this be more desirable.

5.2.2 Research Question 2

In short, Cisco Meraki has a substantially higher automation potential than a traditional network. Although this is partly due to the programmable nature of SDNs, Cisco Meraki provides their users with an array of different tools in order to make network automation accessible and a viable option for any type of network. Not only can these tools be used for network deployment, but they can also be used for dynamically monitoring and managing networks in real-time.

The Meraki dashboard API allows for direct HTTP requests to be made towards the dashboard using widely available platforms such as Postman, or by using appropriate programming languages. A Python library is also publicly available, allowing users to more easily utilize the API for creating their own network applications and scripts according to their requirements.

Third-party network applications, both open source and paid, are additionally available through platforms such as the Cisco Developer Code Exchange and the Meraki Developer Hub, allowing users to easily utilize solutions made by others, as well as sharing their own.

6 Conclusion

6.1 Summary

At the beginning of this thesis, we posed two questions: how easy would it be for an SMB with limited expertise to use Cisco Meraki and what is the automation potential of a Meraki based network, compared to a traditional network?

To address these questions, we first provided some foundational information about core networking concepts, network architecture, and Software Defined Networking. We then introduced the two network solutions we would be comparing, which was a traditional network solution simulated within Packet Tracer, and a Meraki-based network solution, and outlined the criteria for our comparison. These criteria were: ease of configuration and setup, ease of management, and automation.

We then described the use-cases we created to compare the two network solutions. In our first use-case, we focused on the implementation of a single-branch network (LAN), and in our second use-case, we built upon the first by implementing a site-to-site VPN to establish connectivity between two single-branch networks (WAN).

To answer our research questions, we assessed the results of these implementations based on our predefined criteria. Based on our findings, we concluded that the Cisco Meraki-based network is indeed easier to configure and manage than a traditional network. Its user-friendly dashboard GUI provides a streamlined approach to network management, making it suitable for SMBs regardless of their level of expertise.

In terms of automation, we found Cisco Meraki to be versatile and accessible. For showcasing some of the possibilities for automation, we explored how we could automate the deployment of our first use case's network. We achieved this by enabling access to the dashboard API, and created a Python script that would implement our desired network configurations and features by directly communicating with the API.

6.2 Future Work

In this thesis, we have only scratched the surface of Cisco Meraki's capabilities as an SDN solution, as we focused primarily on the ease of implementation for those with limited experience. The next step would be to explore some of Meraki's more elaborate functionalities from the perspective of more experienced users, as well as examining some of its limitations by expanding the testing scope to several branches.

Network automation is another topic we were not able to delve too deeply into, and the true scope of what is possible to achieve with the various tools and mechanisms that are available is worth exploring further.

The extent of Cisco Meraki's security measures, both for securing networks and the dashboard, the threats it faces as an SDN solution, how Meraki themselves mitigate these threats, and how administrators can avoid them are some other vital topics that were left unexplored by us.

Lastly, migration from a traditional network to a software defined network (regardless of vendor), and the challenges of such a change should also be examined, especially as we slowly move away from legacy solutions and adapt to new requirements and technologies.

References

[1] D. Kreutz, F. M. V. Ramos, P. E. Veríssimo, C. E. Rothenberg, S. Azodolmolky, and S. Uhlig, "Software-defined networking: A comprehensive survey," *Proceedings of the IEEE*, vol. 103, no. 1, pp. 14–76, 2015.

[2] A. Ghodsi, S. Shenker, T. Koponen, A. Singla, B. Raghavan, and J. Wilcox, "Intelligent design enables architectural evolution," in *Proceedings of the 10th ACM Workshop on Hot Topics in Networks*, HotNets-X, (New York, NY, USA), Association for Computing Machinery, 2011.

[3] H. Kim and N. Feamster, "Improving network management with software defined networking," *IEEE Communications Magazine*, vol. 51, no. 2, pp. 114–119, 2013.

[4] C. Meraki, "It products and technology." `https://meraki.cisco.com/products/`, October 2022.

[5] Cisco, "Cisco packet tracer - networking simulation tool." `https://www.netacad.com/courses/packet-tracer`, Apr 2023.

[6] MerriamWebster, "Network definition and meaning." `https://www.merriam-webster.com/dictionary/network`, n.d. (Accessed 11.05.23).

[7] S. Laan, *IT Infrastructure Architecture - Infrastructure Building Blocks and Concepts Third Edition*. Lulu Press Inc., 2017.

[8] D. Comer, *Internetworking with TCP/IP.* Pearson Education, Inc., 2014.

[9] R. T. Braden, "Requirements for Internet Hosts - Communication Layers." RFC 1122, Oct. 1989.

[10] J. F. Kurose and K. W. Ross, *Computer networking: A top-down approach*. Pearson, 2017.

[11] ISO/IEC, "Iso/iec 7498-1:1994," Nov. 1994.

[12] C. N. Academy, *Routing and Switching Essentials Companion Guide*. Cisco Press Inc., 2014.

[13] C. N. Academy, *Routing and Switching Essentials v6 Companion Guide*. Cisco Press Inc., 2017.

[14] E. Haleplidis, K. Pentikousis, S. Denazis, J. H. Salim, D. Meyer, and O. Koufopavlou, "Software-Defined Networking (SDN): Layers and Architecture Terminology." RFC 7426, Jan. 2015.

[15] S. H. Haji, S. R. Zeebaree, R. H. Saeed, S. Y. Ameen, H. M. Shukur, N. Omar, M. A. Sadeeq, Z. S. Ageed, I. M. Ibrahim, H. M. Yasin, and et al., "Comparison of software defined networking with traditional networking," *Asian Journal of Research in Computer Science*, p. 1–18, May 2021.

[16] I. Bholebawa and U. Dalal, "Performance analysis of proposed network architecture: Openflow vs. traditional network," *International Journal of Computer Science and Information Security*, vol. 14, pp. 30–39, 03 2016.

[17] J. Mogul and J. Postel, "Internet Standard Subnetting Procedure." RFC 950, Aug. 1985.

[18] R. Droms, "Dynamic Host Configuration Protocol." RFC 2131, Mar. 1997.

[19] J. M. Kizza, *Guide to computer network security*. Springer International Publishing, 2017.

[20] C. Sheng, J. Bai, and Q. Sun, *Software-defined wide area network architectures and technologies*. Data Communication Series, London, England: CRC Press, May 2021.

[21] M. D. Lemmon, J. Ganguly, and L. Xia, "Model-based clock synchronization in networks with drifting clocks," in *Proceedings. 2000 Pacific Rim International Symposium on Dependable Computing*, pp. 177–184, IEEE, 2000.

[22] T. Benson, A. Akella, and D. A. Maltz, "Unraveling the complexity of network management.," in *NSDI*, pp. 335–348, 2009.

[23] J. Pan, S. Paul, and R. Jain, "A survey of the research on future internet architectures," *IEEE Communications Magazine*, vol. 49, no. 7, pp. 26–36, 2011.

[24] S. Alabady, "Design and implementation of a network security model for cooperative network.," *Int. Arab. J. e Technol.*, vol. 1, no. 2, pp. 26–36, 2009.

[25] J. Kindervag, S. Balaouras, *et al.*, "No more chewy centers: Introducing the zero trust model of information security," *Forrester Research*, vol. 3, 2010.

[26] K. Benzekki, A. El Fergougui, and A. Elbelrhiti Elalaoui, "Software-defined networking (sdn): a survey," *Security and communication networks*, vol. 9, no. 18, pp. 5803–5833, 2016.

[27] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "Openflow: enabling innovation in campus networks," *ACM SIGCOMM computer communication review*, vol. 38, no. 2, pp. 69–74, 2008.

[28] S. Jain, A. Kumar, S. Mandal, J. Ong, L. Poutievski, A. Singh, S. Venkata, J. Wanderer, J. Zhou, M. Zhu, *et al.*, "B4: Experience with a globally-deployed software defined wan," *ACM SIGCOMM Computer Communication Review*, vol. 43, no. 4, pp. 3–14, 2013.

[29] J. A. Wickboldt, W. P. De Jesus, P. H. Isolani, C. B. Both, J. Rochol, and L. Z. Granville, "Software-defined networking: management requirements and challenges," *IEEE Communications Magazine*, vol. 53, no. 1, pp. 278–285, 2015.

[30] D. Kreutz, F. M. Ramos, and P. Verissimo, "Towards secure and dependable software-defined networks," in *Proceedings of the second ACM SIGCOMM workshop on Hot topics in software defined networking*, pp. 55–60, 2013.

[31] C. Meraki, "Mx67 and mx68 datasheet." `https://documentation.meraki.com/MX/MX_Overviews_and_Specifications/MX67_and_MX68_Datasheet`, November 2022.

[32] C. Meraki, "Ms120 overview and specifications." `https://documentation.meraki.com/MS/MS_Overview_and_Specifications/MS120_Overview_and_Specifications`, September 2022.

[33] C. Meraki, "Mr36 datasheet." `https://documentation.meraki.com/MR/MR_Overview_and_Specifications/MR36_Datasheet`, October 2022.

[34] iReasoning, "ireasoning mib browser." `https://www.ireasoning.com/browser/help.shtml`. (Accessed 12.05.23).

[35] C. Meraki, “Meraki dashboard-api-python.” https://github.com/meraki/dashboard-api-python. (Accessed 14.05.23).

[36] Postman, “What is postman?.” https://www.postman.com/what-is-postman/.

[37] C. Meraki, “Alerts and notifications.” https://documentation.meraki.com/General_Administration/Cross-Platform_Content/Alerts_and_Notifications, May 2023.

[38] C. Meraki, “Meraki dashboard api - v1.33.0.” https://documenter.getpostman.com/view/897512/SzYXYfmJ, May 2023. (Accessed 14.05.23).

[39] T. H. Simmons, “Toddomation meraki scripts.” https://github.com/toddhsimmons/Toddomation/tree/main/Meraki, January 2023. (Accessed 17.05.23).

[40] Cisco, “Cisco developer code exchange meraki.” https://developer.cisco.com/codeexchange/search/?q=meraki, 2023. (Accessed 14.05.23).

[41] C. Meraki, “Meraki developer hub marketplace.” https://apps.meraki.io/, 2023. (Accessed 14.05.23).

[42] Meraki, “automation-scripts.” https://developer.cisco.com/codeexchange/github/repo/meraki/automation-scripts/. (Accessed 14.05.23).

[43] PackeTsar, “meraki-cli.” https://developer.cisco.com/codeexchange/github/repo/PackeTsar/meraki-cli/. (Accessed 14.05.23).

[44] O. S. Initiative, “Osi approved licenses.” https://opensource.org/licenses/. (Accessed 14.05.23).

[45] Cisco, “Cisco sample code license, version 1.1.” https://developer.cisco.com/site/license/cisco-sample-code-license/, January 2018. (Accessed 14.05.23).

[46] P. Recognizer, “Plate recognizer alpr.” https://apps.meraki.io/en-US/apps/380793/plate-recognizer-alpr. (Accessed 14.05.23).

[47] Datadog, “Datadog.” https://apps.meraki.io/en-US/apps/380817/datadog. (Accessed 14.05.23).

[48] C. Meraki, “Managing multiple networks with configuration templates.” https://documentation.meraki.com/General_Administration/Templates_and_Config_Sync/Managing_Multiple_Networks_with_Configuration_Templates, November 2022. (Accessed 14.05.23).

[49] C. Meraki, “Managing multiple networks with configuration templates.” https://documentation.meraki.com/General_Administration/Cross-Platform_Content/Creating_and_Applying_Group_Policies, November 2022. (Accessed 14.05.23).

A Appendix

A.1 ”Internet” Configuration

The following list of commands were used to configure the ISP-router, static routes between TRD-R and the ISP router, and static routes between OSL-R and the ISP router to emulate the connectivity to an ISP providing internet access to Stooges’ network.

ISP-Router

```
hostname ISP
!
no ip domain lookup
!
ip cef
ipv6 unicast-routing
ipv6 cef
!
interface Loopback0
 ip address 192.0.2.1 255.255.255.255
!
interface GigabitEthernet0/0
 shutdown
!
interface GigabitEthernet0/1
 shutdown
!
interface Serial0/0/0
 ip address 192.0.2.33 255.255.255.252
!
interface Serial0/0/1
 ip address 192.0.2.37 255.255.255.252
!
end
```

TRD-R Static Routing to ISP Router

```
ip route 0.0.0.0 0.0.0.0 192.0.2.33
```

OSL-R Static Routing to ISP Router

```
ip route 0.0.0.0 0.0.0.0 192.0.2.37
```

A.2 Python Scripts

A.2.1 Use case 1 configuration script

All code snippets in this subsection belongs to the main configuration script ”uc1-configure-script.py”. In the following snippet we import modules, define recurring session variables,

and start a session with the API. The "Script activation parts" is included to make the script easier to use.

```

# Meraki repo: https://github.com/meraki/dashboard-api-python
# Some code copied from https://github.com/toddhsimmons/Toddomation/tree/main/Meraki

# Modules
import time
import meraki
from pprint import pprint
from script_functions import acl, devices, mx_ports, \
network, organization, ssid, switch_ports, vlan, snmp

### Session Variables
# API-KEY
MERAKI_DASHBOARD_API_KEY = "API-Key-from-dashboard"

# Organization name
ORGANIZATION_NAME = "NTNU"
ORGANIZATION_ID = ""

# Network Name and id
NETWORK_NAME = "Home Network"
NETWORK_ID = ""

# Device Serials:  Router, Switch, AP
DEVICES_NAMED = [
    {"device_serial":"MX-device-serial", "device_name":"MX-Router"},
    {"device_serial":"MS-device-serial", "device_name": "MS Switch 1"},
    {"device_serial":"MR-device-serial", "device_name":"MR WIFI AP 1"}
    ]

DEVICES = [
    DEVICES_NAMED[0]["device_serial"],
    DEVICES_NAMED[1]["device_serial"],
    DEVICES_NAMED[2]["device_serial"]
]

# Start session
dashboard = meraki.DashboardAPI(api_key=MERAKI_DASHBOARD_API_KEY, suppress_logging=True)

# script parts activation - For ease of use during testing
network_create = False
device_assign = False
device_name_assign = False
device_ip_assign = False
router_config = False
switch_port_config = False
router_port_config = False
vlan_config = False
ssid_config = False
acl_update = False
snmp_config = False

###################################################################
### Variables

## Organization variables retrival
ORGANIZATION_ID = str(organization.getOrganizationId(dashboard,ORGANIZATION_NAME))

## Network variable retrieval
if (network_exist == True):
    NETWORK_ID = network.getNetworkId(dashboard, ORGANIZATION_ID, NETWORK_NAME)
```

```
 
 
```

In the following snippet we assign some variables for the new network, and call the ”createNewNetwork” function using the defined variables as arguments.

```

    ### Network creation

    ## New Network variables
    # Product types are what devices should be in the new network
    PROD_TYPES = ["appliance", "switch", "wireless"]

    # Time zone of new network
    TIMEZONE = "Europe/Berlin"

    # Network notes are not necessary, but provide helpful info
    NETWORK_NOTES = "This network has been created using a script, \
    to show off the power of automation through the Meraki API"


    ### create network
    if network_create == True:

        new_net = network.CreateNewNetwork(
            session=dashboard,
            org_id=ORGANIZATION_ID,
            net_name=NETWORK_NAME,
            prod_types=PROD_TYPES,
            time_zone=TIMEZONE,
            network_notes=NETWORK_NOTES)

        NETWORK_ID = new_net['id']
        # status response from creation
        pprint(new_net)
        print("New network ID:", new_net['id'])

```

In the following snippet the ”assignDevices” function claims devices into the created network. The devices are defined in the session variables. The ”nameDevices” function assigns names to each device from a predefined list.

```

    ###  Assign devices to network
    if (device_assign == True):
        assigned_devices = devices.assignDevices(dashboard, NETWORK_ID, DEVICES)
        pprint(assigned_devices)


    ### Name devices
    if (device_name_assign == True):
        device_named = devices.nameDevices(dashboard, DEVICES_NAMED)
        pprint(device_named)

```

The following snippet defines a set of VLANs in a list, after which they are used in the ”createVlans” function to create these new VLANs with specified configurations. There are also the ”VLAN 1” functions inserted to handle a special case of the default VLAN.

```

    ### New network content
    ## VLANs, subnets, and DHCP
```

```
    if (vlan_config == True):

        new_vlans_dict=[
            {
            'vlan_id': '10',
            'vlan_name': 'Management-api',
            'vlan_subnet': '192.168.128.0/27',
            'gateway_ip': '192.168.128.1',
            'mandatory_dhcp': {"enabled": False}
            },
            {
            'vlan_id': '20',
            'vlan_name': 'IT-admins-api',
            'vlan_subnet': '192.168.128.32/27',
            'gateway_ip': '192.168.128.33',
            'mandatory_dhcp': {"enabled": False}
            },
            {
            'vlan_id': '30',
            'vlan_name': 'Employees-api',
            'vlan_subnet': '192.168.128.64/26',
            'gateway_ip': '192.168.128.65',
            'mandatory_dhcp': {"enabled": False}
            },
            {
            'vlan_id': '40',
            'vlan_name': 'Guest-api',
            'vlan_subnet': '192.168.128.128/25',
            'gateway_ip': '192.168.128.129',
            'mandatory_dhcp': {"enabled": True}
            }
        ]


        # turn on VLAN
        vlan_enabled = vlan.enableVlan(dashboard, NETWORK_ID, True)
        # print("vlan enabled:", vlan_enabled)
        time.sleep(1)

        # To handle vlan 1 special case
        vlan.UpdateVlan1(dashboard, NETWORK_ID)
        time.sleep(1)

        # new vlans
        new_vlans_response = vlan.createVlans(dashboard, NETWORK_ID, new_vlans_dict)
        pprint(new_vlans_response)
        time.sleep(1)

        #delete vlan 1
        vlan.deleteVlan(dashboard, NETWORK_ID, "1")

        pprint(vlan.getVlanStatus(dashboard, NETWORK_ID))

```

In the following snippet we assign define a management VLAN which is assigned to the switch using the function ”ipSwitchVlanManagement” using the defined variable. The second function ”ipDeviceAll” is called to assign a VLAN to all network devices from where they can be assigned an IP-address automatically.

```

    ### Configure ports and management IPs
    ## Assign VLAN and IP to management devices
    if (device_ip_assign == True):
```

```
        device_management_vlan = {
            "wan1": {
                "UsingStaticIp": False,
                "vlan": 10
            }
        }

        # switch management IP
        devices.ipSwitchVlanManagement(dashboard, NETWORK_ID, 10)

        devices.ipDeviceAll(dashboard, DEVICES_NAMED, device_management_vlan)

```

In the following snippet we define variables for all the router ports, which are used in the ”updateMxPortArray” function to set port configurations.

```

    ## Router ports
    if (router_port_config == True):

        ## VLAN variables for ports
        router_port_values=[
            {
            "port_id" : "2",
            "port_enable" : False,
            "port_type" : "access",
            "vlan_id" : 20,
            "allowed_vlan_list" : "20"
            },
            {
            "port_id" : "3",
            "port_enable" : False,
            "port_type" : "access",
            "vlan_id" : 20,
            "allowed_vlan_list" : "20"
            },
            {
            "port_id" : "4",
            "port_enable" : False,
            "port_type" : "access",
            "vlan_id" : 20,
            "allowed_vlan_list" : "20"
            },
            {"port_id" : "5",
            "port_enable" : True,
            "port_type" : "trunk",
            "vlan_id" : "",
            "allowed_vlan_list" : "all"
            }
        ]

        router_port_updated = mx_ports.updateMxPortArray(dashboard,NETWORK_ID, router_port_values)
        pprint(router_port_updated)

```

In the following snippet we define variables for all the switch ports, which are used in the ”UpdateSwitchPortList” function to set port configurations.

```

    ## Switch Ports
    if (switch_port_config == True):
        update_ports_list=[
            {
```

```
6              'port_id': '1',
7              'port_name': 'WAP-api',
8              'port_enabled': True,
9              'poe_enabled': True,
10             'port_type': "trunk",
11             'allowed_vlan': "all"
12             },
13             {
14             'port_id': '2',
15             'port_name': 'Router-api',
16             'port_enabled': True,
17             'poe_enabled': False,
18             'port_type': "trunk",
19             'allowed_vlan': "all"
20             },
21             {
22             'port_id': '3',
23             'port_name': 'Employee-api',
24             'port_enabled': True,
25             'poe_enabled': False,
26             'port_type': "access",
27             'vlan_id': 30
28             },
29             {
30             'port_id': '4',
31             'port_name': 'Employee-api',
32             'port_enabled': True,
33             'poe_enabled': False,
34             'port_type': "access",
35             'vlan_id': 30
36             },
37             {
38             'port_id': '5',
39             'port_name': 'Employee-api',
40             'port_enabled': True,
41             'poe_enabled': False,
42             'port_type': "access",
43             'vlan_id': 30
44             },
45             {
46             'port_id': '6',
47             'port_name': 'Employee-api',
48             'port_enabled': True,
49             'poe_enabled': False,
50             'port_type': "access",
51             'vlan_id': 30
52             },
53             {
54             'port_id': '7',
55             'port_name': 'Management-sensor-api',
56             'port_enabled': False,
57             'poe_enabled': True,
58             'port_type': "access",
59             'vlan_id': 10
60             },
61             {
62             'port_id': '8',
63             'port_name': 'Server-api',
64             'port_enabled': True,
65             'poe_enabled': False,
66             'port_type': "access",
67             'vlan_id': 20
68             },
69             {
```

```
            'port_id': '9',
            'port_name': 'unavailable-api',
            'port_enabled': False,
            'poe_enabled': False,
            'port_type': "access",
            'vlan_id': 30
            },
            {
            'port_id': '10',
            'port_name': 'unavailable-api',
            'port_enabled': False,
            'poe_enabled': False,
            'port_type': "access",
            'vlan_id': 30
            }
        ]

    update_switch_ports = switch_ports.UpdateSwitchPort(
        dashboard,
        DEVICES_NAMED[1]["device_serial"],
        update_ports_list
    )
    pprint(update_switch_ports)

```

In the following snippet we define variables for the SSIDs in the network, before using them in the ”UpdateSwitchPortList” function.

```

    ### SSID
    if ssid_config == True:
        ssid_list=[
            {
            'ssid_number': '1',
            'ssid_name': 'IT-admins-api',
            'ssid_enabled': True,
            'ssid_use_vlan': True,
            'ssid_default_vlan_id': int(20),
            'ssid_psk': "adminPassword",
            },
            {
            'ssid_number': '2',
            'ssid_name': 'Employee-api',
            'ssid_enabled': True,
            'ssid_use_vlan': True,
            'ssid_default_vlan_id': int(30),
            'ssid_psk': "employeePassword",
            },
            {
            'ssid_number': '3',
            'ssid_name': 'Guest-api',
            'ssid_enabled': True,
            'ssid_use_vlan': True,
            'ssid_default_vlan_id': int(40),
            'ssid_psk': "guestPassword",
            }
        ]

        new_ssid = ssid.updateSsidList(
            dashboard,
            NETWORK_ID,
            ssid_list)
        pprint(new_ssid)
```

```

```

In the following snippet we define variables for a list of ACL rules, before applying them by calling the ”updateAcl” function.

```

    ###  ACL creation, Standard and Extended
    if acl_update == True:
        ## ACL rules
        NETWORK_ACL_RULES = [
            {
                'comment': 'Deny SSH to Management',
                'dstCidr': '192.168.128.0/27',
                'dstPort': 22,
                'ipVersion': 'ipv4',
                'policy': 'deny',
                'protocol': 'tcp',
                'srcCidr': 'any',
                'srcPort': 'any',
                'vlan': 'any'
            },
            {
                'comment': 'Deny guest VLAN to Management',
                'dstCidr': '192.168.128.0/27',
                'dstPort': 'any',
                'ipVersion': 'ipv4',
                'policy': 'deny',
                'protocol': 'any',
                'srcCidr': '192.168.128.128/25',
                'srcPort': 'any',
                'vlan': 'any'
            },
            {
                'comment': 'Deny guest VLAN to IT-Admin',
                'dstCidr': '192.168.128.32/27',
                'dstPort': 'any',
                'ipVersion': 'ipv4',
                'policy': 'deny',
                'protocol': 'any',
                'srcCidr': '192.168.128.128/25',
                'srcPort': 'any',
                'vlan': 'any'},
            {
                'comment': 'Deny guest VLAN to Employees',
                'dstCidr': '192.168.128.64/26',
                'dstPort': 'any',
                'ipVersion': 'ipv4',
                'policy': 'deny',
                'protocol': 'any',
                'srcCidr': '192.168.128.128/25',
                'srcPort': 'any',
                'vlan': 'any'
            }
        ]

        new_acls = acl.updateAcl(dashboard, NETWORK_ID, NETWORK_ACL_RULES)
        pprint(new_acls)

```

In the following snippet we define variables for SNMP configuration, before using the ”configureSNMP” to apply them to the network.

```python

    ### SNMP configurations
    if snmp_config == True:

        snmp_config_list = {"version_access":"community", "community_string":"communitySNMP"}
        snmp_allow_list = ["192.168.128.71", "192.168.128.0/25", "192.168.128.32/25"]

        snmp_configured = snmp.configureSNMP(
            dashboard,
            ORGANIZATION_ID,
            True,
            NETWORK_ID,
            snmp_config_list["version_access"],
            snmp_config_list["community_string"],
            snmp_allow_list)
        pprint(snmp_configured)

```

A.2.2 Function imports for use case 1 configuration script

Function for getting organization ID from the cloud using the organization name.

```python

    ## Organization
    # Get organization ID
    def getOrganizationId(session, org_name):
        my_orgs = session.organizations.getOrganizations()

        for listedOrg in my_orgs:
            if listedOrg['name'] == org_name:
                ORGANIZATION_ID = listedOrg['id']

        return ORGANIZATION_ID

```

Functions for creating network with supplied variables, and retrieving network ID when supplied with name.

```python

    ## network
    # create
    def CreateNewNetwork(session, org_id, net_name, prod_types, time_zone, network_notes):

        new_network = session.organizations.createOrganizationNetwork(
            organizationId=org_id,
            name=net_name,
            productTypes=prod_types,
            timeZone=time_zone,
            notes=network_notes)

        return new_network

    # get id
    def getNetworkId(session, org_id, net_name):
        my_networks = session.organizations.getOrganizationNetworks(org_id)
        for entry in my_networks:
            if entry['name'] == net_name:
                return entry['id']

```

Functions for claiming devices into network, and naming devices.

```python

    ## Devices
    # claim
    def assignDevices(session, net_id, device_sn_list):

        assigned_status = session.networks.claimNetworkDevices(net_id, device_sn_list)
        return assigned_status


    # name
    def nameDevices(session, device_array):
        name_assigned=[]
        for device in device_array:
            name_assigned.append(session.devices.updateDevice(
                device["device_serial"], device["device_name"]
                ))
        return name_assigned

```

Functions for toggling VLAN on/off, handling the existing VLAN 1, and creating new VLANs.

```python

## VLANs

    def enableVlan(session, network_id, vlan_bool):
        vlan_enabled = session.appliance.updateNetworkApplianceVlansSettings(
            networkId=network_id,
            vlansEnabled=vlan_bool
        )
        return vlan_enabled


    # update vlan 1
    def UpdateVlan1(session, net_id):
        session.appliance.updateNetworkApplianceVlan(
        networkId=net_id,
        vlanId="1",
        subnet="172.16.255.0/16",
        applianceIp="172.16.255.1"
    )

    # create vlans
    def createVlans(session, network_id, list_dict):
        new_vlans = []
        for vlan in list_dict:
            new_vlans.append(
                session.appliance.createNetworkApplianceVlan(
                    networkId=network_id,
                    id=vlan['vlan_id'],
                    name=vlan['vlan_name'],
                    subnet=vlan['vlan_subnet'],
                    applianceIp=vlan['gateway_ip'],
                    mandatoryDhcp=vlan['mandatory_dhcp']
                )
            )
        return new_vlans

    # delete vlans
    def deleteVlan(session, network_id, vlan_id):
        session.appliance.deleteNetworkApplianceVlan(network_id, vlan_id)

```

Functions for assigning switch management VLAN, and assigning network devices with

IP addresses.

```
## Management VLAN for switch
# Set switch management VLAN
def ipSwitchVlanManagement(session, net_id, vlan_id):
    return session.switch.updateNetworkSwitchSettings(networkId=net_id, vlan=vlan_id)

## Assign management VLAN to network devices for IP
def ipDeviceAll(session, devices, device_object):

    device_vlan_status=[]
    i = 1
    while (i < len(devices)):
        device_vlan_status.append(
            session.devices.updateDeviceManagementInterface(
                serial=devices[i]["device_serial"],
                wan1=device_object
            )
        )
        i+= 1
    return device_vlan_status
```

Function for configuring ports on the router/security appliance.

```
## Router ports
    def updateMxPortArray(
        session,
        net_id,
        port_list,
        drop_untag = True ,
        access_policy = "open"):
            new_ports_status = []

            for port in port_list:
                if (port["PORT_TYPE"] == "access"):
                    drop_untag = False

                if (port["port_type"] == "trunk"):
                    new_ports_status.append(session.appliance.updateNetworkAppliancePort(
                        networkId=net_id,
                        portId=port["port_id"],
                        enabled=port["enabled"],
                        dropUntaggedTraffic=drop_untag,
                        type=port["port_type"],
                        allowedVlans=["allowed_vlan_list"]))

                elif (port["port_type"] == "access"):
                    new_ports_status.append(session.appliance.updateNetworkAppliancePort(
                        networkId=net_id,
                        portId=port["port_id"],
                        enabled=port["enabled"],
                        type=port["port_type"],
                        vlan=port["vlan_id"],
                        accessPolicy=access_policy))

            return new_ports_status
```

Function for configuring ports on switch.

```

## Switch ports

def UpdateSwitchPortList (session, sn, port_list):
    port_changed = []
    for port in port_list:
        if (port['port_type'] == "access"):
                port_changed.append(
                    session.switch.updateDeviceSwitchPort(
                    serial=sn,
                    portId=port['port_id'],
                    name=port['port_name'],
                    enabled=port['port_enabled'],
                    poeEnabled=port['poe_enabled'],
                    type=port['port_type'],
                    vlan=port['vlan_id'],
                    ))
        elif (port['port_type']== "trunk"):
            for port in port_list:
                port_changed.append(
                    session.switch.updateDeviceSwitchPort(
                        serial=sn,
                        portId=port['port_id'],
                        name=port['port_name'],
                        enabled=port['port_enabled'],
                        poeEnabled=port['poe_enabled'],
                        type=port['port_type'],
                        allowedVlans=port['allowed_vlan']
                )
        )
    return port_changed

```

Functions for configuring SSIDs.

```

## SSID

def updateSsidList(session, networkID, ssid_list):
    ssid_new = []
    for ssid in ssid_list:
        ssid_new.append(
            session.wireless.updateNetworkWirelessSsid(
                networkId =  networkID,
                number=ssid['ssid_number'],
                name=ssid['ssid_name'],
                enabled=ssid['ssid_enabled'],
                authMode="psk",
                encryptionMode="wpa",
                psk=ssid['ssid_psk'],
                wpaEncryptionMode="WPA1 and WPA2",
                ipAssignmentMode="Bridge mode",
                useVlanTagging=ssid['ssid_use_vlan'],
                defaultVlanId=int(ssid['ssid_default_vlan_id']),
                visible=True,
                mandatoryDhcpEnabled=True)
        )
    return ssid_new

```

Function for updating the ACL on network.

```python

## ACL

def updateAcl(session, net_id, rule_array):
    update_status = session.switch.updateNetworkSwitchAccessControlLists(
        networkId=net_id,
        rules=rule_array)
    return update_status

```

Functions for configuring SNMP access to organization and network.

```python

## SNMP
def configureSNMP (session, org_id, v2_enable, net_id, version_access, comm_string, allow_list):

    snmp_return = []
    snmp_return.append(
        session.organizations.updateOrganizationSnmp(
            session,
            organizationId=org_id,
            v2cEnabled=v2_enable)
        )
    snmp_return.append(
        session.networks.updateNetworkSnmp(
        session,
        networkId=net_id,
        access=version_access,
        communityString=comm_string)
    )
    snmp_return.append(
        session.appliance.updateNetworkApplianceFirewallFirewalledService(
            session,
            networkId=net_id,
            service="SNMP",
            access="restricted",
            allowedIps=allow_list
        )
    )

    return snmp_return

```

A.2.3 Automation Get-script

The following snippet contains variables and code used to retrieve information from the dashboard API.

```python

    # Modules
    import meraki
    from pprint import pprint
    from finished_scripts import acl, devices, get_client, \
    mx_ports, network, organization, ssid, switch_ports, vlan

    # API-KEY
    MERAKI_DASHBOARD_API_KEY = "API-KEY"

    # Organization name
    ORGANIZATION_NAME = "NTNU"
    ORGANIZATION_ID = ""
```

```
14
15      # Network Name and id
16
17      NETWORK_NAME = "Home Network"
18      NETWORK_ID = ""
19
20      # Device Serials:  SA, Switch, AP
21      DEVICES_NAMED = [
22          {"device_serial":"MX-device-serial", "device_name":"MX-Router"},
23          {"device_serial":"MS-device-serial", "device_name": "MS Switch 1"},
24          {"device_serial":"MR-device-serial", "device_name":"MR WIFI AP 1"}
25      ]
26
27      DEVICES = [
28          DEVICES_NAMED[0]["device_serial"],
29          DEVICES_NAMED[1]["device_serial"],
30          DEVICES_NAMED[2]["device_serial"]
31      ]
32
33
34      # Start session
35      dashboard = meraki.DashboardAPI(api_key=MERAKI_DASHBOARD_API_KEY, suppress_logging=True)
36
37      ###############################################################################################
38
39      # Script activation
40      network_exist = True
41      device_name = True
42      change_log = True
43      router_ports = True
44      Managment_VLAN = True
45      switches_ports = True
46      uplink_status = True
47      bandwidth_usage = True
48
49      ###############################################################################################
50
51      ## Get org id
52      ORGANIZATION_ID = str(organization.getOrganizationId(dashboard,ORGANIZATION_NAME))
53
54      ## Network variable retrieval
55      if (network_exist == True):
56          NETWORK_ID = network.getNetworkId(dashboard, ORGANIZATION_ID, NETWORK_NAME)
57
58      # Get all API requests for organization
59      def getOrgAPI():
60          return dashboard.organizations.getOrganizationApiRequests(ORGANIZATION_ID)
61
62      ## Make device LED blink
63      def deviceLives(session, deviceId):
64          session.devices.blinkDeviceLeds(deviceId)
65
66      ### Get commands for monitoring bit
67
68      def getDeviceNames(session, device_list):
69          device_names = []
70          for device in device_list:
71              device_get = session.devices.getDevice(device)
72              device_names.append(device_get["name"])
73
74          return device_names
75
76      def portApplianceGet(session, net_id):
77          # Appliance ports
```

```
78          ports_got = []
79          ports_got.append(session.appliance.getNetworkAppliancePorts(net_id))
80
81          return ports_got
82
83      def getSwitchPort(session, device_serial):
84          # Gets switch ports
85          ports_got = session.switch.getDeviceSwitchPorts(device_serial)
86
87          return ports_got
88
89      def SwitchManagementVlanGet(session, serial):
90          switch_management_got = []
91          # Switch ports
92          switch_management_got.append(session.devices.getDeviceManagementInterface(serial))
93
94          return switch_management_got
95
96      def routingGet(session, net_id):
97          # Static routes within the network. Currently none.
98          routes_got = session.appliance.getNetworkApplianceStaticRoutes(net_id)
99
100         return routes_got
101
102     def wanConnectionGet(session, org_id, time_span="60"):
103         # Returns appliance device, network, uplink avg.latency, and uplink %loss of packets.
104         # Time is minutes.
105         connection_got = dashboard.organizations.getOrganizationDevicesUplinksLossAndLatency(
106             organizationId=org_id,
107             timespan=time_span)
108
109         return connection_got
110
111     def trafficGet(session=dashboard, net_id=NETWORK_ID, time_span_sec ="216000"):
112         # minimum two hours in seconds = "216000"
113         traffic_got = session.networks.getNetworkTraffic(networkId=net_id, timespan=time_span_sec)
114
115         return traffic_got
116
117     def trafficDestination(session, net_id):
118         # destination for connectivity testing MX
119         traf_dest_got = session.appliance.getNetworkApplianceConnectivityMonitoringDestinations(net_id)
120
121         return traf_dest_got
122
123     def changelogGet(session, org_id, tot_pages, time_span):
124         # Gets all changes made in the organization, for all networks.
125         # Can be narrowed down. Max 365 days back, written in seconds.
126         changelog_got = session.organizations.getOrganizationConfigurationChanges(
127             organizationId=org_id,
128             total_pages=str(tot_pages),
129             timespan=time_span,
130             direction="next")
131
132         return changelog_got
133
134     def getUplinkLossLatency(session, org_id):
135         upLL = session.organizations.getOrganizationDevicesUplinksLossAndLatency(org_id, timespan="180")
136         return upLL
137
138     def getBandwidth(session, net_id):
139         band_got = session.networks.getNetworkClientsBandwidthUsageHistory(net_id, "all")
140         return band_got
141
```

```
142     ### Callbacks for gets
143     ## get device names
144     if(device_name == True):
145         pprint(
146             getDeviceNames(dashboard, DEVICES)
147         )
148 
149     ## Changelog for organization
150     if(change_log == True):
151         pprint(
152             changelogGet(dashboard, ORGANIZATION_ID, "all", 100000)
153         )
154 
155     ## Get ports of router
156     if(router_ports==True):
157         pprint(
158             portApplianceGet(dashboard, NETWORK_ID)
159         )
160 
161     ## Get Management VLAN of switch
162     if(Managment_VLAN==True):
163         pprint(
164             SwitchManagementVlanGet(dashboard, DEVICES_NAMED[1]["device_serial"])
165         )
166 
167     ## Get ports of switch
168     if(switches_ports==True):
169         pprint(
170             getSwitchPort(dashboard, DEVICES_NAMED[1]["device_serial"])
171         )
172 
173     ## Router uplinks loss and latency
174     if(uplink_status==True):
175         pprint(
176             getUplinkLossLatency(dashboard, ORGANIZATION_ID)
177         )
178 
179     ## bandwidth
180     if(bandwidth_usage==True):
181         pprint(
182             getBandwidth(dashboard, NETWORK_ID)
183         )
184 
```

A.3 Best and Worst Cases for Ease of Configuration

Category	Nr	Question	Best Case	Worst Case
Execution	1.1	Was seeking out an external source necessary for completion?	✗	✓
	1.2	Was physical access to a device necessary for execution?	✗	✓
	1.3	Were you provided with any guidance/instructions where the device was configured?	✓	✗
	1.4	Was it clear where you need to go to do the task?	✓	✗
	1.5	Was there any official documentation available to you?	✓	✗
	1.6	*If yes to above:* Was the documentation's information sufficient for completing this task?	✓	✗
Transparency	2.1	Were you given feedback throughout the of this task?	✓	✗
	2.2	*If yes to above:* Did the feedback provide information about the state of the device?	✓	✗
	2.3	Did you encounter any false positive feedback?	✗	✓
Success	3.1	Could you verify your success in completing the task?	✓	✗
	3.2	Was the task completed in a timely manner?	✓	✗
Duration	4.1	Was the majority of your time spent on actually executing the task?	✓	✗
Errors	5.1	Were you given feedback about the error or misconfiguration?	✓	✗
	5.2	Was the feedback sufficient for identifying the source of the error?	✓	✗
	5.3	Was the configuration executed with the error present?	✗	✓

Table 13: Best and worst case results for ease of configuration rubric

NTNU
Norwegian University of
Science and Technology