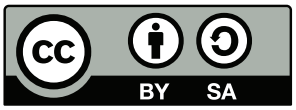

Technische Universität München

Department of Informatics


Bachelor's Thesis in Informatics

# Information Collection for Temporal Variation Analysis on Networks


Marko Dorfhuber



# Technische Universität München

## Department of Informatics

Bachelor's Thesis in Informatics

# Information Collection for Temporal Variation Analysis on Networks

# Methoden zur Erhebung von Netzwerkdaten für eine zeitbasierte Abweichungsanalyse


| | |
|---|---|
| *Author* | Marko Dorfhuber |
| *Supervisor* | Prof. Dr.-Ing. Georg Carle |
| *Advisor* | Nadine Herold, M.Sc. |
| | Dr. Matthias Wachs |
| | Stefan Liebald, M.Sc. |
| *Date* | December 15, 2016 |

Informatik VIII
Chair for Network Architectures and Services


I confirm that this thesis is my own work and I have documented all sources and material used.

Garching b. München, December 15, 2016

Signature

**Abstract**

For security and planning reasons, it is important to know which network components are operating in a network. In addition, changes in the network topology can be indicators for security and performance issues, e.g. if a service is no longer reachable. Therefore, an application which is able to generate network plans automatically and to detect deviations in the network topology is desired.

As network topology information is stored on different devices in the network, like routers, there is no central network component to gather the information from. Thus, we want to combine the information from different specialized modules to get an overall depiction of the network topology.

To be able to store the heterogeneous information from different types of modules in an uniform way, a data model is used. This ensures that we are able to combine the information at one central component. In addition, we develop an approach to detect network topology variations in the central component.

These concepts are implemented in the IT NetworkS AnaLysis And deploymenT Application (INSALATA). INSALATA is able to collect network topology information from different environments at the same time. In addition, the information can be stored persistently and periodic collections are used to map the network topology history. To demonstrate INSALATA, a function test and a case study are performed in the thesis.

**Zusammenfassung**

Um Änderungen an einem Netzwerk planen zu können und Sicherheitsprobleme feststellen zu können, ist es wichtig, alle Netzkomponenten und deren Dienste zu kennen. Zudem deuten Veränderungen in einem Netzwerk auf Performanz- und Sicherheitsprobleme hin. Ein Beispiel hierfür ist ein Dienst, der nicht mehr erreichbar ist. Dadurch ist ein Programm, das Netzpläne automatisch erstellt und Veränderungen erkennt, hilfreich für den Betrieb eines Netzwerkes.

Informationen über die Netzwerktopologie sind nicht an einer zentralen Stelle im Netzwerk gespeichert. Daher werden die Informationen verschiedener Module kombiniert, um einen Netzplan erstellen zu können.

Die unterschiedlichen Module sind auf Netzkomponenten spezialisiert und liefern daher heterogene Informationen. Um die Informationen in einer zentralen Komponente einheitlich verarbeiten zu können, wird ein Datenmodell verwendet. Zudem wird ein Ansatz entwickelt, mit dem Veränderungen im Netzwerk erkannt werden können.

Die entwickelten Konzepte werden in der IT NetworkS AnaLysis And deploymenT Application (INSALATA) umgesetzt. INSALATA ermöglicht es aus verteilten Quellen einen gesamtheitlichen Netzplan zu erstellen und Veränderungen zu erkennen. Um die Arbeitsweise von INSALATA zu testen, wird in der Bachelorarbeit ein Funktionstest und eine Fallstudie durchgeführt.

# Contents

# Chapter 1

# Introduction

For security reasons, it is important to know which network components and services are running in a network. In addition, it is important to recognize, which network components can reach others, e.g. a database server. Therefore, a network plan is required.

Network plans are also important for planning reasons. If a network administrator has to change the network topology because of new or updated hardware, he must have the possibility to plan the changes accordingly to the current network topology.

The generation of network plans is a time-consuming and error-prone task. Further, network plans are often not up-to-date. As a consequence, we do not know exactly which network components or services are operating in the network. This results in a lack of security and complicates planning of future changes. Therefore, a tool which is able to generate network plans automatically is useful. Such a tool could also be utilized to detect configuration mistakes which is not possible with manual collecting.

In addition, undesired changes in the network topology can be indicators for security or performance problems in the network, like a service that is not reachable. We refer to such temporal changes in the network topology as network topology variation. Thus, a continuous tracking of the network topology is helpful to fulfill security requirements.

Summarizing, a tool which is able to generate network plans automatically and which allows the tracking of network topology variations is desired. In this thesis we deal with the question how such an application can be realized.

## 1.1 Goals of the Thesis

The goal of the thesis is to develop an approach for the automated generation of network plans for network topology variation analysis. Therefore, different network topology

information is essential. We develop a data model, to ensure that all relevant information is modeled. The data model is designed to support the analysis and tracking of network topology variations.

In addition, we develop an approach to gather the required network topology information in a network. The developed concept is implemented in a collector application which stores the information compliant to the data model. The network topology information is collected continuously to enable the tracking of network topology variations. The collector application provides an interface to add custom collector modules for specific data.

## 1.2 Research Questions

For the detection and analysis of network topology variations, diverse data is necessary. The following overall research objective has to be answered, to be able to provide the relevant information.

**How can network topology variations be detected?**

We want to provide an approach to detect network topology variations, as this is important for the operating of a network. Therefore, we have to collect required network topology information and store it in a format that supports network topology variation analysis. To be able to develop this approach, we have to answer the following research questions.

**Q1 Which information is required to detect network topology variations?**
We want to detect variations in networks. In this thesis we focus on the network topology. We do not consider the behavior of network components, like packet flow data, as they are not relevant for the network topology.

As the gathering of every available information would lead to confusing network plans, we have to investigate which information is essential for the analysis of the network topology.

**Q2 Which possibilities exist for collecting the required information?**
By answering this research question, we clarify how we can collect the information which is necessary for the depiction of the network topology. In addition, we consider which network topology information is provided by the different types of network components.

We have to determine which collecting techniques are applicable in different network environments. For example it is not possible to gather information in productive networks if this generates much additional traffic. Often direct access, like a SSH connection, is used to collect network topology information. Due to privacy reasons, such methods

can not be used in networks with private data on the network components. Therefore, we have to evaluate the intrusiveness of collecting techniques. In addition, we consider if collecting quality can be increased by combining different techniques.

**Q3 How can temporal network topology variations be traced?**
In this part we clarify how the network topology information collection must be executed to detect temporal network topology variations.

We develop an approach to store the temporal variations persistently for later analysis.

## 1.3 Outline

This section gives an overview over the chapters of the thesis. In Chapter 2 we introduce different methods to gather information about the network topology.

In Chapter 3 we define the terminology for this thesis. We analyze which information is necessary in a mapping of the network topology and develop a general approach how we can combine different sources of information to get an overall depiction. Further, we evaluate the information collection methods of Chapter 2. We take a look at the intrusiveness of the gathering and the quality of network topology information. In addition, we define the requirements of the data model and the collector application.

In Chapter 4 we take a look at some existing solutions for the data model and the collector application. We give an introduction to the data models and the gathering application. In addition, we evaluate the existing solutions using the requirements defined in Chapter 3.

Chapter 5 introduces the components of the system. In addition, we present the design and structure of the collector application to be able to fulfill every requirement of Chapter 3. Afterwards, we define the entities of the data model and their relationships.

In Chapter 6 we take a look at some important parts of the implementation. The used libraries and tools and the implemented collector modules are presented.

In Chapter 7, we verify that the collector application and the data model fulfill the requirements. We perform a function test and a case study to evaluate the quality of the collector application.

Chapter 8 summarizes the thesis and presents future work.

# Chapter 2

# Network Topology Information Collection Methods

In this chapter, we introduce methods to collect network topology information. We consider network scanning and Simple Network Management Protocol (SNMP) and Network Configuration Protocol (NETCONF) as protocols designed for information collection.

## 2.1 Network Scanning

Network scanning is divided in active and passive methods. Passive network scanning monitors the network and analyzes the data other entities produce. Therefore, we do not generate additional network traffic. Passive scanning must be executed on a central network component to have the possibility to see as many packets as possible.

Active network scanning collects information by sending packets to network components and interpreting the answer. As we often have to scan large address ranges, some scans, for example a port scan, are time consuming. As we are able to send packets to network components explicitly, we do not need access to a central network component.

In the following we describe different types of information which can be gathered by network scanning.

### 2.1.1 Layer Three Addresses

We are able to detect layer three addresses using network scanning. Passive detection gathers the addresses of communicating network components using network traffic dumps.

When we execute active layer three address detection, packets are sent to a specified address range. We know that an address is used by a network component if we receive an answer. Examples for active scanning techniques are Internet Control Message Protocol (ICMP) and Address Resolution Protocol (ARP) scanners.

One special form of layer three address detection is the network component detection. Network component detection is able to identify single network components using multiple layer three addresses. This increases the quality of the gathered information.

### 2.1.2 Network Infrastructure

Network infrastructure scanning allows the gathering of information about network components and their linkage. An example for a network infrastructure scanner is *traceroute* which shows the route from the scanning network component to another one. Therefore, we are able to determine the linkage between the network components on the route. Further information can be gathered by direct access to forwarding network components, like routers.

Another part of infrastructure information is the structuring of network components in layer three subnets. It is difficult to gather this information as it is not stored somewhere explicitly.

### 2.1.3 Network Services

Network service detection enables the collection of information about running services on a network component. Port scanners send packets to a network component and determine if the port is opened or closed. Knowing opened ports, we are able to determine running network services as important ones are mostly provided on IANA-assigned ports. [1, 2] Examples for port scanners are *Nmap* [3] and *OpenVAS* [4].

Banner grabbing allows the detection of additional information, like the application offering the network service and its version. For this, packets are sent to the network components and the answers are compared with a fingerprint database. [5]

### 2.1.4 Operating Systems

OS detection software tries to determine the operating system running on a network component. This is possible as the implementations of a protocols, like the Transmission Control Protocol (TCP), the User Datagram Protocol (UDP) and the Hypertext Transfer Protocol (HTTP), are not standardized. Therefore, operating systems set for example header fields, as the TTL, differently.

OS detection tools compare packets from th e scanned network component to database records. If a matching fingerprint is found, the tool is able to determine the operating system with a certain probability. OS detection can be executed using active and passive network scanning.

Due to minor differences in the implementations, it is difficult to determine the operating system exactly without direct access [6]. Important OS detection tools are *Nmap*, *OpenVAS* and *Xprobe2* [7].

### 2.1.5 Firewalls and Applied Rule Sets

Port scanners, like *Nmap*, are able to show filtered ports. This is an indication that a firewall is operating in the network.

There are different methods to determine the used firewall application. Those are the identification of standard rules, processing time and reaction on workload and used algorithms for packet classification. [8, 9]

Using the detected filtered ports, it is able to determine simple firewall rule sets. However, the detection of more complex firewall rules without direct access is almost impossible.

## 2.2 Protocols for Information Gathering

There are protocols designed for requesting network topology information from network components. In the following we present SNMP and NETCONF.

### 2.2.1 SNMP

SNMP [10] allows network management applications to monitor a network component using the `GET` command. In addition, we can control the network component by setting values using the `SET` command. SNMP requires an agent running on the network component which answers the requests and executes the configuration changes. SNMP can be used independently from the type of the network component.

Which values the user is able to request and set is specified by the Management Information Base (MIB) which has a tree structure. SNMP specifies a standard MIB [11]. For extensibility reasons the user or manufacturer of a network component is able to define own MIB modules.

A single value of the MIB is identified by an Object Identifier (OID). For example the `1.3.6.1.2.1.1.1` OID identifies the `sysDesc` value which provides a description of the

entity. Values for *Cisco* network components are defined in the subtree specified by the OID `1.3.6.1.4.1.9`.

SNMPv1 specifies no authentication. It was added in Secure SNMP and the second version of the protocol. The third version allows authentication and integrity checks.

### 2.2.2 NETCONF

The NETCONF [12] protocol allows the gathering of information and configuration of network components. NETCONF divides the retrievable information in configuration data, which is read- and writable, and state data. State data contains information resulting from the configuration of the network component and is therefore read-only.

NETCONF uses the Extensible Markup Language (XML) to specify the configuration of a network component and for the representation of gathered information. Remote Procedure Calls (RPCs) are used for the communication.

Listing 2.1 shows how to gather information using NETCONF. The request is contained in Lines 1-5. We request every data with is available using NETCONF by the namespace specified in the Lines 2 and 3.

Lines 7-13 depict the answer of the network component which contains the requested data. The mapping of request and answer is possible using the `message-id` field. In addition, the identifier of the requested data is specified in the answer (Lines 8 and 9).

```
1  <rpc message-id="101"
2     xmlns="urn:ietf:params:xml:ns:netconf:base:1.0"
3     xmlns:ex="http://example.net/content/1.0">
4     <get/>
5  </rpc>
6
7  <rpc-reply message-id="101"
8     xmlns="urn:ietf:params:xml:ns:netconf:base:1.0"
9     xmlns:ex="http://example.net/content/1.0">
10    <data>
11       <!-- content -->
12    </data>
13 </rpc-reply>
```

Listing 2.1: Example NETCONF Data Request

# Chapter 3

# Analysis

In this section we introduce terminology for this thesis. We evaluate different network topology information gathering methods referring to their information quality and intrusiveness. In addition, we introduce requirements for a data model and a collector application that allows network topology information collection and variation detection.

## 3.1 Terminology

We refer to every device that is part of a network as a *network component*. All network components and the linkage between them is called the *network infrastructure*. Network components only processing data on the first and second ISO/OSI layer, e.g. switches and hubs, are called *passive network components*. We refer to all network components which are able to process data on higher ISO/OSI layers as *active network components*.

In this thesis, we call all information containing CPUs, main memory and storage devices, like HDDs, the *hardware* of a network component. Network components communicate with each other using two types of links. Physical links connect exactly two network components through some medium, e.g. a cable. Logical links, e.g. IP connections, are build using multiple physical links and forwarding network components. Network components can provide different network services, e.g. web, Virtual Private Network (VPN) and File Transfer Protocol (FTP) services. We refer to these network services and the network infrastructure, i.e. links, subnets and network components, as the *network topology*.

The network topology varies over time, e.g. newly provided or no longer reachable services. In the thesis we refer to such changes as *network topology variations*.

We monitor network topology variations in different networks. We will refer to networks whose network topology data is stored in one data structure as one *environment*.

The gathering of network topology information is called *topology information collection*. A *network topology collector* is an application executing topology information collection. A topology collector utilizes *collector modules* for the information gathering. These collector modules use different information collection methods to gather dedicated network topology information. In the thesis the collection of information via login, using protocols like Secure Shell (SSH), Windows PowerShell-Remoting [13] and others, is called *direct access*. Direct access can be gained by username and password or a public key infrastructure.

We call the topology collector implemented in this thesis the *collector application*. The collector application is part of a larger tool. We refer to this tool as the *system*.

## 3.2 Problem Statement and Approach

In this section we present the problem the thesis deals with and the environment the it emerged from. An abstract approach to solve the problem is developed.

### 3.2.1 Environment

The Chair of Network Architectures and Services operates a testbed for students and employees. Several people are working on this testbed simultaneously which leads to difficulties in the configuration. The INSALATA project was introduced to solve these problems.

The goal of the INSALATA project is to reproduce network topologies. One part of the project shall develop an approach to store states of the network topology persistently and to detect network topology variations. We deal with this part of the project in this thesis. In addition, the project shall enable the redeployment of a stored network topology in another environment. An approach to handle this is developed in [14].

### 3.2.2 Approach

In this subsection we develop an approach to map a network topology. The issue of depicting network topologies is that the information is not stored at one central component in the network. If we want to depict the network topology, we have to merge several information sources.

The information we have to process is heterogeneous as different network components operate in a network. Therefore, it is almost impossible to provide a single tool which is able to collect all required information.

We want to use separated modules collecting the network topology information which are specialized on one type of devices, like *Cisco* routers. The modules convert the information to a standardized format and send it to a central component. To get an overall mapping of the network topology, the data received from the modules is merged to one single representation.

We collect the network topology information continuously to be able to update the network plan on changes. This enables the detection of network topology variations at every point of time and we are able reproduce the sequence of those variations.

We define an information model to have the ability to process the collected information in an uniform way. In addition, this ensures that all required information is depicted in the network plan. The information model contains following network topology information.

- **Network Components**
  The information model has to map the network topology and shall allow to analyze network topology variations. Many network topology variations are based on network components which are removed from or added to the network. Therefore, network components are an essential part of the network topology.

  Following additional information must be stored about network components.

  - **Addressing**
    A network component has to identify the recipient of data, by an address. We need multiple layers of this identification as communication can take place in a direct or logical link networks. Therefore, the addresses of a network component on different layers are a crucial part in the information model.

  - **Hardware**
    Hardware information is important for the specification of a network component. It is especially important if we want to detect performance issues and therefore necessary infrastructure changes. An example is required load balancing between two webservers as one of them is not able to process all requests.

  - **Reachability of Network Components**
    Links between network components allow their communication. However, there is the possibility to communicate with network components that are not directly reachable by physical links. To map such behavior we have to depict reachability extending network components, like routers.

    In addition, different network components limit the reachability, e.g. firewalls. This means that reachability limiting network components restrict communication which would be possible. As such devices are important for

the communication and security in a network, we have to map them in the information model.

- **Networks**
  Networks on different ISO/OSI layers are build using multiple physical and logical links. Communication is enabled by networks and therefore they have to be mapped in the information model.

- **Network Services**
  Network services are the reason for communication in the network. Therefore, we have to map them in the information model.

  Some network services are important for the management of the network. Examples are the Dynamic Host Configuration Protocol (DHCP) and the Domain Name System (DNS). We refer to such network services as basic network services. Often we have to store additional information about basic network services. As an example, it is often enough to know that a webserver is operating in the network, but we do not need to store the offered websites. However, the domain of a DNS service is important for its specification. As a consequence, we have to treat many basic network services separately.

## 3.3 Evaluation of Information Collection Methods

In this section we evaluate different information collection methods referring to their information quality and intrusiveness. The criteria for the evaluation are presented in the first part.

### 3.3.1 Criteria

We evaluate the *intrusiveness* of the network topology information collection methods. This is important as high intrusive collection methods must not be used in productive networks. To evaluate this criteria we take a look on following elements.

**Additional Network Traffic** Does the information collecting method generate additional network traffic?

**Direct Access** We evaluate if the method needs direct access to the scanned devices as this is not possible in many network environments.

**Installed Agent** Do we have to install software on the network components to use the considered information collecting method?

Every method is marked with 0, + or ++ for each element in Table 3.1a. We use 0 if the method does not generate additional network traffic or no installation or direct access is necessary. Moderate intrusiveness is marked with +, high intrusiveness with ++.

The second criteria we want to evaluate is the *quality* of information. Therefore, following elements are considered.

**Reliability** Does the information collection method deliver reliable results? We evaluate, for example, if many false-positives are yielded.

**Recentness** We investigate if it is possible to get just-in-time information using this technique and how long it lasts until changes in the network topology are realized.

**Ability to Update Information** Is it possible to update the stored data using this information collection method, e.g. a network component going offline?

**Diversity of Information** Which diversity is offered by the information collection method?

In the evaluation of this criteria we use 0 for bad, + for moderate and ++ good quality in Table 3.1b.

### 3.3.2 Evaluation of Information Collecting Methods

In this section we evaluate different information collecting methods. The results are presented in Table 3.1 and Figure 3.1.

#### 3.3.2.1 Manual Information Collection

This information collection method is least intrusive as no installed client application or access to network components is necessary. Obviously, no network traffic is generated.

The quality of this method is moderate as many mistakes are made. This method is used to depict the network topology in diagrams which are only updated manually if larger changes in the network topology are made. Therefore, undesired network topology variations are not depicted and the recentness is low. In addition, the diversity of information is low.

#### 3.3.2.2 Passive Network Scanning

The intrusiveness of passive network scanning is low, because no installation is required and no additional network traffic is generated. However, access to as many packets as possible is necessary.

The diversity of information is moderate as a passive network scanner only delivers information, the network components are sending. The gathered information can not be updated. For example, a passive network scanner is not able to detect a network component going offline. Passive network scanning is not able to deliver just-in-time information as information can only be collected when it is sent by the network components. However, the reliability is high.

#### 3.3.2.3 Active Network Scanning

Active network scanning generates much additional traffic. However, no access to the target device or installation is required.

Port scanning, for example, determines a service depending on the used port. Therefore, sometimes wrong information is generated as applications can use other ports than the specified ones. Active network scanners deliver the information delayed as a new scan is required to update the information. In addition, the diversity of information is dependent from the used scanner and is limited. Active network scanning is able to update information. Therefore, the quality of information is moderate.

#### 3.3.2.4 Protocols for Information Gathering

In Section 2.2 we presented SNMP and NETCONF as examples for protocols allowing network topology information collection. Both need an installed agent application that is providing the information. However, many network components are equipped with the required agents. This method produces low amounts of additional network traffic as only the required information is transfered. The intrusiveness of this method is higher as using manual collection as we have to gain access to the agents.

The gathered information is collected directly on the network components. Therefore, the reliability is high. Network topology information is gathered delayed as periodic collecting is necessary to get it. This method is able to update information. The diversity of collecting is dependent from the used protocol and client agent.

#### 3.3.2.5 Direct Access

Direct access needs client software enabling the access, like a running SSH server on the network component. However, this software is installed on many network components by default. Direct access generates additional network traffic on connection establishment and for key exchanges. However, compared to active network scanning, the generated network traffic is moderate. Therefore, the intrusiveness of direct access is high.

Direct access can gather information faster as many active network scanning methods as only the login process is time consuming. The information is gathered directly on the network components using configuration files and running processes. Therefore, the reliability and diversity is high and data can be updated. As we only get updates if we retrigger the collection, the recentness is moderate.

#### 3.3.2.6 Agent-based Approaches

Agent-based approaches, like *Zabbix*, produce low amounts of overhead network traffic as mostly data containing the required information has to be transmitted. In addition, no access on the target machine is necessary. However, sometimes authentication is required to allow the collection of information published by the agents.

The quality of information is high, as the network component sends the data. Installed client software could use push notifications which enables just-in-time information gathering. The diversity of gathered information is dependent from the installed application. If an open source or self-made application is used, the collectible information is extensible.

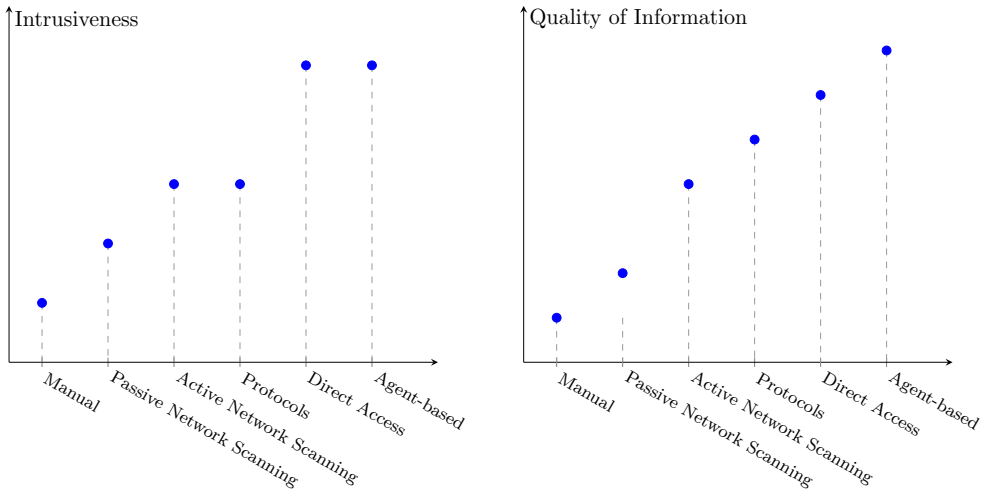
#### 3.3.2.7 Summary

In this section we summarize the results of the evaluation of information collection methods. Table 3.1 and Figure 3.1 show the results of the evaluation. We did not use an explicit metric for the values, but we rate the criteria with 0, + and ++ accordingly to their description.

Table 3.1: Summary of the Evaluation of Collecting Techniques

| | Manual | Passive Scanning | Active Scanning | Protocols | Direct Access | Agent-based Approaches |
|---|---|---|---|---|---|---|
| Additional Network Traffic | 0 | 0 | ++ | + | + | + |
| Clientsoftware | 0 | 0 | 0 | + | + | ++ |
| Direct Access | 0 | + | 0 | 0 | ++ | + |
| Reliability | + | ++ | + | ++ | ++ | ++ |
| Recentness | 0 | 0 | + | + | + | ++ |
| Diversity | 0 | + | + | + | ++ | ++ |
| Updateable | + | 0 | ++ | ++ | ++ | ++ |

In Figure 3.1a we show the intrusiveness of the collecting methods in comparison. In Figure 3.1b the same is done for the quality of gathered information. The values used in the figures result from the number of + in Table 3.1. The results demonstrate that an information collection method is the more reliable the more intrusive it is.




(a) Intrusiveness (b) Quality

Figure 3.1: Evaluation of Information Collection Methods

## 3.4 Requirements

In this section we present the requirements to the data model and the collector application. For each requirement we mention the research question it is assigned to.

### 3.4.1 Data Model

We want to map the network topology information specified in the information model in Section 3.2.2. This leads to the following requirements to a data model used by a network topology collector.

**DM-REQ1 Network Components (Q1)**
We want to map network components in the data model as they are an essential part of the network topology.

**DM-REQ2 Linkage Between Network Components (Q1)**
According to the information model, the data model must have a possibility to depict physical and logical links.

**DM-REQ3 Addressing (Q1)**
The information model defined that we have to map different layers of addressing. As we want to depict ISO/OSI layers one to four in the data model we have to have a possibility to depict layer two addresses and layer three addresses. In addition we must depict ports, which are used as addresses on the fourth ISO/OSI layer.

Layer three addresses also enable the structural arrangement of network components

using subnets. As most networks are separated into subnets, we have to map them in the data model.

**DM-REQ4 Reachability of Network Components (Q1)**
A single mapping of physical and logical links does not depict which communication is possible in the network as different network components can extend or limit the reachability. Therefore, we have to depict reachability extending and limiting network components, like routers and firewalls.

**DM-REQ5 Network Services (Q1)**
Without network services, no communication in the network is necessary. Therefore, we want to map them in the data model.

According to the information model (Section 3.2.2), we have to treat basic network services in a special way.

**DM-REQ6 Hardware Information (Q1)**
As hardware specifications describe essential properties of network components, we have to map them in the data model.

**DM-REQ7 Extensible Information (Q1)**
The presented requirements ensure that the basic network topologies can be mapped. However, users want to have the opportunity to map specific information in the data model, like custom network services. Therefore, the data model has to be customizable and extensible.

**DM-REQ8 Enable Network Topology History (Q3)**
The user wants to track different states of the network topology. This means that we have to provide the ability to reconstruct the network topology for arbitrary points of time. This is important to detect and track network topology variations.

### 3.4.2 Collector Application

In this subsection we display functional requirements to the collector application. We note the research question the requirements are resulting from. Some of the requirements do not result from research questions, but from requirements to the data model. Therefore, the data model requirement is mentioned at these points. If no reference is given for a requirement, it defines a feature of the collector application which does not directly result from a research question or a requirement to the data model.

**SCAN-REQ1 Configurable Collector (Q2)**
Various technologies are used in computer networks. To be able to collect network topology information in as many networks as possible, we must have the possibility to arrange different collector modules dynamically. In addition, we can use collector

modules that are specialized on one type of network component, e.g. a *Cisco* router, which increases the quality of the gathered information.

**SCAN-REQ2 Compatibility to the Data Model (Q1)**
The data model is designed to depict the required information for network topology variation analysis. The network topology collector gathers the required information about the network topology. To ensure that the data can be used in the defined way, the internal data structure of the application must be compatible to the data model.

**SCAN-REQ3 Extensible Collector Modules (DM-REQ7)**
It is not possible to provide collector modules for every infrastructure and network component. To be able to collect the required information from all existing network components, the user has to be able to define custom collector modules. Further, we must provide a possibility to gather custom network topology information (**DM-REQ7**) using the collector application.

**SCAN-REQ4 Periodic Collecting (Q3)**
To track different network topology states, we have to collect the information regularly.

**SCAN-REQ5 Customizable Collecting Interval (Q2)**
To be able to adjust the precision of the topology information collection to the users needs, the interval between the collections must be customizable. In addition, we do not waste resources as we do not have to gather network topology information in static parts of a network as often as in highly dynamic ones.

**SCAN-REQ6 Support Continuous Collecting and Information Adaption (Q2)**
Some collector modules are not able to deliver the gathered information at once, e.g. passive network scanners like *tcpdump*. Such collector modules use information appearing at unpredictable points of time and must therefore be executed permanently.

This requires the ability to add network topology information to the internal data structure asynchronous and not in predefined time slots. In addition, we have to handle race conditions.

Continuous collector modules are often not able to detect disappearing network components. The collector application has to handle such behavior.

**SCAN-REQ7 Support Multiple Environments**
We want to have the ability to track the network topology in different networks simultaneously.

An important use case of INSALATA deals with is the mirroring of network components. To support this, the collector application must be able to collect network topology information in different environments at the same time.

**SCAN-REQ8 Different and Extensible Output Formats (Q3)**
We need to store the network topology information persistently. Therefore, an output

format is necessary. As the data model is extensible (**DM-REQ7**), the output format must be dynamic.

To enable further processing of the network topology information using other applications, we must support the export to different extensible output formats at the same time.

# Chapter 4

# Related Work

In this chapter we present related work for the data model and the collector application and evaluate it using the requirements defined in Section 3.4. The results are presented in the Tables 4.1 and 4.2.

## 4.1 Data Model

In this section we introduce related work for the data model. The design of and use cases for each data model will be presented. In addition, we evaluate if the data models fulfill the requirements defined in Section 3.4.1.

### 4.1.1 IF-MAP

The Interface for Media Access Points (IF-MAP) protocol is used by network components, to share metadata with each other. When someone asks for access to a Media Access Point (MAP)-Client, it must send a request to the MAP-Server. The MAP-Server evaluates the request using the metadata and tells the MAP-Client, if it should grant access.

IF-MAP uses a data model to map network components and requests in the network. This data model contains three elements. *Identifiers* map the entities in a network, like network components or IP addresses. A relationship between two identifiers, e. g. between a network component and its interfaces, is expressed by a *link*. To add attributes to links and identifiers the data model uses the *metadata* entity. [15, 16]

Linkage on different ISO/OSI layers is mapped using links between identifiers. The addresses of a network component is mapped through identifiers representing the address, like an IPv4 identifier. Links assign the address identifier to the network component. Therefore, addressing can be mapped using the IF-MAP data model.

Network services can be depicted by identifiers and links assign them to the identifiers of network components. Service information, like port numbers, can be added as metadata to the links or as identifiers. The same approach allows the mapping of hardware information. IF-MAP's data model is designed to be extensible as you can add new identifiers and metadata [16].

The data model is not able to show reachability limiting elements. The only way to restrict reachability is to delete a link, but we are not able to depict firewalls.

IF-MAP is not designed to store past network topology information. In addition, every identifier exists forever. Therefore, topology variations, like a network component leaving the network, can not be mapped in the data model. [17, 18]

### 4.1.2 A Target-Centric Ontology for Intrusion Detection (IDS Ontology)

Undercoffer et al. [19] designed an ontology for Intrusion Detection Systems (IDSs) and network attack classification. For attack classification a network topology plan is necessary and is mapped using the ontology. It is possible to map the network in dedicated nodes for different entities and relationships are depicted using generic links. In addition, they designed the ontology extensible.

Undercoffer et al. [19] use entities to map network components, but their hardware is not represented. Linkage is depicted by a node called *network*.

Different protocol stacks, like TCP and UDP, are mapped in the data model. However, there is no entity for addressing. As Undercoffer et al. [19] do not consider routers and firewalls, the ontology is not able to depict the reachability of network components.

The ontology has the possibility to map processes, e. g. *httpd* and *sshd*, which can be used to represent network services.

The ontology has no entity to represent the lifetime of a network component. Therefore, the network topology history is not supported.

### 4.1.3 Infrastructure and Network Description Language

The Network Description Language (NDL) [20] is a data model for network topologies that uses XML and Resource Description Framework (RDF). NDL is used to define a desired network topology before applying modifications. After the modifications a snapshot of the network topology is compared to the desired one. Thus, NDL is used for configuration management, fault detection and consistency investigation.

The Infrastructure and Network Description Language (INDL) is an improvement of NDL. [20, 21]

NDL has a layered approach with a *device object* on every layer. Those device objects are abstract nodes which are implemented by virtual nodes. Dedicated virtual nodes are used for different entities in the network. Every device object defines relationships it can be used in, like the *locatedAt* association between *Locations* and *NetworkElements*.

NDL allows the mapping of network components. INDL has a virtual node for the mapping of hardware information.

NDL has the ability to map logical links which can be used to represent physical ones. Therefore, linkage is shown in the data model. However, NDL and INDL are not able to map addresses.

There is no possibility to reflect the reachability between network components, e.g. for firewalls and routing.

NDL considers some network services and it is possible to show the type of the service, like DNS or DHCP. Ports are not represented in the data model, but can be added, as NDL is extensible. [20–22]

### 4.1.4 Network Markup Language

The Network Markup Language Working Group (NML-WG) developed the Network Markup Language (NML) [23] to have a description language for networks which allows the interoperability between different projects. Therefore, NML is standardized and is used by different network monitoring tools. For the data model they deployed a multi layered approach, like NDL. For storage XML combined with RDF is used.

The NML defines nodes for entities in the network and their attributes. In addition, they define relationships between the nodes, like the *hasPort* relationship between *Ports* and *AdaptionServices*.

NML has the possibility to map network components. The data model uses unidirectional ports which allows the depiction of linkage between network components. However, no addresses can be mapped.

Routing tables and firewall rules are not supported by the schema. Therefore, reachability extending and limiting devices/capabilities can not be mapped in the data model.

NML has no entities for storage and CPUs. Thus, no hardware information can be mapped in the data model.

NML defines an abstract *Service* class, but it is not possible to add required information, like ports.

NML has a node called *lifetime* which allows the mapping of appearing and disappearing entities. Therefore, the network topology history is supported by NML. [23–26]

### 4.1.5 Summary

In this section we examined related work of the data model. The results are summarized in Table 4.1. We recognize that no existing data model is able to map all required network topology information, especially **DM-REQ3** and **DM-REQ4**/**DM-REQ8** are not fulfilled by any or only few data models. Fulfilled requirements are marked with ✓and not fulfilled ones with ✗in Table 4.1.

Table 4.1: Evaluation of Existing Solutions for a Data Model

| | IF-MAP | IDS Ontology | INDL | NML |
|---|---|---|---|---|
| **DM-REQ1** Network components | ✓ | ✓ | ✓ | ✓ |
| **DM-REQ2** Linkage between network components | ✓ | ✓ | ✓ | ✓ |
| **DM-REQ3** Addressing | ✓ | ✗ | ✗ | ✗ |
| **DM-REQ4** Reachability of network components | ✗ | ✗ | ✗ | ✗ |
| **DM-REQ5** Network services | ✓ | ✓ | ✓ | ✗ |
| **DM-REQ6** Hardware information | ✓ | ✗ | ✓ | ✗ |
| **DM-REQ7** Extensible information | ✓ | ✓ | ✓ | ✓ |
| **DM-REQ8** Enable network topology history | ✗ | ✗ | ✗ | ✓ |

## 4.2 Network Topology Collection Applications

In this section we examine related work for the collector application. We give an overview over each application and present which tasks it is able to handle. In addition, we reflect if the related work fulfills the requirements specified in Section 3.4.2.

### 4.2.1 IO-Framework

The Interconnected-asset Ontology Framework (IO-F) [27] is developed for risk management in networks which requires that all assets in the network are known. The IO-F collects raw information from managed network components through IO-Collect (IO-C). The IO-C uses the Discovery Interface (DI) to locate managed network components and the Collector Interface (CI) to gather the information using, amongst others, SSH and SNMP. The Interconnected-asset Ontology (IO) is developed to store heterogeneous network topology information in an uniform way.

IO-Write (IO-W) parses the raw input information and stores it to the IO. Therefore, IO-W uses different interfaces to handle heterogeneous information. When the data is

stored to the IO the user can acquire information through IO-Query (IO-Q), which is similar to a database interface.

The IO has the possibility to store layered information. This enables the mapping of network components and linkage. In addition, it contains a layer three forwarding interfaces to map routing and a firewall interface. Network services can be mapped through a layer four address (port) that is connected with a layer three address. Some network services are already defined in the data model, others can be added due to extensibility of IO. Therefore the IO-F is compatible to the data model.

The IO-F collects data from heterogeneous sources. Therefore, IO-C is able to use every collection mechanism that implements the requirements of [28]. In addition, IO-C is extensible as it uses a generic parsing interface.

The IO-F is able to collect information using different collector modules. However, there is no possibility to explicitly exclude a module from information collection.

There is no explicit information, if IO-F supports periodic scanning. According to [27] IO-F is designed to be able to detect inconsistencies between asset behavior and asset configuration. But they do not mention if multiple scans of the network topology are necessary. As a consequence, customizable scanning intervals are not mentioned, too.

The user can export information using IO-Q and store it in another data format. However, the IO-F does not support the export to different output formats by itself

IO-F stores information about one environment in its ontology. There is the opportunity to map different networks, but users can not monitor multiple environments in one instance of the IO-F. [27, 29, 30]

### 4.2.2 cNIS

Common Network Information Service (cNIS) [31] is a project of GÉANT and is designed to be a large scale network topology scanner. It collects network topology information using higher level services, like SSH or SNMP, or the information other tools, like *PerfSONAR*, gathered beforehand.

Network components and their linkage are mapped in the data model. Although, there is no possibility to extend the data model and to map services. Therefore, the used data model in cNIS does not fulfill our requirements (Section 3.4.1).

CNIS stores information about historic network topologies and the current one. Therefore, it is able to scan the network periodically. The user can decide which external tools he wants to use for information collection and implement and load additional plug-ins. It is not mentioned how cNIS handles the merging of different sources of information.

As the tools work independently from cNIS, it is not possible to set a custom collecting interval.

It is possible to scan different networks and filter the information to view only parts of the gathered network topology information using cNIS. Therefore, we are able to view the network topology information for a single environment. [31–33]

### 4.2.3 MonALISA

The California Institute of Technology (Caltech) developed Monitoring Agents using a Large Integrated Services Architecture (MonALISA) [34] for large scale network monitoring. MonALISA uses an assembly of services for information collection which are called monitoring modules. Those gather information about network components and their linkage. The monitoring modules, run independently and are dynamically loadable, if they are required. It is not mentioned how MonALISA handles inconsistent data.

MonALISA uses the information about network components and their linkage to restore the network infrastructure. In addition, performance data about network services is gathered and stored in the data model. Therefore, the used data model is compatible to the requirements defined in Section 3.4.1.

To be able to collect specific network information, the user can define MonALISA monitoring modules. These monitoring modules run periodically, but the user can not set the interval. As network topology information is returned at the end of collection, MonALISA does not support plug-ins that work continuously.

MonALISA allows to visualize parts of the scanned network. Therefore different environments are be supported.

The gathered information is stored in a data base, where the user can get the network topology information from. MonALISA does not define a built-in functionality to output the data in a custom data format. [34–36]

### 4.2.4 PerfSONAR

Performance focused Service Oriented Network Monitoring Architecture (PerfSONAR) is used for performance analysis and operates on the network infrastructure, performance data of services and latency data. It is a distributed application that collects information about network components. The network components push the required data to the PerfSONAR server themselves. The collector modules of the PerfSONAR server use this data to extract the network topology information. It is not discussed how different sources of information are merged in PerfSONAR.

The user has the opportunity to decide which collector modules shall be used by PerfSONAR and implement custom ones. Although, the user often does not have to implement own collector modules for his devices, as PerfSONAR is widely-spread and many organizations provide their collector modules for usage.

PerfSONAR maps network components and links between them. Although, it is not possible to show single services running on a network component. Therefore, the used data model does not fulfill the requirements specified in Section 3.4.1.

To be able to detect networking problems, PerfSONAR has to scan the network periodically. The user often wants to monitor an important link more precisely. To match this user requirement, PerfSONAR is able to handle custom scanning intervals by the use of a scheduler component. As the data is published to the PerfSONAR server by the network components, continuous delivery of information is not supported.

It is possible to monitor multiple networks at the same time. As the user is able to filter the data for a part of the network, PerfSONAR is able to handle multiple environments.

The data is stored in a measurement archive. The application is able to visualize this data, but it is not designed to output the information, as only live and (early) history data is needed for performance analysis. Therefore, output of information is no use case of PerfSONAR. [37, 38]

### 4.2.5 OpenVAS

Open Vulnerability Assessment System (OpenVAS) [4] separated from the commercial penetration testing software Nessus and is open source. It uses a client server architecture for scanning. Client applications are able to send tasks to the server. A task is a request to the server to execute several plug-ins for information gathering. Therefore, the user is able to configure the scanner. The actual scans are executed by the OpenVAS server.

OpenVAS is able to run user defined plug-ins. Those are written in the Nessus Attack Scripting Language (NASL) [39]. This method allows the user to run customized information collection methods. OpenVAS always returns the whole gathered information of all plug-ins. Therefore, they do not have to merge different information sources.

OpenVAS is able to collect information about network components, operating systems and running services. Although, it is not able to detect routers or firewalls. Therefore, the used data model does not fulfill all requirements of the data model.

The user can define that the server shall execute a task periodically. The used interval is customizable, but it is not possible to use a plug-in which is scanning continuously.

A task defines the target and the server delivers the scanning results independently from other tasks. Therefore, multiple networks can be monitored simultaneously.

The result format used by OpenVAS is the only one supported by the server. The output formats can only be extended by editing the source code. [4]

### 4.2.6 Nmap

Network Mapper (Nmap) [40] is a command line tool for penetration testing. It uses active scanning methods to collect network topology information. Nmap is able to execute port scanning, banner grabbing, network component and OS detection and provides different strategies to execute those.

The user is able to choose which scans Nmap shall execute. In addition, the Nmap Scripting Engine (NSE) allows to combine multiple scanning methods. Therefore, Nmap is a configurable and extensible scanner. Nmap executes single network scans and returns all information. Therefore, it does not have to join the network topology information from different collector modules.

Nmap is not intended to be a network topology collector. Therefore, it is not possible to gather every required information for network topology variation detection, e.g. routing tables.

As Nmap is a command line tool for penetration testing, it has no built-in procedure for scanning a network periodically. As a consequence scanning intervals are not supported. Nmap sends requests to network components for information gathering and stops as soon as all network components are scanned. Therefore, Nmap does not support continuous network topology information collection.

It is possible to scan multiple networks using Nmap, but the results are not separated for different environments. Thus, multiple environments are not supported.

Nmap does support different output formats. For example the 'normal' output on the command line is useful for humans. To be able to parse the output of Nmap, it supports XML. But there is no build-in possibility to define own output formats. [3, 40]

### 4.2.7 Summary

In Table 4.2 we summarize the results of this section. In the table is listed which application fulfills a requirement (✓) or does not (✗). We can detect that none of the already existing network topology scanners fulfills all requirements.

Table 4.2: Evaluation of Existing Solutions for a Collector Application

| | IO-Framework | cNIS | MonALISA | PerfSONAR | OpenVAS | Nmap |
|---|---|---|---|---|---|---|
| **SCAN-REQ1** Configurable collector | ✗ | ✓ | ✓ | ✓ | ✓ | ✓ |
| **SCAN-REQ2** Compatibility to the data model | ✓ | ✗ | ✓ | ✗ | ✗ | ✗ |
| **SCAN-REQ3** Extensible collector modules | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| **SCAN-REQ4** Periodic collecting | ✗ | ✓ | ✓ | ✓ | ✓ | ✗ |
| **SCAN-REQ5** Customizable collecting interval | ✗ | ✗ | ✗ | ✓ | ✓ | ✗ |
| **SCAN-REQ6** Continuous collecting and information adaption | ✗ | ✓ | ✗ | ✗ | ✗ | ✗ |
| **SCAN-REQ7** Support multiple environments | ✗ | ✗ | ✓ | ✓ | ✓ | ✗ |
| **SCAN-REQ8** Output format | ✗ | ✓ | ✗ | ✗ | ✗ | ✗ |

# Chapter 5

# Design

The collector application is part of a larger system. As this influences some design decisions, we introduce the system in the first section. In the second section the collector application and its data model are designed.

## 5.1 Overall System – INSALATA

The goal of the INSALATA project is to depict the network topology and to provide the ability to detect network topology variations. In addition, an approach to redeploy a stored network topology is developed.

No existing solution is able to fulfill all requirements. Therefore, we develop the *It NetworkS AnaLysis And deploymenT Application (INSALATA)*.

An overview of the INSALATA system is depicted in Figure 5.1. INSALATA consists of the following three main components.

**Collector Application** The collector application is able to generate network topology plans for different physical and virtual networks and is designed in Section 5.2. The gathered network topology information can be directly used by INSALATA or be stored persistently. It is represented by the *Collector* box on the left side of Figure 5.1.

**Setup Application** The setup application is responsible for the deployment of network topologies on managed infrastructures, e.g. testbeds. It is depicted as the *Setup* box in Figure 5.1 and is developed in [14].

**Management Unit** The management unit controls every operation executed by the INSALATA system. It provides an interface to download network topology plans or to upload such for deployment.

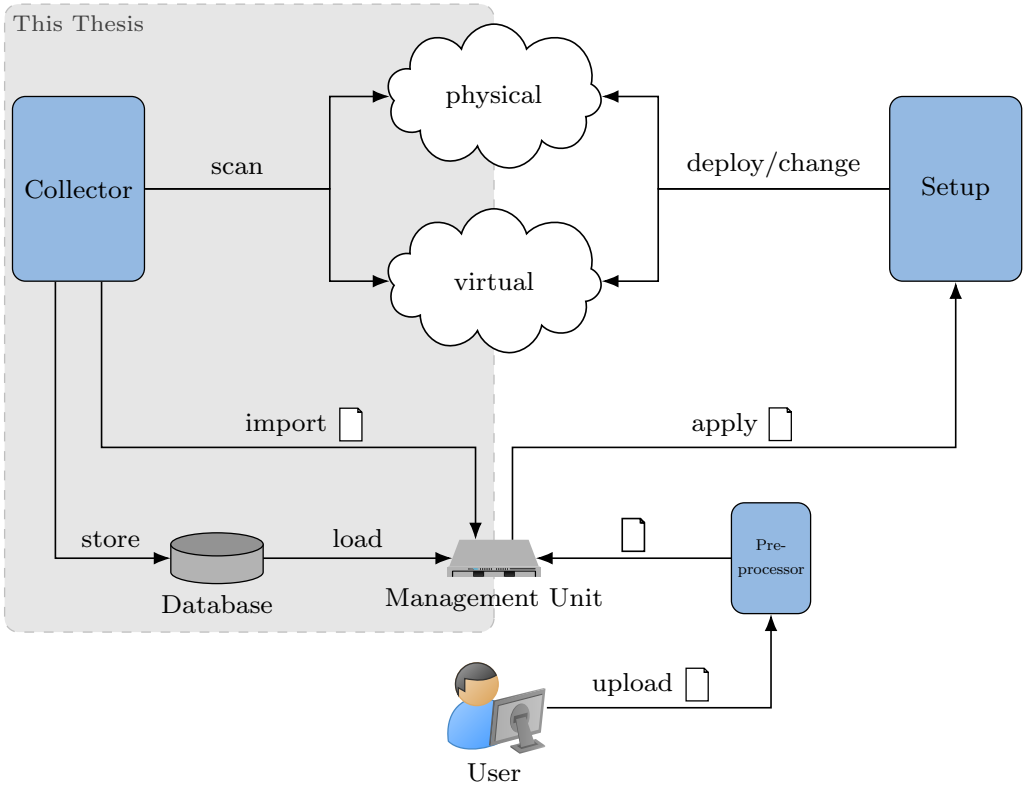



Figure 5.1: Schema of INSALATA

## 5.2 Collector Application

In this section we design the collector application. As shown in Figure 5.2, the management unit utilizes the collector application represented by $Env_1$ and $Env_2$) to gather network topology information.

It is not possible to gather all network topology information in one central tool. Therefore, different collector modules ($Mod_{i,j}$) are used by the collector application to gather the network topology information.

The collector modules gather the network topology information in the network. The information from the different collector modules is merged in the collector application to obtain an overall depiction of the network topology. The user has the opportunity to configure which collector modules shall be used to gather network topology information. This is represented by the two `.ini` files of $Env_1$ and $Env_2$. The design of the collector modules is developed in Section 5.2.1.

INSALATA has to store the network topology information as defined in the information model (Section 3.2.2). The internal data structure is designed in Section 5.2.2.

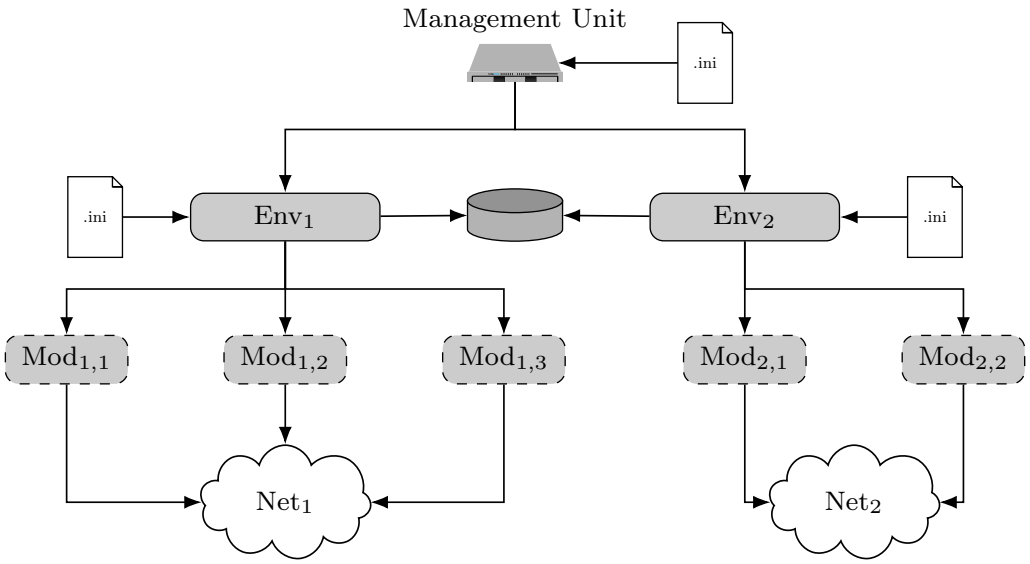



Figure 5.2: Schema of the Network Topology Information Collection

The gathered network topology information can be exported by the collector application. The output mechanism is designed in Section 5.2.3.

To be able to monitor different environments at the same time, the management unit has the ability to create multiple instances of the collector application. The instances have to be executed independently and must store the network topology information separated. Therefore, an `EnvironmentHandler` is used for every environment. Which environments INSALATA has to monitor is configurable using the `.ini` file of the management unit.

### 5.2.1 Collector Modules

An EnvironmentHandler utilizes different collector modules to gather the network topology information. A collector module can use various methods for network topology information collection. It is not possible, to provide a collector module for every network component that can be used in a network. As a consequence, it must be possible to add custom collector modules. Therefore, a generic interface is necessary to invoke the collector modules. We describe the parts of this interface in the following.

**Access to the Graph** The EnvironmentHandler must enable the access to the currently stored network topology information.

**Additional Information** The collector modules need additional information to be able to gather network topology information, e.g. credentials for direct access. To provide the required information generically we use a configuration file for

every used collector module. The EnvironmentHandler loads the configuration and passes the information to the collector modules.

**Logger** We have to report errors on network topology information gathering to the user, but we do not have a direct connection. We use a logger for each environment to handle this task.

**Finished Callback** The EnvironmentHandler has to trigger the collector modules periodically (**SCAN-REQ4**). A callback function is used to inform the EnvironmentHandler about the end of execution. This is required, as the EnvironmentHandler must retrigger the collecting after a configured time interval (**SCAN-REQ5**).

**Synchronization** Different collector modules are executed continuous and do not finish their execution, like a passive network scanner. This means that they do not stop to gather network topology information until INSALATA is shut down. Therefore, the EnvironmentHandler has to execute the collector modules independently and concurrently. As a consequence, we have to handle race conditions. To enable a clean shutdown of INSALATA, a synchronization concept is necessary to inform all continuous collector modules about it.

Summarizing, a collector module must retrieve access to the currently stored network topology information, its configuration, a logger, a `finished` callback function and must have a possibility to synchronize on a system shutdown. This information defines the generic interface all collector modules must have.

### 5.2.2 Storing of Network Topology Information

In this section we design an approach to store the collected network topology information in INSALATA. Therefore, an internal representation of the information in the EnvironmentHandler and a mechanism to merge the information from different collector modules is developed. To define which information is stored in the internal data structure, a data model is used.

#### 5.2.2.1 Data Model

In this subsection we develop the data model which is depicted in Figure 5.3. The data model is used to store the heterogeneous network topology information from the different collector modules in an unified way. The developed data model depicts the basic network topology information and it is extensible (**DM-REQ7**).

The data model consists of entities containing different fields of a type. We refer to every entity in the data model as a *network topology object*. When we introduce a field, we name its type in brackets. There are fields, every entity in the graph must have and

therefore they are stored in the **NetworkTopologyObject** class. For reasons of clarity we do not depict the generalization of the NetworkTopologyObject to every other class in Figure 5.3.

In addition, we introduce the relationships between the network topology objects in this section. On the fist occurrence the name of the relationship is written cursive.

To support the network topology history (**DM-REQ8**) we use the fields *lifetime_start (Time)* and *lifetime_end (Time)*. Every network topology object must have an *identifier (String|Integer)* which is unique in the scope of the data model. The identifiers vary between the network topology objects. Therefore, we do not store them in the NetworkTopologyObject class, but ensure that every subclass implements it. We chain the identifiers of some classes to get unique identifiers, e.g. the name of a disk is only unique in the scope of a network component. The identifiers are emphasized by underlining in Figure 5.3. Some classes must use multiple fields as identifier. If this happens, we do not underline the single parts of the identifier, but name them in the description.

The data model maps active network components (**DM-REQ1**) in the **NetworkComponent** class which has its *name (String)* as identifier. Network components have a type, like host or router, which is stored in the field *template (String)*. Every network topology object has a number of *cpus (Integer)* operating on a certain *cpuSpeed (Integer)*. Memory information is depicted using the *memoryMax (Integer)* and *memoryMin (Integer)* fields. We use upper and lower bounds as many virtualization environments do so. On a physical network component those values are equal. A network component has a *powerState (String)*, like 'Running' or 'Halted', it can adopt.

A network component stores its data on multiple **Disks**. We map these as network topology objects with its *size (Integer)* as a field. The identifier of a Disk is the *name (String)* of the disk on the network component. A network component can *contain* multiple disks. For data sharing, one disk can be plugged into several network components.

We introduce two kinds of linkage (**DM-REQ2**). **Layer2Networks** depict connections between network components on the second layer of the ISO/OSI model, like switches and hubs. Those map passive network components. The identifier is either a generated unique name or the name of the network in the, possibly used, virtualization software and is stored in the *name (String)* field.

**Layer3Networks** depict logical links between network components. As IP is the most widespread layer three protocol, it is used as template for the fields. Therefore, a Layer3Network has the fields *address (String)* and *netmask (String)*. The address is used as identifier in the data model.

One environment can include physical and several virtualized network components and layer two networks. This is mapped in the **Location** class which has an *id (String)*

as identifier. Layer two networks and network components are *in* exactly one location, whereas one location can contain multiple objects.

A network components *has* multiple **Interfaces**. It is not possible that an interface is shared between network components. As we have a one to one association of interfaces and layer two addresses, we use the *mac (String)* as the identifier of an interface. Therefore, the Interface class depicts addressing on the second layer of the ISO/OSI model (**DM-REQ3**). To depict the performance of an Interface, we map the used *mtu (Integer)* and its *rate (Integer)*. An Interface is *connected_to* exactly one layer two network and a layer two network contains several interfaces.

Multiple **Layer3Addresses** can be *configured_on* one interface. A Layer3Address is unique in the data model if we chain its identifier, called *address (String)*, with the one of the associated interface. However, there are layer three addresses that are not configured on any interface, like the destination address in a routing table. Layer3Addresses can be configured using protocols like DHCP. To map this behavior we use the field *static (Boolean)*. A set static flag means that the address is configured directly on the network component. A Layer3Address has a corresponding *netmask (String)* and *gateway (String)*. A layer three address is *connected_to* up to one layer three network and a layer three network contains multiple layer three addresses.

Network services are *running_on* on layer three addresses and are depicted in the **Service** class. A network service is reachable by sending packets to a layer three address using a *ports (Integer)* of a *protocol (String)*. The port is the unique identifier of a network service in the scope of a layer three address. A network service has a *type (String)*, like DHCP or a webservice. If the exact software of a network service is necessary, we can use the fields *product (String)* and *version (String)*.

The two basic network services DNS and DHCP are depicted as the separated classes **DnsService** and **DhcpService**. Those classes specialize the Service class. A DNS service has an extra field called *domain (String)*. The DHCP service contains its *lease (Integer)* time as field. The *start* and *end* of the provided address range is specified by a layer three address. One layer three address can be used by multiple DHCP services. In addition, a DHCP service provides a default gateway which is mapped using the *hop* relationship.

Routing implements the extension of reachability (**DM-REQ4**) and is enabled by routing tables. Those can be generated using protocols or are statically defined on routers. As we want to be able to describe every routing behavior and static routing tables are able to depict the behavior of routing protocols, we decided to map static routing tables in the data model. A router is depicted by a network component *using* multiple **Routes**. The Route class specifies following fields and relationships.

**destination** An incoming packet is matched to this route, if the destination layer three network of the packet is equal to the layer three network specified by the

*destination* relationship. A route has exactly one destination layer three network, but there can be multiple routes that match on the same layer three network.

**genmask** The *genmask (String)* of a route is used to generate the destination layer three network using the address of the received packet.

**metric** If multiple routes match to the received packet the *metric (Integer)* is used to determine the best one.

**hop** If a route matches, the router forwards the packet to the next *hop* which is specified by a layer three address. A layer three address can be specified as next hop in multiple routes.

**interface** It is possible to specify the *interface* the router shall use if a route matches on a forwarded packet. Multiple routes of a router can use the same interface.

No field of a route can be used as identifier. Therefore, it is generated using a hash on all fields.

Reachability limiting network components are depicted by firewalls. Similar to routers, a firewall is a network component defining **FirewallRules**. We oriented the format of a firewall rule on the output format of *FFFUU* [41] which is able to simplify the rules to a standardized format. Therefore, a firewall rule can define following match conditions in the data model.

**chain** The *chain (String)* of a firewall rule specifies when we shall check if the rule matches, like the IN chain for all incoming packets.

**action** The *action (String)* specifies how to deal with the packet, like DROP or ACCEPT, if the rule matches.

**protocol** The rule only matches on packets of a given *protocol (String)*.

**source ports** We are able to specify on which *srcPorts (list<Integer>)* the rule shall match on.

**destination ports** Equally to the source ports, we can specify the *dstPorts (list<Integer>)*.

**destination network** We can specify on which *dstNet* the rule shall match. A rule can only specify one single layer three destination network, but one network can be used by multiple firewall rules.

**source network** Equally to the destination network, we can specify the *srcNet*.

As identifier we use the destination and source network together with the interface and destination port in the scope of a network component.

FFFUU is sometimes not able to represent the rules exactly. This means that the rules can be more or less strict as the original ones. Therefore, we introduce the **FirewallRaw**

class to be able to store the raw firewall rules in the *data (String)* field. As an example, we can store a dump of the iptables rules. As this rules are only useful if we know the exact firewall application, we store its *name (String)* which is the identifier of the firewall raw dump for one network component. A network component can *use* multiple firewall raw dumps, but the dumps can not be shared between network components.

#### 5.2.2.2 Internal Data Structure

The internal data structure is the representation of the data model in INSALATA. One possibility is to store the network topology information in an hierarchical structure, like a tree.

It is not always possible for the collector modules to deliver complete information. This means that they are not able to add complete subtrees to the data structure. For example, a collector module is able to gather information about layer three networks but is not able to bind this network topology information to an interface.

In addition, not every network topology information can be arranged in a strict hierarchical structure. As an example, network components and layer two networks would be at the same level.

Because of this two reasons we can not use a hierarchical structure for the network topology information. As a consequence, we use a generic depiction as a graph. Every network topology object defined in Section 5.2.2.1 is a node in the graph. The fields are mapped as attributes of the nodes and associations as edges between them.

#### 5.2.2.3 Merging of Network Topology Information

The collector modules add network topology information to the graph. The types of information different collector modules deliver can overlap, like two collector modules gathering information about network components. Therefore, the EnvironmentHandler must be able to merge different sources of information.

To demonstrate the used algorithm clearly, we demonstrate it for one type of network topology information. Assume, that three different network component information collector modules provided the following information.

$CollectorA : \{NC_1, NC_2, NC_3\}$

$CollectorB : \{NC_2, NC_3, NC_4\}$

$CollectorC : \{NC_1, NC_2\}$

We are able to identify equal objects using the identifiers in the data model. Therefore, we store following set in the graph $\{NC_1, NC_2, NC_3, NC_4\}$.

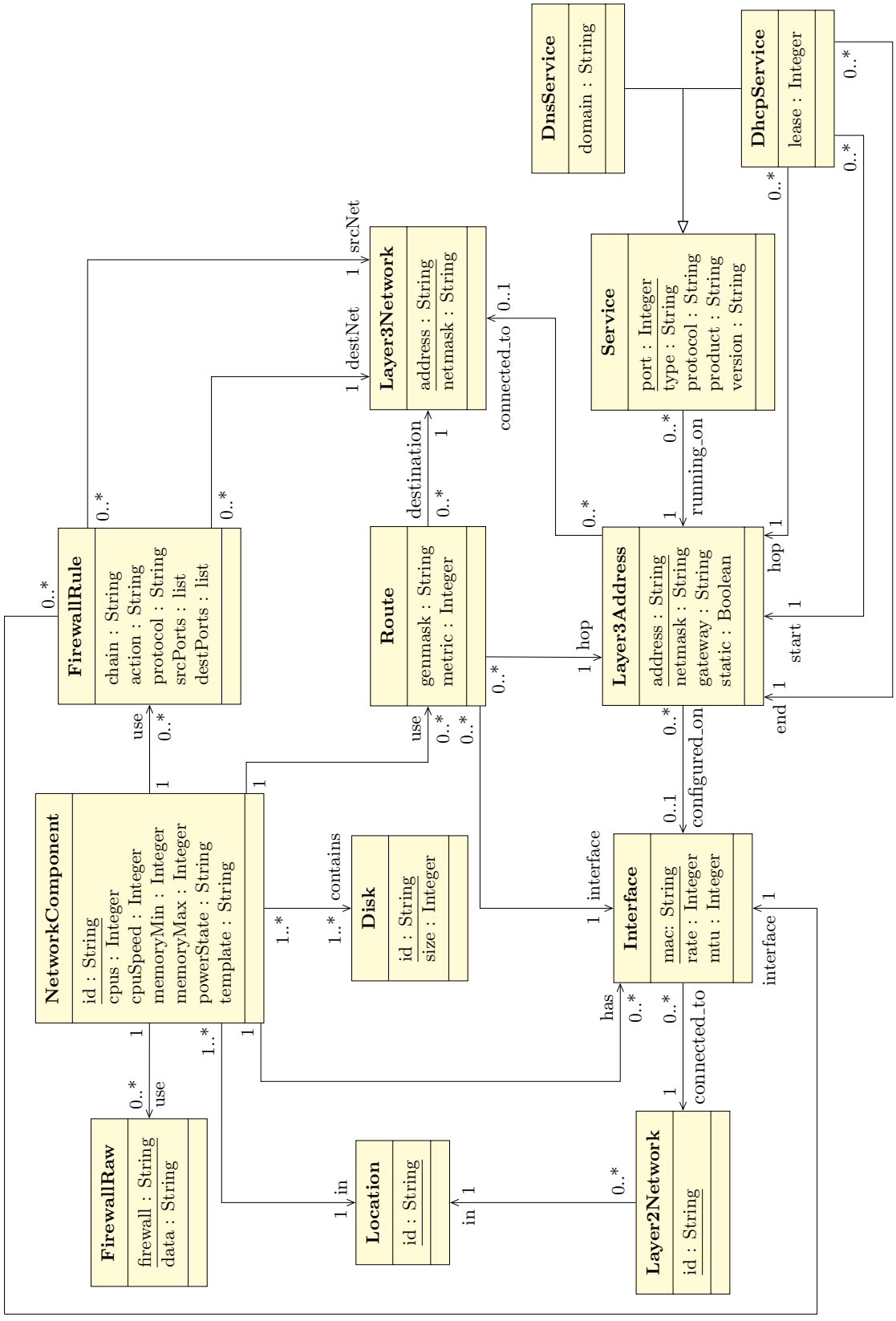



Figure 5.3: Data Model with All Entities and Relationships

As we do not use probabilities for the provided network topology information, we are not able to merge inconsistent network topology information in the EnvironmentHandler. This means, that we are not able to evolve the correct network topology information if two collector modules provide different values for a field. Therefore, we assume that the collector modules provide correct but not the entire information.

#### 5.2.2.4 Verification of the Network Topology Information

Assume, that collector module $CM_1$ provided the information that the network topology object $NTO_1$ is present in the network. This is called a verification of CM1 for $\mathrm{NTO}_1$.

The collector modules can delete such verifications. If a network topology object in the graph is not verified by any collector module, we have to delete it.

An issue of such verifications is that some collector modules are not able to delete them, like passive network scanners. Therefore, we store the verification together with a timeout. This means that a collector module only verifies the existence of a network topology object for the duration of the timeout. If the timeout elapses without a reverification of the same collector module, the verification is deleted automatically. The user can configure the used timeout of a collector module in its configuration file.

### 5.2.3 Output of Information

The EnvironmentHandler stores all retrieved network topology information in the graph. We have to be able to export this information in different and extensible data formats (**SCAN-REQ8**). Therefore, we use independent export modules. Figure 5.4 shows a snapshot of the EnvironmentHandler and the corresponding graph. Every circle on the left side of Figure 5.4 represents a network topology object in the graph. The different export modules are depicted as $\mathrm{Out}_\mathrm{i}$.

INSALATA supports triggered export modules. Those export modules are invoked by the EnvironmentHandler after a specified time interval and export all network topology information at once. The triggering interval of each export module can be configured in the environment configuration file. Triggered export modules can use the *lifetime_start* and *lifetime_end* fields of the network topology objects to store the network topology history. They access the required information from the graph of the EnvironmentHandler.

INSALATA also supports continuous export modules that export every change in the graph. This means that triggered export modules are invoked every time we add, delete or change something to, from or in the graph. As it is impossible to bind every export module to every network topology object, the graph publishes all changes at a central place by providing following three events.

- ADD - A new entity is added to the graph.
- SET - A field of a network topology object in the graph was set to a new value.
- DELETE - A network topology object in the graph was deleted.

The continuous export modules can subscribe to this events and are therefore triggered every time a change is made in the graph. Continuous export modules do not need access to the network topology objects as they are able to track all changes of the graph. Therefore, they know the state of the graph at every point of time.

The user has the ability to configure which continuous and triggered export modules shall be used in the configuration file of the environment.

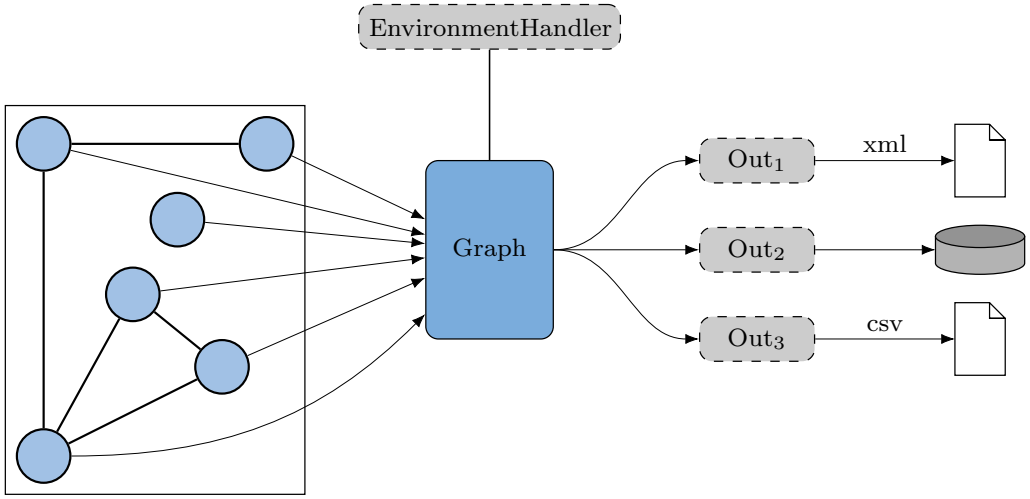



Figure 5.4: Tracking of Changes in the Data Structure

# Chapter 6

# Implementation

In this chapter we present important aspects of the implementation of the collector application and the data model. We give an overview of the tools we used in the implementation. In addition, we introduce the configuration possibilities, the implemented collector modules and the implementation of the data model and the export mechanism.

The implementation and code documentation is available on the INSALATA GitHub repository [42].

## 6.1 Programming Language

We want to monitor different networks with varying device types and operating systems. As it is not always possible to include an additional device in the network, it must be possible to execute the management unit on different platforms. Thus, we decided to use an platform independent programming language for INSALATA.

To be able to use the extensible design of the collector application, we wanted to use an interpreted programing language.

Python is a programming language that fulfills both requirements. In addition, Python is very widespread which eases the extension of the collector application, because many programmers know it. Therefore, we use *Python 3.5.0+* [43] for the implementation of INSALATA.

## 6.2 Tools

In this section we present the tools and frameworks we use int he implementation of the collector modules.

***Xen Management API (Rev. 2.3)*** *[44]*—The testbed of the Chair of Network Architectures and Services uses *Xen* [45] as hypervisor. Therefore, many collector modules implemented in this thesis are Xen-specific. We use Python's build-in library **xmlrpclib** to utilize Xen's Management API using Extensible Markup Language Remote Procedure Call (XML-RPC).

***Ansible (2.1.2.0)*** *[46]*—The Ansible framework is able to configure environments with heterogeneous machines. We use the *setup* command to request device information, such as IP addresses and netmasks. Ansible gathers the information independently from the used operating system of the target.

***FFFUU (Commit: 48052df168ee50eadaccfc1f9b4332bacf2e137b)*** *[41]*—This tool analyzes firewall rule sets of an *iptables* firewall and is able to simplify them. We use this to simply the firewalls rules before storing them in the internal data structure. The tool needs an *iptables-save* dump of the firewall rules.

***Zabbix (3.0.5)*** *[47]*—Zabbix is a network monitoring tool. The Zabbix agent is installed on every network component we want to monitor. As Zabbix is used in many infrastructures and much information is stored in the Zabbix Server without further installation or configuration, it is a useful source of information for collector modules. We utilize the **py-zabbix (0.7.4)** [48] library to request information from Zabbix server using the offered API.

***Nmap (6.47)*** *[3]*—Nmap (Section 2.1) is an active port scanner and is able to detect the applications listening on open ports and their versions. It is used by the `NmapService` collector module.

***Paramiko (2.0.2)*** *[49]*—This library implements SSHv2 functionality in Python. We use Paramiko to establish SSH connections to network components. It is utilized in all collector modules using direct access for information collection.

***PySNMP (4.3.2)*** *[50]*—SNMP is a protocol designed to gather information about network components. PySNMP is a Python library implementing version one to three of the SNMP protocol. We use it in the `insalata.helper.SNMPWrapper` to request information from network components providing SNMP.

***lxml (3.4.4)*** *[51]*—Lxml is a Python library to handle XML files. It is used to import and export network topology information in XML files.

***configobj (5.0.6)*** *[52]*—The Python build-in library for configuration files is not able to handle subsection. Therefore, we use configobj to read and write configuration files in the management unit and the EnvironmentHandler.

***netaddr (0.7.18)*** *[53]*—Netaddr is a small Python library which allows to handle IP addresses easily. This means that we are able to convert string formatted IP addresses into a format which allows calculations. Therefore, it is used to calculate for example the address of the network using the netmask and an IP address.

## 6.3 Configuration of the Collector Application and the Collector Modules

It must be possible to configure which environments the collector application shall monitor (Section 5.2). The monitored environments and other global settings, like the port and address the management unit listens on for commands, are configured in a global configuration file. The global configuration is stored in `/etc/insalata/insalata.conf`. The path of this file is fixed.

Listing 6.1 shows a sample configuration of INSALATA. Lines 1-3 define values about the logging of the system. The management unit uses a rotating file logger with a configurable log file size and number of files. To be able to send commands to the management unit, we developed a Python client. Line 4 and 5 define on which address and port we are able to communicate with the management unit. The `plannerPath` is a configuration value of the Setup component. The `environments` section contains all environments the system shall handle. In this example we handle one environment called *Environment1* and all of its data is stored in the directory specified by `path`.

```
1  logLevel = info            # Define the verbosity of the log messages
2  logSize = 104867600        # Define the maximum size of the log files
3  backup = 2                 # Number of backup files of the rotating file logger
4  rpcServerAddress = 0.0.0.0 # Address the management unit listens on for commands
5  rpcServerPort = 5340       # Port the management unit listens on
6  plannerPath = planner.conf # Path to the planner used by the setup application
7
8  [environments]             # This section contains all environments we shall monitor
9     [[Environment1]]
10       path = /etc/insalata/environment1/  # Path to the directory of the environment
```

Listing 6.1: Sample Configuration of INSALATA

Listing 6.2 shows an example configuration file for one environment. In Line 1 we specified where the system shall store the exported network topology information. We are able to configure in which export formats we want to output the network topology information (Line 2 and Lines 10-12). Each subsection of the `modules` section defines

a collector module that shall be used in the environment. The EnvironmentHandler triggers the `scan` method of the `insalata.scanner.modules.<type>` collector module in the configured time interval (Line 8). The EnvironmentHandler loads the additional information for Collector1 from the file specified by the `config` parameter (Line 7). Listing 6.3 shows an example of such a file.

```
1  dataDirectory = data/            # Path to the data directory of this environment
2  continuousExporters = JsonOutput # Comma separated list of continuous export modules
3
4  [modules]                        # Collector modules used for information gathering
5     [[Collector1]]                # User specified name of the collector module
6        type = XenHostScan         # Type of the collector module
7        config = scannerConf/host.conf # Configuration file for this module
8        interval = 15              # Triggering interval for this module
9
10 [triggeredExporters]             # Triggered export modules used in this environment
11    [[XmlExporter]]               # Type of the export module
12       interval = 600             # Triggering interval for this export module
```

Listing 6.2: Sample Environment Configuration

Listing 6.3 shows the required additional information for the `XenHostScan` collector module. The login credentials and address of the used Xen server as specified in Lines 1-3. The verification timeout is set by the `timeout` parameter in Line 4.

```
1 xenuri = yggdrail    # Address/DNS name of the Xen server
2 xenuser = *****      # Username for the login at the Xen Server
3 xenpw = *****        # Password for the login
4 timeout = 60         # User 60 seconds for the verification timeout
```

Listing 6.3: Additional Information Provided to the XenHostScan Collector Module

## 6.4 Interface of the Collector Modules

In Section 5.2.1 we defined that we need a generic interface for the collector modules. If the EnvironmentHandler has to trigger a collector module, the `scan` function is invoked which has the following interface.

```
scan(insalata.model.Graph, dict, Logger, Thread)
```

The EnvironmentHandler passes the currently stored network topology information to the collector modules using a `insalata.model.Graph` object. This class is discussed in Section 6.6.2.

Section 5.2.1 also defines that every collector module has specific login credentials which are passed by the `dict` parameter. Which login information is necessary for a collector module is documented in [42].

The `Logger` parameter allows the collector module to report error messages.

To be able to execute the collector modules independently and concurrently, the invocation of the collector modules is executed in `insalata.scanner.Worker` threads. For synchronization, the collector module is able to access this thread using the `Thread` parameter. The `finished` callback function is invoked by the Worker Thread. Therefore, we do not have to pass it to the collector module.

## 6.5 Implemented Collector Modules

In Section 3.3 we evaluated different information collection methods. To proof that every of these methods is useful to gather network topology information, we implemented at least one of every kind. As we used a Xen-based environment for the development, many collector modules are Xen-specific. Those modules are described in Subsection 6.5.1.

### 6.5.1 Xen-Specific Collector Modules

In this subsection we present the provided collector modules utilizing a connection to a Xen-based environment.

***XenHostScan***—The XenHostScan gathers information about all network components in the Xen environment specified in its configuration file. It also collects information about the power state and the used template of the network component. In addition, the Xen server is added to the data structure as a Location object.

***XenHardwareScan***—The collector module gathers the number of CPUs from the Xen environment and updates the information in the graph.

We read the information which Virtual Disk Images (VDIs), used as persistent storage in Xen, are attached to a network component. The gathered VDIs are added as Disk objects to the graph and associated to the correct network component.

***XenNetworkScan***—This collector module requests information about all layer two networks on the Xen server and adds the gathered information to the graph.

***XenInterfaceScan***—Xen explicitly connects interfaces to network components and layer two networks. Therefore, we can associate the created Interface objects to the corresponding network components and layer two networks. In addition, we gather the performance data of the interface from the Xen server.

### 6.5.2 Manual Information Collection

To import manually collected network topology information, we implemented the **XMLScanner** collector module. This collector module is able to parse network topology information provided in an XML file and adds the information to the graph. Appendix A defines the schema of the XML.

The provided XML file is scanned periodically and the network topology information is updated without a restart of INSALATA.

### 6.5.3 Collector Modules Using Passive Network Scanning

To have access to many packets, the **TcpdumpHostDetection** collector module is able to connect to a network component via SSH. The collector module starts packet monitoring on the target machine using *tcpdump* and adds every detected layer three address as a network component to the graph. This is necessary as we are not able to determine if a network component uses multiple layer three addresses.

However, if we detect an address which is contained as Layer3Address object in the graph, we do not add a separated network component, but verify the address.

### 6.5.4 Collector Modules Using Active Network Scanning

*Nmap* has the possibility to detect running services and their version in a network. To access as many networks as possible, the **NmapServiceScan** collector module connects to a remote network component via SSH and starts *Nmap*. The collector module adds the gathered information to the graph.

The user has the possibility to define address ranges the collector module shall not scan in its configuration file. This is necessary, as some networks the scanning network component has access to shall be omitted, e.g. the Internet.

### 6.5.5 Protocol-based Collector Modules

We implemented two collector modules utilizing SNMP for network topology information gathering.

***SnmpRoutingCollector***—The SnmpRoutingCollector collector module gathers routing information from network components running Linux. The used OIDs are documented in [42].

***SnmpInterfaceCollector***—This collector module requests information about interfaces from all network components in the graph. This collector module uses the standard MIB *MIB-2* to get the MAC address of every interface of network component.

### 6.5.6 Collector Modules Using Direct Access

We implemented several collector modules for network topology information collection using direct access to the device. The modules open a SSH connection to the devices and run scripts, called *host scripts*. As Ubuntu is used in the testbed of the Chair of Network Architectures and Services, those host scripts are designed for Debian-based Linux distributions. The scripts collect the requested information, create a JSON representation and send it back.

The functionality of connecting via SSH and executing a host script is wrapped in the `insalata.helper.SSHWrapper` class.

***SSHKeyScriptInterfaceConfigurationScan***—This collector module gathers network topology information about the layer three addresses of the network components. The `read_interfaceConfiguration` host script reads the `/etc/network/interfaces` file or all files in the `/etc/network/interfaces.d` directory and gathers the name of the interface on the network component. If the interface is configured statically, the configured netmasks and addresses are collected using Ansible.

In addition, the collector module associates the Layer3Address objects to the corresponding interfaces and Layer3Networks. Layer3Networks can be created using the addresses and their netmask.

At the moment, only IPv4 is supported by this collector module as we use IPv4 in the testbed of the Chair of Network Architectures and Services.

***SSHKeyScriptDHCPScan***—Dnsmasq is often used to provide DNS and DHCP services. This module reads the configuration files in `/etc/dnsmasq` and `/etc/dnsmasq/dnsmasq.d` and collects the information on which interfaces a DHCP service is provided and its announced address range. We associate the DhcpService object with all Layer3Address objects on the providing interface.

The host script is able to parse the `dhcp-range` and the `dhcp-option` flags in Dnsmasq's configuration file.

***SSHKeyDnsmasqScriptScan***—The collector module gathers information about running DNS servers in the environment which are using Dnsmasq. The `read_DNSServer` host script reads the domain from the Dnsmasq configuration file and sends it back

to the collector module. As Dnsmasq only supports one domain simultaneously, we do not need a mapping from the domain to the interfaces. The created DnsService is associated to every Layer3Address object configured on the interfaces mentioned in the `interface` flag of Dnsmasq's configuration file.

***SSHKeyScriptRoutingScan***—The Linux kernel has the possibility to route IP packets. Therefore, we provide a collector module which gathers the routing tables from network components running Linux. The host script `read_Routing` classifies the network component as a router if the `net.ipv4.ip_forward` kernel parameter is set. The routes, the network component uses, are sent back to the collector module if the network component was classified as a router.

### 6.5.7 Agent-base Collector Modules

The **ZabbixFirewallDump** collector module is able to collect firewall information utilizing an *iptables-save* dump. The module gathers the type, e.g. *iptables*, and the rule set dump of the firewall by requesting a *Zabbix* server monitoring the network. The collector module adds the raw dump to the network component object.

Based on a simplification, executed using *FFFUU*, single firewall rules are added to the graph.

## 6.6 Implementation of the Graph

In this section we describe the implementation of the graph designed in Section 5.2.2. We show up small deviations from the specified data model (Section 5.2.2.1). Afterwards, we present how the graph and the network topology objects are implemented in INSALATA.

### 6.6.1 Deviations from the Specified Data Model

Details of the implementation of the data model deviate from their specification in Section 5.2.2.1. The reason for this is that the setup application has to store additional information for some steps during the deployment. Therefore, the template is mapped as an separated class in the implementation and has an association to the NetworkComponent class. In addition, the NetworkComponent class is called Host in the internal data model. The NetworkTopologyObject class is called Node.

Despite all changes, the implemented data structure stores the network topology information compliant to the data model.

### 6.6.2 Graph

The EnvironmentHandler has to store all gathered network topology information. Therefore, the `insalata.model.Graph` class is used.

Section 5.2.3 defines that every change of the graph is reported using events. As the Python's events implementation does not support the features we need, we use the `insalata.model.Event` class. Event handlers are registered and unregistered using the `add` and `remove` methods. The event handlers must have the following interface.

```
name(sender:object, args:dict)
```

The `sender` parameter contains the object which triggered the event. The sender can add information about the event trigger using the `args` dictionary.

To publish changes, the graph defines following events.

**`ObjectNewEvent`** The Graph class provides `getOrCreate` methods for every entity in the data model. This methods add new network topology objects to the graph or returns the requested one depending on the identifier.

If a new network topology object is added to the graph, the ObjectNewEvent is triggered. The type of the network topology object and the initial values are provided in the arguments.

**`ObjectDeletedEvent`** A network topology object is deleted from the graph. We pass the identifier of the network topology object and its type in the arguments. This event delegates the `OnChange` event of the network topology object (Section 6.6.3).

**`ObjectChangedEvent`** A field of some network topology object in the graph changed. The graph class adds the type and the identifier of the changed network topology object to the arguments. Other values are passed by the network topology object itself as this event is only delegated (Section 6.6.3).

Changes and deletions of network topology objects are recognized by registering to the events of the single network topology objects described in Section 6.6.3.

Triggered export modules and collector modules need access to all network topology objects. Therefore, the `getAllNeighbors(type)` method is provided. By using this method, we are able to obtain all entities of a given type in the graph.

### 6.6.3 Nodes of the Graph

The `insalata.model.Node` class depicts the NetworkTopologyObject class of the data model and is used to build the graph structure. We use the `insalata.model.Edge` class to associate two Nodes.

Every class representing a network topology object of the data model inherits from the Node class. Therefore, we are able to represent the graph as designed in Section 5.2.2.

The Node class defines following two events used by the graph to publish changes and deletions.

**`OnDeleteEvent`** This event is triggered if an object is deleted. As we pass the sender of the event to the event handlers, no values are passed in the arguments.

**`OnChangeEvent`** The event is triggered if some field of the object changed. Following values are passed in the event's arguments.

- **objectType:** Type of the object we changed/added.
- **type:** Type of the change.
  - set: A member is set to a new value.
  - add: A new edge or value in a list is added.
  - delete: A value of a list or an edge is deleted.
- **member:** (Optional) Affected member. Not necessary if a plain edge is added/removed between two objects.
- **value:** New value of the field or an identifier if an association was added/deleted.

In Section 5.2.2.3 we introduced the verification concept. Therefore, we store the name of all collector modules verifying the network topology object in the Node class. Verifications are added and updated by the function `verify`.

The `removeVerification` function removes the verification of a collector module. If the last verification is deleted, the function calls the `delete` method of the object. The delete function removes all edges the Node is incident to. This ensures that the Node is deleted from the internal data structure.

According to Section 5.2.2.3, we have to ensure that a verification is deleted after a given timeout. Therefore, we store a timer for each verification. If the timer expires without a reverification of the collector module, the `removeVerification` method is invoked automatically.

## 6.7 Output of Information

In this section we describe the implementation of the output mechanism designed in Section 5.2.3. Triggered export modules are triggered by the EnvironmentHandler and retrieve all necessary information by accessing the network topology objects stored in

the graph. INSALATA provides an XML output format which is created by an triggered export module (Section 5.2.3).

Continuous export modules register on the events of the Graph class defined in Section 6.6.2. Therefore, they are able to export every change in the graph. To track network topology variations, we implemented the `JSONChangeLog` export module (Section 6.7.2).

### 6.7.1 XML

We implemented the possibility to export the information stored in the graph to an XML file. The format of this XML is defined by the schema in Appendix A. Listing 6.4 shows a small example of such an XML. According to Section 6.6.1, all network components are called hosts in the XML.

The XML file contains one Layer2Network (Line 4) called *network-1* which is located in the Location *yggdrasil*. The XML defines one network component called *host-1* (Lines 8-22) with one interface and the template *host-base* (Line 10). It is located in the Location *yggdrasil* (Line 10) and the disk called *host-1-hdd* (Line 20) is the only one this network component has access to. The interface (Lines 12,13) has the MAC address *DE:AD:64:7C:64:85* and is connected to the Layer2Network called *network-1*. This linkage is depicted by the *network* attribute of interface which references *network-1* by its identifier.

```
1  <?xml version="1.0"?>
2  <config name="case1">
3     <layer2networks>
4        <layer2network id="network-1" location="yggdrasil"/>
5     </layer2networks>
6     <layer3networks/>
7     <hosts>
8        <host cpus="2" id="host-1" memoryMin="536870912"
9                 memoryMax="2147483648" powerState="Running"
10                location="yggdrasil" template="host-base">
11          <interfaces>
12             <interface mac="DE:AD:64:7C:64:85" network="network-1"
13                        rate="125000" mtu="1500">
14                <layer3address/>
15             </interface>
16          </interfaces>
17          <routes/>
18          <firewall/>
19          <disks>
20             <disk id="host-1-hdd" />
21          </disks>
22       </host>
23    </hosts>
24 </config>
```

Listing 6.4: Example XML Output Generated by the XML Output Module

### 6.7.2 JSON Change Log

The module `insalata.output.JSONOutput` prints all changes in the graph to a text file. Every JSON in the output provides all information given in one event trigger.

In the example in Listing 6.5 one network component is added to the graph called *host-1* and is resident in the location *yggdrasil* (Lines 1-10). According to Section 6.6.1, network components are called hosts. In the JSON the initial values used to create the Host object are printed. In addition, an interface with the MAC address *DE:AD:BE:EF:11:22* is added to the Host object (Lines 11-18). This interface is connected to the Layer2Network called *network-1* (Lines 19-26). In the next step, the interface *DE:AD:BE:EF:11:22* is deleted (Lines 43-48). Therefore, the edge from the interface to the host and to the network has to be deleted beforehand (Lines 27-42).

```
1  {
2     "time":"2016-11-03@19:14:50",
3     "type":"new",
4     "objectType":"Host",
5     "initialValues":
6        {
7           "id":"host-1"
8           "location":"yggdrasil"
9        }
10 }
11 {
12    "time":"2016-11-03@19:14:53",
13    "type":"change_add",
14    "objectType":"Host",
15    "object":"host-1",
16    "member":"interface",
17    "value":"deadbeef1122",
18 }
19 {
20    "time":"2016-11-03@19:14:54",
21    "type":"change_add",
22    "objectType":"Interface",
23    "object":"deadbeef1122",
24    "member":"network",
25    "value":"network-1",
26 }
27 {
28    "time":"2016-11-04@15:00:40",
29    "type":"change_delete",
30    "objectType":"Host",
31    "object":"host-1",
32    "member":"interface",
33    "object":"deadbeef1122"
34 }
35 {
36    "time":"2016-11-04@15:00:40",
37    "type":"change_delete",
38    "objectType":"Interface",
39    "object":"deadbeef1122",
40    "member":"network",
```

```
41     "object":"network-1"
42 }
43 {
44     "time":"2016-11-04@15:00:40",
45     "type":"delete",
46     "objectType":"Interface",
47     "object":"deadbeef1122"
48 }
```

Listing 6.5: Example Change Log Printed by the JSONOutput Module

By parsing this output, it is possible to detect every change in the network. The time stamp in the output enables a reconstruction of the graph at every point of time.

# Chapter 7

# Evaluation

In this chapter we evaluate the collector application and the data model. We verify that we fulfilled every requirement defined in Section 3.4 by the design and the implementation of INSALATA. Afterwards a function test and a case study are performed.

## 7.1 Verification of Requirements

In this section we analyze if the collector module itself and its data model does fulfill the requirements we defined in the Sections 3.4.

### 7.1.1 Data Model

In this section we evaluate if all requirements to the data model are fulfilled. Therefore, we analyze if all necessary information is covered by the the collector application's data model.

***DM-REQ1*** *Network Components*—Active network components are directly mapped in the NetworkComponent class. Passive network components, like switches and hubs, only influence the network topology by setting up links between other network components. Therefore, passive network components are represented by Layer2Networks. Special network components can be depicted using a specialization or by associations to the NetworkComponent class, as routers and firewalls do.

***DM-REQ2*** *Linkage between Network Components*—Physical links between network components are depicted by Layer2Networks. Logical links are represented by the Layer3Network class. Therefore, both considered concepts of linkage are mapped in the data model.

***DM-REQ3*** *Addressing*—We provide different layers of addresses. We use the Interface class and its mac field for the identification on the second ISO/OSI layer. The Layer3Address class is used for addressing in Layer3Networks.

For the addressing on layer four of the ISO/OSI model we map ports in the Service class.

***DM-REQ4*** *Reachability of Network Components*—Reachability extending network components are mapped by a plain network component defining routes. Using this approach, we are able to store static routing tables and routing tables generated by routing protocols compliant to the data model.

The limitation of reachability is depicted by the use of a network component that uses FirewallRaw data or multiple firewall rules. This enables the mapping of dedicated network firewalls and firewalls which are installed directly on a network component.

Therefore, the reachability of network components is mapped for IP-based networks which is the most widespread layer three protocol. Other techniques can be depicted by an extension of the data model.

***DM-REQ5*** *Network Services*—In the data model we provide the generic Service class which allows the mapping of every network service by specifying a type and its port. We decided to use a generic class as there is no possibility to depict every existing network service separately. If the user wants to specify additional data, a new subclass or a field in the Service class has to be introduced.

The two basic network services DHCP and DNS are mapped as a separated class.

***DM-REQ6*** *Hardware Information*—To provide hardware information, like memory and CPU speed, we use fields in the NetworkComponent class. In addition, we introduced the class Disk which enables the mapping of storage information.

***DM-REQ7*** *Extensible Information*—Additional data can be added to the data model easily by defining custom fields or classes with corresponding associations. The implementation of the data model allows extensible information by adding the specified entities to the graph components. Thus, the extensibility of information entities is fulfilled by the data model and its implementation.

***DM-REQ8*** *Enable Network Topology History*—Every entity in the data model inherits from the NetworkTopologyObject. Therefore, all entities contain the attributes

`lifetime_start` and `lifetime_end` which allow the tracking of variations in the network topology. In addition, there is the possibility to detect changes in the graph of INSALATA by using the events defined in Section 5.2.3.

### 7.1.2 Collector Application

In this section we evaluate if the requirements to the collector application are fulfilled by the design and the implementation.

***SCAN-REQ1*** *Configurable Collector*—The user has the ability to define which environments shall be monitored by INSALATA in a configuration file. In addition, he has the possibility to configure which collector modules are used to gather network topology information for each environment.

***SCAN-REQ2*** *Compatibility to the Data Model*—We defined a graph to store the network topology information in Section 5.2.2. Expect for minor changes (Section 6.6.1), the implementation of the graph is compatible to the data model.

***SCAN-REQ3*** *Extensible Collector Modules*—Additional collector modules can be added to the `insalata.scanner.modules` folder. The user is able to utilize the new collector modules without further installation. Therefore, the user is able to extend the set of collector modules easily.

***SCAN-REQ4*** *Periodic Collecting*—The configuration of the environment (Section 6.3) specifies the collecting interval for every single collector module. This interval is used by the EnvironmentHandler to start the collecting periodically.

***SCAN-REQ5*** *Customizable Collecting Interval*—The user can specify the collecting interval for each collector module in the configuration file of the environment.

***SCAN-REQ6*** *Support Continuous Collecting and Information Adaption*—We execute each collector module in a separated thread and the graph is able to insert network topology information asynchronous. Therefore, the collector application is able to handle continuous collector modules.

***SCAN-REQ7*** *Support Multiple Environments*—In the configuration file of system, the user has the possibility to define which environments are monitored. Each environment is handled by a separated EnvironmentHandler which executes in its own thread and contains an isolated graph. Therefore, the environments are independent and INSALATA is able to handle different environments.

***SCAN-REQ8*** *Different and Extensible Output Formats*—INSALATA supports triggered and continuous export modules. Both can access all required information using the graph of the EnvironmentHandler. Therefore, the adoption of additional export modules is simplified.

It is configurable which export modules shall be used for each environment. Thus, the collector application is able to handle different export modules at the same time.

## 7.2 Function Test

In this section we test the function of the collector application. Therefore, we describe the procedure of the function test and the test environment. Afterwards, the results are presented.

### 7.2.1 Goal and Procedure of the Function Test

We want to evaluate the resource consumption of the collector application and the implemented collector modules for a test network. Therefore, we deploy the test network shown in Figure 7.1 on a virtual, Xen-based environment. The test network contains a *DMZ* and a *internal* network. Three hosts, an DHCP server and a database are resident in the internal network. The DMZ contains a webserver and an DNS server. The internal network and the DMZ are connected by the internal router. All network components can reach the Internet using the external router.

We take a look on the network traffic generation and time consumption of each implemented collector module. We measure the incoming and outgoing network traffic on the management unit from the point of time where we invoked the collector module, using *bwm-ng* [54]. The measurement is stopped until all network topology information, the considered collector module is responsible for, is gathered. This measurements are performed multiple times for each collector module. We depict the results of only one measurement, as all measurements delivered similar results. In addition, it is not possible to calculate the average of all measurements as this would falsify the results.

The collector application is designed to gather network topology information about the whole virtual environment and there are some network components already deployed

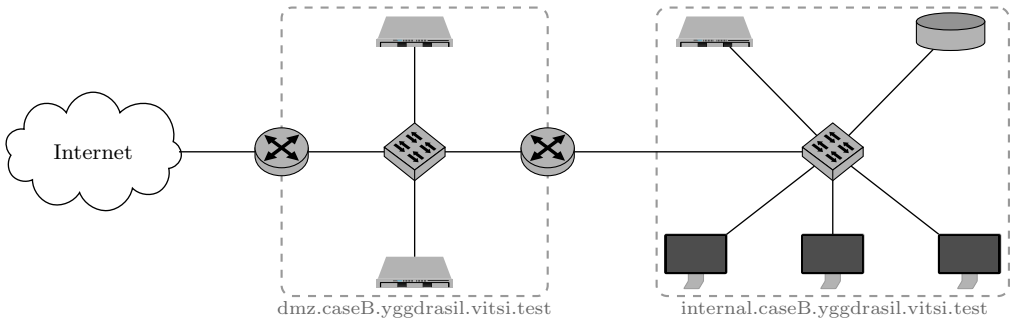



Figure 7.1: Typical Network Infrastructure for the Function Test

on the test environment. Therefore, we measure the resource consumption before the deployment of the test network and afterwards. The difference between the two measurements gives an assumption of the resource consumption for the test network.

At the end, we combine all collector modules to gather the whole network topology information.

### 7.2.2 Test Environment

We execute the function test in the Xen environment of the Chair of Network Architectures and Services at the TUM. The Xen server we use for the function test has the following specifications.

**Operating System** Debian 7.6

**CPU** 24x Intel(R) Xeon(R) CPU E5-2620 v2 2.10 Ghz

**Memory** 128 GB

The management unit is a virtual network component on the Xen server which has the following specifications.

**Operating System** Ubuntu 15.10

**CPU** 4x Intel(R) Xeon(R) CPU E5-2620 v2 2.10 Ghz

**Memory** 8 GB

### 7.2.3 Results

In this section we list the measurement results for the different collector modules and the overall collection. It is important to mention that the scaling of the y-axis of the plots is different.

#### 7.2.3.1 Xen Specific

The behavior of the resource consumption is similar for all Xen-specific collector modules. Figure 7.2 shows the measurement results.

We recognize that the collection produces high amounts of incoming network traffic in a short time interval. In this interval all information from the Xen server is gathered. Except for the *XenHardwareScan* the network traffic generation after this peak is almost zero as no additional information is required. As the *XenHardwareScan* has to gather some data in a second request, we detect traffic for a longer period of time.

The generation of incoming network traffic rises proportional to the number of network components in the Xen environment. The requests we send to the Xen server do not contain any data and therefore the outgoing network traffic stays the same after the setup. The duration of the information collection is almost equal.

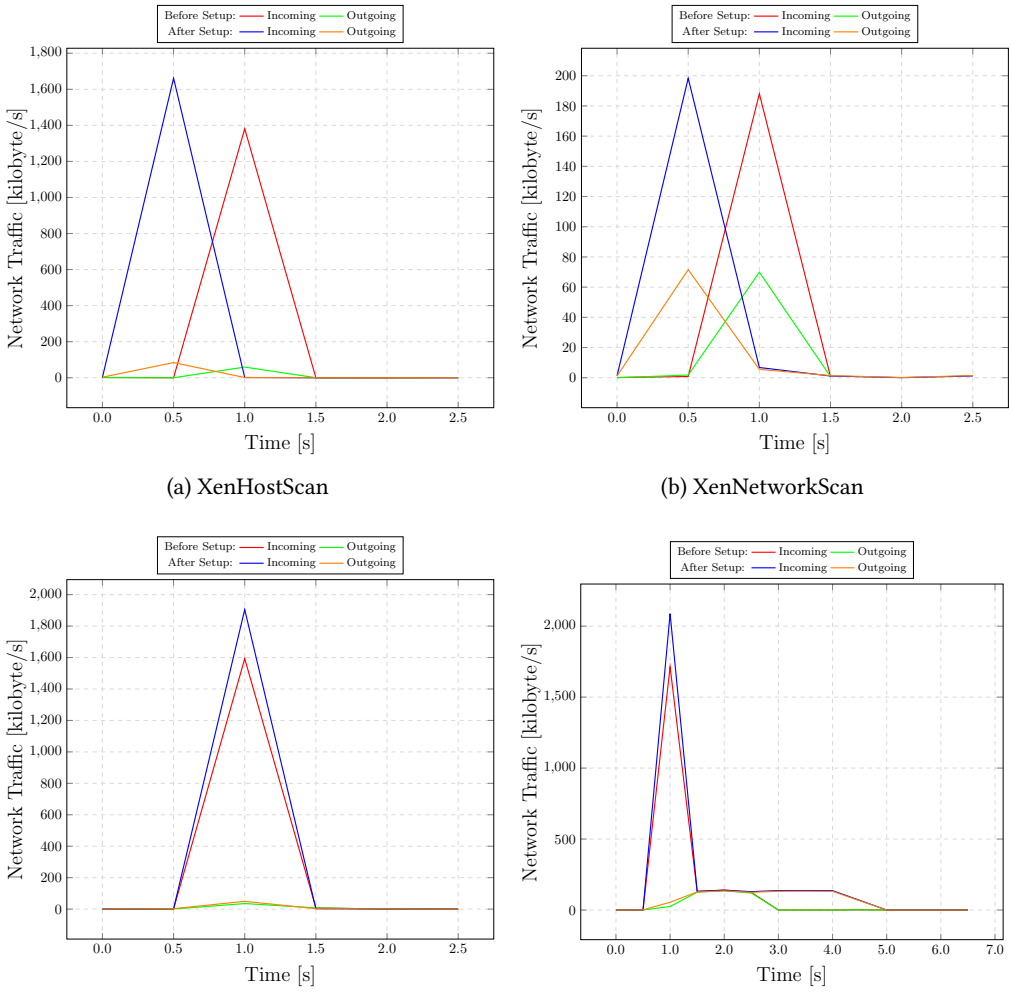
#### 7.2.3.2 Network Scanning

Collector modules using passive network scanning techniques do not produce any network traffic. In addition, the duration of the collection is highly dependent of the network components, the network environment and the packets the collector module has access to. Because of this dependency, it is not meaningful to evaluate the quality of collector modules using passive network scanning in this function test. We evaluate the *TcpdumpHostDetection* collector module in the case study (Section 7.3).

The *NmapServiceScan* executes Nmap on a remote device. Therefore, we do not depict the resource consumption of the management unit. The measurements on the network component executing Nmap are shown in Figure 7.3. This collector module does produce much additional traffic and the duration of the collection is much larger after the setup. The reason for this is that we have to run port scans in more networks which is a resource consuming task.

#### 7.2.3.3 Protocol

Figure 7.4 depicts the measurements of the collector modules using SNMP. We can see that very low additional traffic is generated before the setup. The reason for this is that we do not have any network component in the environment supporting SNMP. Every newly deployed network component supports SNMP. The peaks are the points of time where the data is gathered. As we have to receive less data as in the *SnmpRoutingCollector* collector module, the periods we receive data are shorter.




(a) XenHostScan

(b) XenNetworkScan

(c) XenInterfaceScan

(d) XenHardwareScan

Figure 7.2: Resource Consumption of Xen-based Collector Modules

#### 7.2.3.4 Direct Access

The collector modules using direct access have a similar result as those using protocols. However, the network traffic consumption per connection is higher. The reason for this is overhead produced by the transmission of SSH login credentials.

Figure 7.5 depicts the measurements for the *SSHKeyScriptRoutingScan*, *SSHKeyScriptDHCPScan* and *SSHKeyScriptDNSScan*. In the figure we see the the separated connections to the network components. As we connect to every network component the number of peaks mirrors the number of network components.

The *SSHKeyScriptInterfaceConfgurationScan* is different as the other collector modules using direct access. Using the measurements (Figure 7.5d) we determine that we have

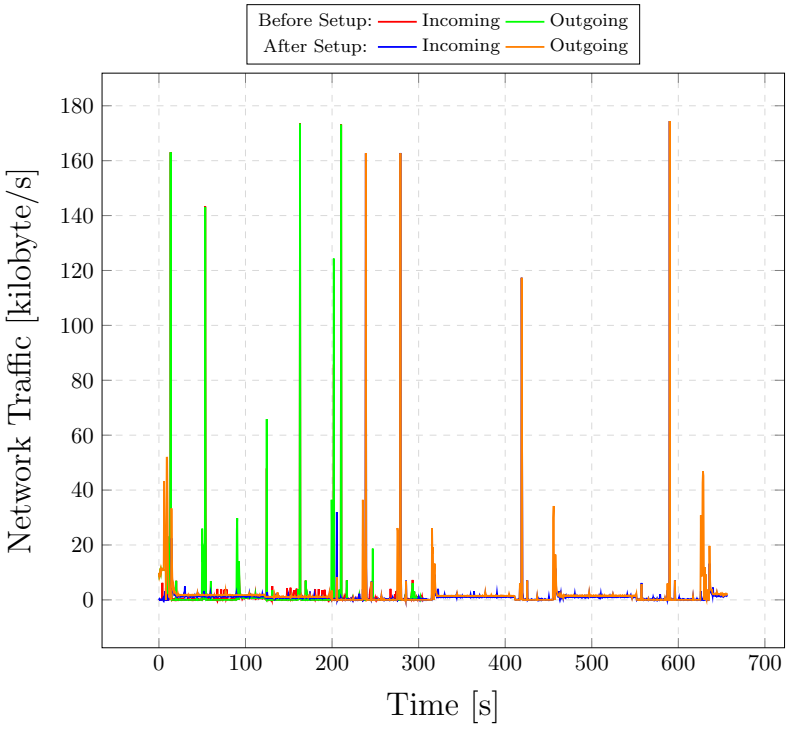
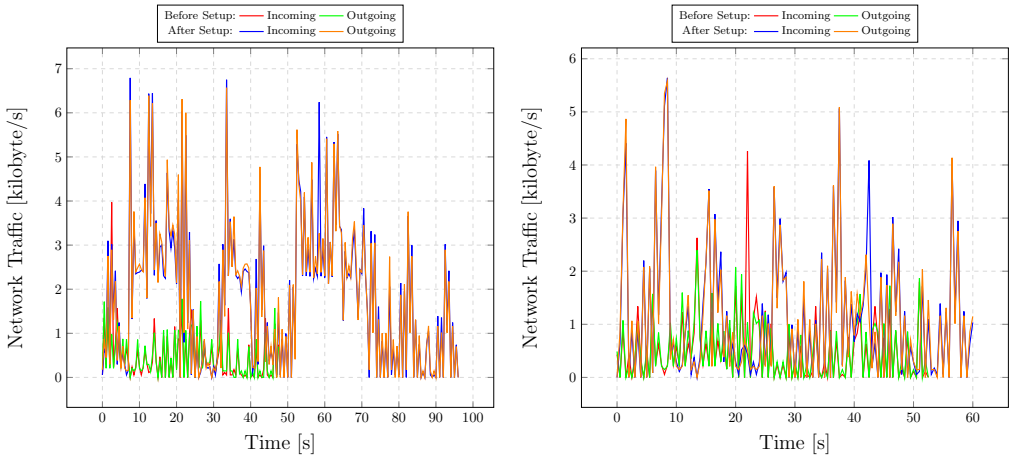
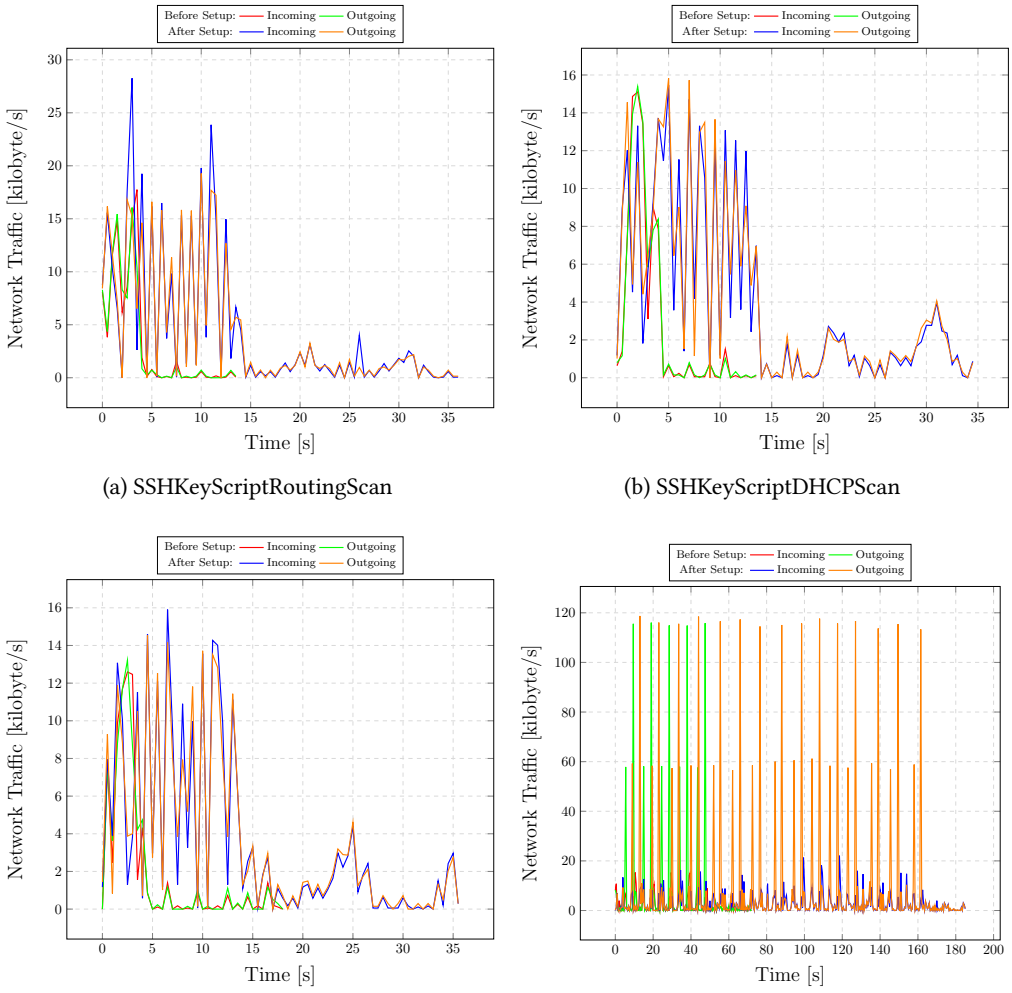



Figure 7.3: NmapServiceScan: Resource Consumption on the Executing Device




(a) SnmpRoutingCollector (b) SnmpInterfaceCollector

Figure 7.4: Resource Consumption of Protocol-based Collector Modules




(a) SSHKeyScriptRoutingScan

(b) SSHKeyScriptDHCPScan

(c) SSHKeyScriptDNSScan

(d) SSHKeyScriptInterfaceConfgurationScan

Figure 7.5: Resource Consumption of Collector Modules Using Direct Access

more connections per network component. The reason for this is that the collector module connects to the network components using SSH and gathers layer three address information from configuration files. In addition, we collect dynamic address information using *Ansible*. Ansible gathers high amounts of data and therefore we get the high peaks in resource consumption. In addition, we determine that this collector module generates more outgoing as incoming network traffic.

After the setup, we have to proceed the collection for more network components. Therefore, the number of connections is higher.

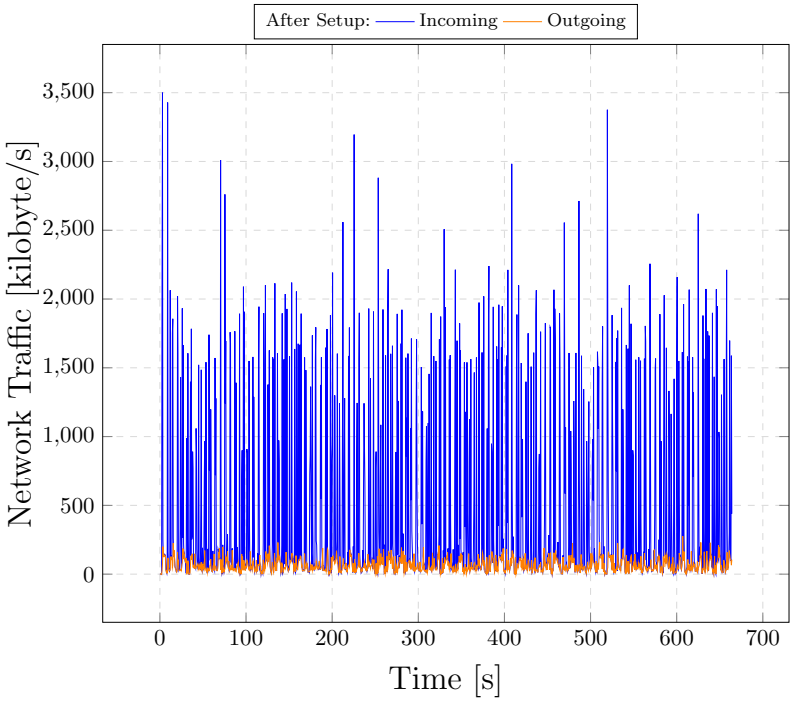



Figure 7.6: Resource Consumption of all Collector Modules Collaborating

#### 7.2.3.5 Agent-based

The last collector module is the *ZabbixFirewallDump* module. In your testbed the Zabbix server is installed on the management unit. Therefore, this collector module does not produce any network traffic during the execution.

#### 7.2.3.6 Collaboration of All Implemented Collector Modules

Figure 7.6 shows the resource consumption of the collector application if we collect network topology information about the whole testbed using all implemented collector modules. We only depict the measurements of a collection after the setup. This measurement shows the resource consumption we have to expect, if we gather network topology information in a virtual testbed with 27 network components and 52 layer two networks.

The configuration for this collection is shown in Appendix B. As we do not want to have time delays based on intervals between the iterations of the collector modules, we set the intervals very low.

The resource consumption of the collector application is highly dependent from the configuration of the environment. The used configuration for the measurement does

produce unnecessary network traffic as we restart every collector module almost right after it finished. Therefore, we can decrease the resource consumption by configuring the environment in a way that we reach the required quality and just-in-time requirements.

## 7.3 iLab Case Study

The Chair of Network Architectures and Services offers a practical course, called *iLab*. In this course students build small network infrastructures to test different network topologies and their behavior. INSALATA could be used in this course to record the results of the students' work and to virtualize parts of the physical devices. Therefore, we mirrored a network infrastructure at the iLab to a virtual environment to estimate the use of INSALATA for the iLab and to take a look at the behavior and quality of the system in a physical environment.

For the case study we use the network infrastructure shown in Figure 7.7 which is an exercise of the iLab. This network infrastructure contains three productive networks which are connected using three routers. Two of the routers are Cisco routers and the third one is a device running Linux which is set up to forward packets. Every network component in this setup is able to access every other network component on every interface. In addition, there is a management network containing every host ($PC_i$) and the *Linux-Router*. As the management network is not important for the network infrastructure and its behavior, we do not depict it in Figure 7.7.

In the following we describe the network topology collection part of the case study and present the results. The redeployment of the network topology in the virtual environment is described in [14].

### 7.3.1 Procedure of the Case Study

We performed four steps in the case study. The label of each network component in Figure 7.7 depicts in which step the network component was detected by the collector modules.

**Input of Static Information (1)** We load static network topology information using the *XMLScanner*.

**Traffic Generation on a Single Host (2)** To measure the time the detection of a single network component lasts, we generated network traffic on *PC3*.

**Traffic Generation on all Network Components (3)** In this step, we generated network traffic on all hosts. Therefore, it is possible to measure the time INSALATA needs to gather all available network topology information.

**Adaption of Static Information (4)** To show the dynamic information adaption during runtime, we added one link to the static XML data.

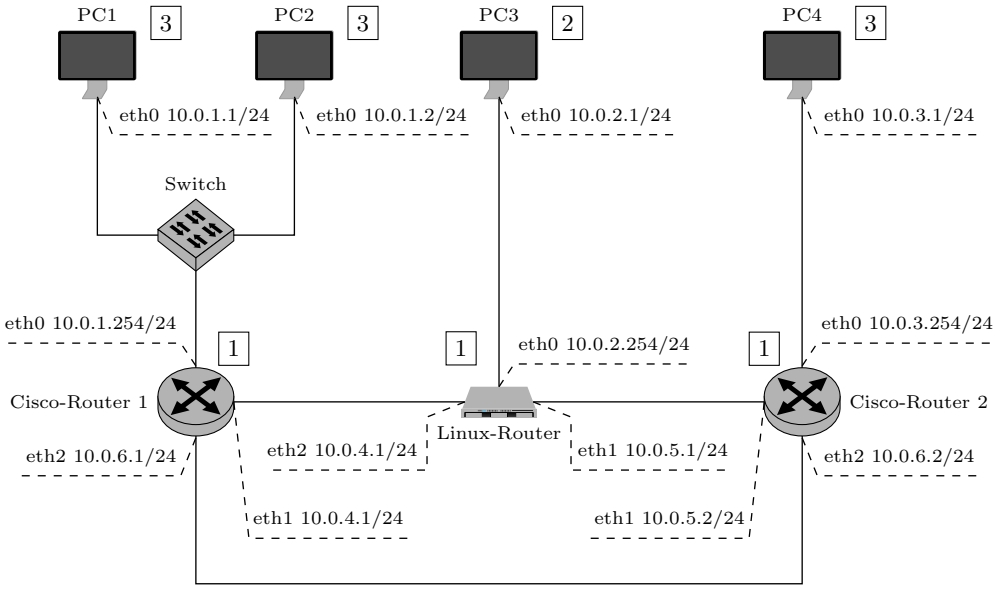



Figure 7.7: iLab Case Study: Target Network Topology

### 7.3.2 Setup for the Case Study

The management unit is connected to the productive network of *Cisco-Router 1.* In addition, it has three monitoring interfaces and retrieves all traffic which is processed by the three routers. The management unit has the following specifications.

**Operating System** Ubuntu 16.04.1 LTS

**CPU** Intel(R) Core(TM) i5 CPU M480 2.67 GHz

**Memory** 8 GB

The environment configuration for this case study is shown in Appendix C.1.

### 7.3.3 Results

In this section we present the results of the case study.

#### 7.3.3.1 Step 1: Input of Static Information

Information about the routers and their interfaces is loaded as static information. Appendix C.2 shows the XML which is provided to the *XMLScanner* collector module.

The static XML file does not contain any information about layer three networks. As the *XMLScanner* adds layer three addresses to the internal graph, we are able to create networks for those addresses. In addition, the *XMLScanner* adds the three routers to the graph. Therefore, we are able to gather interface configuration data from the *Linux-Router*.

Interface and routing information is gathered from the Cisco routers using the *SnmpInterfaceCollector* and the *SnmpRoutingCollector*. The *SSHKeyScriptRoutingScan* gathered routing information from the *Linux-Router*. We can determine that INSALATA is able to adapt the same type of information from different collector modules.

As we did not implement a collector module which is able to detect layer two networks in a physical network environment, we transform every layer three network to a layer two network with the same ID.

Except for the routes, the XML output of the network topology data is the same as the statically loaded one and is therefore not depicted. The routes are contained in the full XML output of this case study (Appendix C.3). The collection of the static network topology information took less than a second. The gathering of routing and interface configuration information 44 seconds.

#### 7.3.3.2 Step 2: Traffic Generation on a Single Host

We generate traffic on *PC3* by sending packets to the *Linux-Router*. As we monitor every packet processes by the routers, the management unit is able to detect *PC3*.

The *TcpdumpHostCollector* detects the network component as soon as the first packet is retrieved on one of the monitoring interfaces. Therefore, the *TcpdumpHostCollector* can be used to monitor a network just-in-time. As *PC3* was added to the graph, the *SSHKeyScriptInterfaceConfigurationScan* collector module is able to gather information about layer three addresses of *PC3*. Therefore, we are able to detect its interfaces and layer three addresses.

The *TcpDumpHostCollector* is not able to detect any network components which do not send any packets and therefore no further information is added to the graph.

As the management unit has a monitoring port on *Linux-Router*, the detection of all available network components took less than a second after the first packet was sent. After the network component detection, other collector modules were able to gather additional network topology information. As there are dependencies between the collector modules, sometimes multiple executions of one collector module are required to gather every available information. Therefore, the gathering of all available information for this step took 127 seconds (171 seconds from the begin of the case study).

#### 7.3.3.3 Step 3: Traffic Generation on all Network Components

In this step, we generated network traffic on every host ($PC_i$) in the network. The *TcpDumpHostCollector* collector module was able to detect every network component.

We do not have to gather network topology information about the routers and PC3 again as we did not restart INSALATA after Step 2.

The *SSHKeyScriptInterfaceConfgurationScan* collector module is used to gather interface and layer three address information from all network components running Linux. In addition, this collector module is able to add layer three networks to the graph by the use of configured addresses and netmasks on the network components.

The gathering of this information took 192 seconds. Therefore, the collection of all network topology information of the network took 363 seconds (~6 minutes). We have to mention, that there are time delays between the steps of the case study which influence the collection time. The resulting network topology information after Step 3 is depicted in Appendix C.3.

#### 7.3.3.4 Step 4: Adaption of Static Information

In the last step we want to show a useful feature of the XmlScanner collector module. We want to add the connection between the Cisco routers as a new connection.

It is possible to add this information during runtime and without the need of restarting the collector application. This is possible as the EnvironmentHandler triggers the XmlScanner periodically as every other collector module. In addition, we have the possibility to delete statically loaded data, by removing it from the XML. This enables the adaptability of manually collected network topology information during runtime.

In this step we update the static network topology information from an XML file and no other collector module has to add information. It takes less than a second to add the information after the XmlScanner is triggered.

#### 7.3.3.5 Summary

Summarizing, we determine that INSALATA is able to collect network topology information in a physical environment. Therefore, there is the possibility to record the results of the students' work in the iLab. However, future work is required to gather all possible network topology information, like the explicit detection of layer two networks. In addition, there are special configuration details we do not map in our data model, e. g. VLANs on a Cisco router. Due to the modular design of INSALATA, it is possible to add such collector modules.

# Chapter 8

# Conclusion

Often mistakes are made while changing the configuration of the network manually. To detect such configuration mistakes, we want to provide an automated mechanism to collect network topology information. Therefore, we developed a general approach to gather network topology information and analyzed which information is necessary to detect network topology variations. As we want to store the information in a standardized way, we examined already existing data models. None of the considered data models was able to map all necessary network topology information. Therefore, we designed a new data model in Section 5.2.2.1.

In addition, we evaluated related work and tools for the automated information collection. No tool was able to fulfill the requirements to an automated network topology information collection tool. Therefore, we provide *INSALATA* and its collector application. The collector application uses the data model in its internal data structure and is designed extensible to collect specific network topology data. In addition, we implemented different collector modules that use different information collection methods.

## 8.1 Research Questions

In this thesis we investigated how network topology variations can be detected. We answered this question by examining following three subordinated research questions.

**Q1: Which information is required to detect network topology variations?**
In Section 3.2.2 we investigated which information is relevant for the network topology. As a result, we determined that we have to depict network components, networks, addresses and services. Addresses and networks have to be considered on different layers of the ISO/OSI model. As logical links are used in networks we also have to map network components that extend or limit the reachability of other network components, like routers and firewalls.

To be able to map network topology information in an uniform way, we developed a data model in Section 5.2.2.1. This data model is designed to depict every relevant network topology information of Section 3.2.2. The user has the possibility to extend the data model to depict special networks.

**Q2: Which possibilities exist for collecting the required information?**
Network topology information is distributed over multiple network components. Therefore, we gather the network topology information by combining different sources of information in the network. This general approach is developed in Section 3.2.2.

In Section 3.3 we concerned with different information collection methods. We considered manual information collection, network scanning, direct access, special protocols and installed agents on the target devices.

We provide the *INSALATA* tool which is defined in Chapters 5 and 6. The Collector part of this application implements different information collection methods in its collector modules. According to Section 3.2.2 the combination of the information delivered by the collector modules is used to generate a depiction of the overall network topology.

Another part of this questions was to evaluate the versatility of the network topology information collection methods. This is important as some of the methods are not applicable in every network. Therefore, we evaluated the intrusiveness of the gathering methods. As a result, we can estimate which information collection methods can be used in a specific network.

In addition, we examined the quality of the gathered information for each method. Therefore, we have the possibility to map the collecting methods to the needs of the user.

**Q3: How can temporal network topology variations be traced?**
We want to provide a possibility to trace variations of the network topology. Therefore, we collect the network topology information continuous and store the different states (Section 3.2.2). By comparing the states, we are able to detect variations in the network topology. This approach is implemented in INSALATA.

INSALATA exports the states of the network topology using triggered export modules. The data model is able to represent the states using the lifetime every network topology object contains.

In addition, INSALATA has the concept of continuous export modules. Those are able to export every state change in the internal graph and therefore in the network. This allows just-in-time tracking of state changes in the graph.

## 8.2 Future Work

We implemented collector modules to gather network topology information in physical networks. However, it is not possible to provide collector modules for all network components. Therefore, the main part of future work is to provide additional collector modules. Those can be added easily due to the modular design of INSALATA.

We implemented a basic set of collector modules that allow the gathering of network topology information virtual environments. However, the collector modules used for virtual environments are Xen-specific and the collector modules that use direct access are designed to operate on Debian-based Linux distributions. Future work is to implement collector modules for other hypervisors and Linux distributions.

We are able to store the changes in the network topology persistently using the export modules. However, we do not provide a tool to import the network topology information to INSALATA again using the change logs.

For the analysis of network topology data, a graphical user interface (GUI) for INSALATA is useful. Currently the user has to download the XML or JSON log file from the management unit and analyze the network topology information manually. A GUI could be used to show the user the gathered network topology data just-in-time. In addition, the monitoring of network topology variations is possible using the change logs.

# Appendix A

# XML Schema for XMLScanner and XML Export

```
<xs:schema attributeFormDefault="unqualified" elementFormDefault="qualified" xmlns:xs="http://
    www.w3.org/2001/XMLSchema">
  <xs:element name="layer2network">
    <xs:complexType>
      <xs:simpleContent>
        <xs:extension base="xs:string">
          <xs:attribute type="xs:string" name="id" use="required"/>
          <xs:attribute type="xs:string" name="location" use="required"/>
        </xs:extension>
      </xs:simpleContent>
    </xs:complexType>
  </xs:element>
  <xs:element name="layer3network">
    <xs:complexType>
      <xs:simpleContent>
        <xs:extension base="xs:string">
          <xs:attribute type="xs:string" name="id" use="required"/>
          <xs:attribute type="xs:string" name="address" use="required"/>
          <xs:attribute type="xs:string" name="netmask" use="required"/>
        </xs:extension>
      </xs:simpleContent>
    </xs:complexType>
  </xs:element>
  <xs:element name="dhcp">
    <xs:complexType>
      <xs:simpleContent>
        <xs:extension base="xs:string">
          <xs:attribute type="xs:string" name="from" use="optional"/>
          <xs:attribute type="xs:string" name="to" use="optional"/>
          <xs:attribute type="xs:string" name="lease" use="optional"/>
          <xs:attribute type="xs:string" name="announcedGateway" use="optional"/>
        </xs:extension>
      </xs:simpleContent>
    </xs:complexType>
  </xs:element>
  <xs:element name="dns">
```

```
    <xs:complexType>
      <xs:simpleContent>
        <xs:extension base="xs:string">
          <xs:attribute type="xs:string" name="domain" use="required"/>
        </xs:extension>
      </xs:simpleContent>
    </xs:complexType>
  </xs:element>
  <xs:element name="services">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="dhcp" minOccurs="0"/>
        <xs:element ref="dns"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <xs:element name="layer3address">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="services"/>
      </xs:sequence>
      <xs:attribute type="xs:string" name="address" use="required"/>
      <xs:attribute type="xs:string" name="network" use="required"/>
    </xs:complexType>
  </xs:element>
  <xs:element name="interface">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="layer3address" minOccurs="0"/>
      </xs:sequence>
      <xs:attribute type="xs:int" name="rate" use="optional"/>
      <xs:attribute type="xs:short" name="mtu" use="optional"/>
      <xs:attribute type="xs:string" name="mac" use="required"/>
      <xs:attribute type="xs:string" name="network" use="optional"/>
    </xs:complexType>
  </xs:element>
  <xs:element name="route">
    <xs:complexType>
      <xs:simpleContent>
        <xs:extension base="xs:string">
          <xs:attribute type="xs:string" name="destination" use="optional"/>
          <xs:attribute type="xs:string" name="gateway" use="optional"/>
          <xs:attribute type="xs:string" name="genmask" use="optional"/>
        </xs:extension>
      </xs:simpleContent>
    </xs:complexType>
  </xs:element>
  <xs:element name="rule">
    <xs:complexType>
      <xs:simpleContent>
        <xs:extension base="xs:string">
          <xs:attribute type="xs:string" name="chain"/>
          <xs:attribute type="xs:string" name="action"/>
          <xs:attribute type="xs:string" name="srcnet"/>
        </xs:extension>
      </xs:simpleContent>
    </xs:complexType>
  </xs:element>
  <xs:element name="firewallRules">
```

```
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="rule"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <xs:element name="raw">
    <xs:complexType>
      <xs:simpleContent>
        <xs:extension base="xs:string">
          <xs:attribute type="xs:string" name="firewall"/>
        </xs:extension>
      </xs:simpleContent>
    </xs:complexType>
  </xs:element>
  <xs:element name="disk">
    <xs:complexType>
      <xs:simpleContent>
        <xs:extension base="xs:string">
          <xs:attribute type="xs:string" name="id" use="required"/>
          <xs:attribute type="xs:int" name="size" use="optional"/>
        </xs:extension>
      </xs:simpleContent>
    </xs:complexType>
  </xs:element>
  <xs:element name="interfaces">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="interface" maxOccurs="unbounded" minOccurs="0"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <xs:element name="routes">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="route" maxOccurs="unbounded" minOccurs="0"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <xs:element name="firewall">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="firewallRules" maxOccurs="unbounded" minOccurs="0"/>
        <xs:element ref="raw" minOccurs="0"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <xs:element name="disks">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="disk"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <xs:element name="host">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="interfaces"/>
        <xs:element ref="routes" minOccurs="0"/>
```

```
154         <xs:element ref="firewall" minOccurs="0"/>
155         <xs:element ref="disks"/>
156       </xs:sequence>
157       <xs:attribute type="xs:byte" name="cpus" use="optional"/>
158       <xs:attribute type="xs:string" name="id" use="optional"/>
159       <xs:attribute type="xs:long" name="memoryMax" use="optional"/>
160       <xs:attribute type="xs:int" name="memoryMin" use="optional"/>
161       <xs:attribute type="xs:string" name="powerState" use="optional"/>
162       <xs:attribute type="xs:string" name="location" use="optional"/>
163       <xs:attribute type="xs:string" name="template" use="optional"/>
164     </xs:complexType>
165   </xs:element>
166   <xs:element name="layer2networks">
167     <xs:complexType>
168       <xs:sequence>
169         <xs:element ref="layer2network" maxOccurs="unbounded" minOccurs="0"/>
170       </xs:sequence>
171     </xs:complexType>
172   </xs:element>
173   <xs:element name="layer3networks">
174     <xs:complexType>
175       <xs:sequence>
176         <xs:element ref="layer3network" maxOccurs="unbounded" minOccurs="0"/>
177       </xs:sequence>
178     </xs:complexType>
179   </xs:element>
180   <xs:element name="hosts">
181     <xs:complexType>
182       <xs:sequence>
183         <xs:element ref="host" maxOccurs="unbounded" minOccurs="0"/>
184       </xs:sequence>
185     </xs:complexType>
186   </xs:element>
187   <xs:element name="config">
188     <xs:complexType>
189       <xs:sequence>
190         <xs:element ref="layer2networks"/>
191         <xs:element ref="layer3networks"/>
192         <xs:element ref="hosts"/>
193       </xs:sequence>
194       <xs:attribute type="xs:string" name="name"/>
195     </xs:complexType>
196   </xs:element>
197 </xs:schema>
```

# Appendix B

# Function Test: Environment Configuration

```
dataDirectory = data/
continuousExporters = JsonOutput

[modules]
   [[XenHostsCollector]]
      type = XenHostScan
      config = scannerConf/xen.conf
      interval = 5

   [[XenHardwareInfoCollector]]
      type = XenHardwareScan
      config = scannerConf/xen.conf
      interval = 5

   [[XenNetworksCollector]]
      type = XenNetworkScan
      config = scannerConf/xen.conf
      interval = 5

   [[XenInterfacesCollector]]
      type = XenInterfaceScan
      config = scannerConf/xen.conf
      interval = 5

   [[ServiceScan]]
      type = NmapService
      config = scannerConf/service.conf
      interval = 40

   [[SnmpRouting]]
      type = SnmpRoutingCollector
      config = scannerConf/snmp.conf
      interval = 20

   [[SnmpInterfaceCollector]]
      type = SnmpInterfaceCollector
      config = scannerConf/snmp.conf
      interval = 10

```

```
40    [[IpAddressInformationCollector]]
41       type = SSHKeyScriptInterfaceConfigurationScan
42       config = scannerConf/ssh.conf
43       interval = 10
44
45    [[DnsInfoCollector]]
46       type = SSHKeyDnsmasqScriptScan
47       config = scannerConf/ssh.conf
48       interval = 10
49
50    [[DhcpInfoCollector]]
51       type = SSHKeyScriptDHCPScan
52       config = scannerConf/ssh.conf
53       interval = 10
54
55    [[RoutingInformationCollector]]
56       type = SSHKeyScriptRoutingScan
57       config = scannerConf/ssh.conf
58       interval = 10
59 [triggeredExporters]
```

# Appendix C

# iLab Case Study

## C.1 iLab Case Study: Environment Configuration

```
dataDirectory = data/
continuousExporters = JsonOutput

[modules]
   [[DumpCollector]]
      type = TcpdumpHostCollector
      config = scannerConf/dump.conf
      interval = 20

   [[InterfaceCollector]]
      type = SSHKeyScriptInterfaceConfigurationScan
      config = scannerConf/interface.conf
      interval = 5

   [[StaticInformation]]
      type = XmlScanner
      config = scannerConf/static.conf
      interval = 10

   [[L3toL2]]                    # Converts layer 3 networks to layer 2 networks
      type = L3toL2Network
      config = scannerConf/empty.conf
      interval = 10

   [[RoutingCisco]]
      type = SnmpRoutingCollector
      config = scannerConf/ciscoRouting.conf
      interval = 20

   [[RoutingLinux]]
      type = SSHKeyScriptRoutingScan
      config = scannerConf/interface.conf
      interval = 20
```

## C.2 iLab Case Study: Static Information

```
<config id="Case2Environment">
   <hosts>
      <host id="10.0.1.254" location="physical" template="router-base">
         <interfaces>
            <interface mac="64:f6:9d:12:db:7c" network="10.0.1.0" static="True">
               <layer3address address="10.0.1.254" network="10.0.1.0/24" />
            </interface>
            <interface mac="64:f6:9d:12:db:7d" network="10.0.4.0" static="True">
               <layer3address address="10.0.4.1" network="10.0.4.0/24" />
            </interface>
         </interfaces>
         <disks />
         <routes />
         <firewall>
            <firewallRules />
         </firewall>
      </host>
      <host id="10.0.3.254" location="physical" template="router-base">
         <interfaces>
            <interface mac="a8:9d:21:86:d0:e8" network="10.0.3.0" static="True">
               <layer3address address="10.0.3.254" network="10.0.3.0/24" />
            </interface>
            <interface mac="a8:9d:21:86:d0:e9" network="10.0.5.0" static="True">
               <layer3address address="10.0.5.2" network="10.0.5.0/24" />
            </interface>
         </interfaces>
         <disks />
         <routes />
         <firewall>
            <firewallRules />
         </firewall>
      </host>
      <host id="10.0.2.254" location="physical" template="router-base">
         <interfaces>
            <interface mac="90:e2:ba:7f:78:4b" network="10.0.2.0" static="True">
               <layer3address address="10.0.2.254" network="10.0.2.0/24" />
            </interface>
            <interface mac="90:e2:ba:7f:78:4a" network="10.0.5.0" static="True">
               <layer3address address="10.0.5.1" network="10.0.5.0/24" />
            </interface>
            <interface mac="90:e2:ba:7f:78:49" network="10.0.4.0" static="True">
               <layer3address address="10.0.4.2" network="10.0.4.0/24" />
            </interface>
         </interfaces>
         <disks />
         <routes />
         <firewall>
            <firewallRules />
         </firewall>
      </host>
   </hosts>
</config>
```

## C.3 iLab Case Study: Full Network Topology Information

```
<?xml version="1.0"?>
<config name="case2">
  <hosts>
    <host id="10.0.1.2" location="physical" template="host-base">
      <interfaces>
        <interface mac="90:e2:ba:7e:71:91" mtu="1500" />
        <interface mac="90:e2:ba:7e:71:93" mtu="1500" network="10.0.1.0" rate="1000000">
          <layer3address address="10.0.1.2" gateway="10.0.1.254" netmask="255.255.255.0"
      network="10.0.1.0/24" static="True">
            <services/>
          </layer3address>
        </interface>
        <interface mac="48:5d:60:76:bc:eb" mtu="1500" />
        <interface mac="d0:50:99:52:cc:80" mtu="1500" network="192.168.38.0" rate="1000000">
          <layer3address address="192.168.38.2" netmask="255.255.255.0"
      network="192.168.38.0/24" static="True">
            <services/>
          </layer3address>
        </interface>
      </interfaces>
      <disks/>
      <routes/>
      <firewall>
        <firewallRules/>
      </firewall>
    </host>
    <host id="10.0.2.1" location="physical" template="host-base">
      <interfaces>
        <interface mac="90:e2:ba:82:52:2d" mtu="1500" />
        <interface mac="00:25:d3:39:ec:10" mtu="1500" />
        <interface mac="90:e2:ba:82:52:2f" mtu="1500" network="10.0.2.0" rate="1000000">
          <layer3address address="10.0.2.1" gateway="10.0.2.254" netmask="255.255.255.0"
      network="10.0.2.0/24" static="True">
            <services/>
          </layer3address>
        </interface>
        <interface mac="d0:50:99:52:cd:2e" mtu="1500" network="192.168.38.0" rate="1000000">
          <layer3address address="192.168.38.3" netmask="255.255.255.0"
      network="192.168.38.0/24" static="True">
            <services/>
          </layer3address>
        </interface>
      </interfaces>
      <disks/>
      <routes/>
      <firewall>
        <firewallRules/>
      </firewall>
    </host>
    <host id="10.0.2.254" location="physical" template="router-base">
      <interfaces>
        <interface mac="90:e2:ba:7f:78:49" mtu="1500" network="10.0.4.0" rate="100000">
          <layer3address address="10.0.4.2" netmask="255.255.255.0" network="10.0.4.0/24"
      static="True">
            <services/>
```

```
51          </layer3address>
52        </interface>
53        <interface mac="90:e2:ba:7f:78:4a" network="10.0.5.0" mtu="1500" rate="100000">
54          <layer3address address="10.0.5.1" netmask="255.255.255.0" network="10.0.5.0/24"
      static="True">
55            <services/>
56          </layer3address>
57        </interface>
58        <interface mac="48:5d:60:77:32:47" mtu="1500" />
59        <interface mac="d0:50:99:52:cd:15" mtu="1500" network="192.168.38.0" rate="1000000">
60          <layer3address address="192.168.38.5" netmask="255.255.255.0"
      network="192.168.38.0/24" static="True">
61            <services/>
62          </layer3address>
63        </interface>
64        <interface mac="90:e2:ba:7f:78:4b" mtu="1500" network="10.0.2.0" rate="1000000">
65          <layer3address address="10.0.2.254" netmask="255.255.255.0" network="10.0.2.0/24"
      static="True">
66            <services/>
67          </layer3address>
68        </interface>
69      </interfaces>
70      <disks/>
71      <routes>
72        <route destination="10.0.1.0" gateway="10.0.4.1" genmask="255.255.255.0"
      interface="enx90e2ba7f7849"/>
73        <route destination="10.0.3.0" gateway="10.0.5.2" genmask="255.255.255.0"
      interface="enx90e2ba7f784a"/>
74      </routes>
75      <firewall>
76        <firewallRules/>
77      </firewall>
78    </host>
79    <host id="10.0.1.1" location="physical" template="host-base">
80      <interfaces>
81        <interface mac="90:e2:ba:7f:78:4f" mtu="1500" network="10.0.1.0" rate="1000000">
82          <layer3address address="10.0.1.1" gateway="10.0.1.254" netmask="255.255.255.0"
      network="10.0.1.0/24" static="True">
83            <services/>
84          </layer3address>
85        </interface>
86        <interface mac="48:5d:60:77:5b:db" mtu="1500" />
87        <interface mac="90:e2:ba:7f:78:4d" mtu="1500" />
88        <interface mac="d0:50:99:52:cc:f3" mtu="1500" network="192.168.38.0" rate="1000000">
89          <layer3address address="192.168.38.1" netmask="255.255.255.0"
      network="192.168.38.0/24" static="True">
90            <services/>
91          </layer3address>
92        </interface>
93      </interfaces>
94      <disks/>
95      <routes/>
96      <firewall>
97        <firewallRules/>
98      </firewall>
99    </host>
100   <host id="10.0.1.254" location="physical" template="router-base">
101     <interfaces>
102       <interface mac="64:f6:9d:12:db:7c" network="10.0.1.0">
```

```
103           <layer3address address="10.0.1.254" netmask="255.255.255.0" network="10.0.1.0/24"
      static="True">
104             <services/>
105           </layer3address>
106         </interface>
107         <interface mac="64:f6:9d:12:db:7d" network="10.0.4.0">
108           <layer3address address="10.0.4.1" netmask="255.255.255.0" network="10.0.4.0/24"
      static="True">
109             <services/>
110           </layer3address>
111         </interface>
112       </interfaces>
113       <disks/>
114       <routes>
115         <route destination="10.0.2.0" gateway="10.0.4.2" genmask="255.255.255.0"/>
116         <route destination="10.0.5.0" gateway="10.0.4.2" genmask="255.255.255.0"/>
117       </routes>
118       <firewall>
119         <firewallRules/>
120       </firewall>
121     </host>
122     <host id="10.0.3.254" location="physical" template="router-base">
123       <interfaces>
124         <interface mac="a8:9d:21:86:d0:e8" network="10.0.3.0">
125           <layer3address address="10.0.3.254" netmask="255.255.255.0" network="10.0.3.0/24"
      static="True">
126             <services/>
127           </layer3address>
128         </interface>
129         <interface mac="a8:9d:21:86:d0:e9" network="10.0.5.0">
130           <layer3address address="10.0.5.2" netmask="255.255.255.0" network="10.0.5.0/24"
      static="True">
131             <services/>
132           </layer3address>
133         </interface>
134       </interfaces>
135       <disks/>
136       <routes>
137         <route destination="10.0.4.0" gateway="10.0.5.1" genmask="255.255.255.0"/>
138         <route destination="10.0.2.0" gateway="10.0.5.1" genmask="255.255.255.0"/>
139       </routes>
140       <firewall>
141         <firewallRules/>
142       </firewall>
143     </host>
144     <host id="10.0.1.3" location="physical" template="host-base">
145       <interfaces/>
146       <disks/>
147       <routes/>
148       <firewall>
149         <firewallRules/>
150       </firewall>
151     </host>
152     <host id="10.0.3.1" location="physical" template="host-base">
153       <interfaces>
154         <interface mac="90:e2:ba:82:51:6f" mtu="1500" network="10.0.3.0" rate="100000">
155           <layer3address address="10.0.3.1" gateway="10.0.3.254" netmask="255.255.255.0"
      network="10.0.3.0/24" static="True">
156             <services/>
```

```
157           </layer3address>
158         </interface>
159         <interface mac="48:5d:60:76:e1:ca" mtu="1500" />
160         <interface mac="d0:50:99:52:cb:e4" mtu="1500" network="192.168.38.0" rate="1000000">
161           <layer3address address="192.168.38.4" netmask="255.255.255.0"
      network="192.168.38.0/24" static="True">
162             <services/>
163           </layer3address>
164         </interface>
165         <interface mac="90:e2:ba:82:51:6d" mtu="1500" />
166       </interfaces>
167       <disks/>
168       <routes/>
169       <firewall>
170         <firewallRules/>
171       </firewall>
172     </host>
173   </hosts>
174   <layer2networks>
175     <layer2network id="10.0.1.0" location="physical" />
176     <layer2network id="192.168.38.0" location="physical" />
177     <layer2network id="10.0.3.0" location="physical" />
178     <layer2network id="10.0.2.0" location="physical" />
179     <layer2network id="10.0.4.0" location="physical" />
180     <layer2network id="10.0.5.0" location="physical" />
181   </layer2networks>
182   <layer3networks>
183     <layer3network address="10.0.2.0" id="10.0.2.0/24" netmask="255.255.255.0"/>
184     <layer3network address="10.0.1.0" id="10.0.1.0/24" netmask="255.255.255.0"/>
185     <layer3network address="192.168.38.0" id="192.168.38.0/24" netmask="255.255.255.0"/>
186     <layer3network address="10.0.3.0" id="10.0.3.0/24" netmask="255.255.255.0"/>
187     <layer3network address="10.0.4.0" id="10.0.4.0/24" netmask="255.255.255.0"/>
188     <layer3network address="10.0.5.0" id="10.0.5.0/24" netmask="255.255.255.0"/>
189   </layer3networks>
190 </config>
```

# List of Figures

# List of Tables

# List of Listings

# Bibliography

[1] Nmap Port Scan Techniques. Accessed: 18/09/2016. [Online]. Available: nmap.org/book/man-port-scanning-techniques.html

[2] M. de Vivo, E. Carrasco, G. Isern, and G. O. de Vivo, "A Review of Port Scanning Techniques," *SIGCOMM Comput. Commun. Rev.*, vol. 29, no. 2, pp. 41–48, Apr. 1999.

[3] G. Lyon.

[4] Openvas Team, "Openvas Homepage," www.openvas.org/about.html, Accessed: 02/09/2016.

[5] Nmap Version Detection. Accessed: 18/09/2016. [Online]. Available: nmap.org/man/de/man-version-detection.html

[6] S. Kalia and M. Singh, "Masking approach to secure systems from Operating system Fingerprinting," in *TENCON 2005 2005 IEEE Region 10*, Nov 2005, pp. 1–6.

[7] F. Yarochkin, O. Arkin, and M. Kydyraliev, "xprobe2(1) – Linux man page," linux.die.net/man/1/xprobe2, Accessed: 11/09/2016.

[8] A. Khakpour, J. Hulst, Z. Ge, A. Liu, D. Pei, and J. Wang, "Firewall fingerprinting," in *INFOCOM, 2012 Proceedings IEEE*, March 2012, pp. 1728–1736.

[9] C. Hamby, "Firewall Fingerprinting: Using default TCP/UDP port combinations and Nmap to identify firewall types in a network," www.giac.org/paper/gsec/2635/firewall-fingerprinting-default-tcp-udp-port-combinations-nmap-identify-firewall-type/104520, February 2003, accessed: 22/08/2016.

[10] J. D. Case, M. Fedor, M. L. Schoffstall, and J. R. Davin, "Simple Network Management Protocol (SNMP)," IETF, IETF, RFC, May 1990, accessed: 10/09/2016. [Online]. Available: www.ietf.org/rfc/rfc1157.txt

[11] K. McCloghrie and M. Rose, "Management Information Base for network management of TCP/IP-based internets," IETF, IETF, RFC 1066, August 1988, accessed: 10/09/2016. [Online]. Available: www.ietf.org/rfc/rfc1066.txt

[12] R. Enns and M. Bjorklund and J. Schoenwaelder and A. Bierman, "Network Configuration Protocol (NETCONF)," IETF, IETF, RFC 6241, June 2011, Accessed: 10/09/2016. [Online]. Available: tools.ietf.org/html/rfc6241

[13] J. Pablo and A. Nikolić, "Running Remote Commands," msdn.microsoft.com/en-us/powershell/scripting/core-powershell/running-remote-commands, Accessed: 14/11/2016.

[14] C. Rudolf, "Automated Planning, Setup and Configuration for Scientific Testbed Environments," Bachelor's Thesis, Technische Universität München, 2016.

[15] J. Ursi, "IF-MAP Overview," www.itu.int/en/ITU-T/studygroups/com17/Documents/tutorials/2011/IFMAPOverview.pdf, 2009, accessed: 27/08/2016.

[16] V. Ahlers, F. Heine, B. Hellmann, C. Kleiner, L. Renners, T. Rossow, and R. Steuerwald, *Integrated Visualization of Network Security Metadata from Heterogeneous Data Sources.* Cham: Springer International Publishing, 2016, pp. 18–34.

[17] Trusted Network Connect Work Group, "TNC IF-MAP Bindings for SOAP, Version 2.2, Revision 10," www.trustedcomputinggroup.org/tnc-if-map-binding-soap-specification, March 2014, accessed: 27/08/2016.

[18] Trusted Network Connect Work Group, "TNC MAP Content Authorization, Version 1.0, Revision 36," www.trustedcomputinggroup.org/tnc-map-content-authorization, June 2014, accessed: 27/08/2016.

[19] J. Undercoffer, A. Joshi, and J. Pinkston, *Modeling Computer Attacks: An Ontology for Intrusion Detection.* Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 113–135.

[20] Jeroen Johannes van der Ham, "A Semantic Model for Complex Computer Networks: The Network Description Language," Ph.D. dissertation, University of Amsterdam, 2010. [Online]. Available: dare.uva.nl/record/1/318784

[21] M. Ghijsen, J. van der Ham, P. Grosso, C. Dumitru, H. Zhu, Z. Zhao, and C. de Laat, "A semantic-web approach for modeling computing infrastructures," *Computers & Electrical Engineering*, vol. 39, no. 8, pp. 2553 – 2565, 2013.

[22] T. Taketa and Y. Hiranaka, "Network Design Assistant System based on Network Description Language," in *Advanced Communication Technology (ICACT), 2013 15th International Conference on*, Jan 2013, pp. 515–518.

[23] J. van der Ham, F. Dijkstra, R. Łapacz, and J. Zurawski, "Network Markup Language Base Schema version 1," hdl.handle.net/11245/1.395255, May 2013.

[24] R. van der Pol and F. Dijkstra, "Data Exchange between Network Monitoring Tools," kirk.rvdp.org/publications/AIMS2009.pdf, 2009.

[25] F. Dijkstra, "NML Topology Model," www.fp7-novi.eu/novidissemination/im-modeling-workshop/doc_view/59-nml-topology-model, September 2012.

[26] J. van der Ham, F. Dijkstra, R. Łapacz, and A. Brown, "The Network Markup Language (NML): A Standardized Network Topology Abstraction for Inter-domain and Cross-layer Network Applications," tnc2013.terena.org/getfile/148, 2013.

[27] H. Birkholz, I. Sieverdingbeck, K. Sohr, and C. Bormann, "IO: An Interconnected Asset Ontology in Support of Risk Management Processes," in *Availability, Reliability and Security (ARES), 2012 Seventh International Conference on*, Aug 2012, pp. 534–541.

[28] L. Lorenzin and N. Cam-Winget, "Security Automation and Continuous Monitoring (SACM) Requirements," Internet Engineering Task Force, Internet-Draft draft-ietf-sacm-requirements-13, Mar. 2016, work in Progress. [Online]. Available: tools.ietf.org/html/draft-ietf-sacm-requirements-13

[29] H. Birkholz and I. Sieverdingbeck, "Improving Root Cause Failure Analysis in Virtual Networks via the Interconnected-asset Ontology," in *Proceedings of the Conference on Principles, Systems and Applications of IP Telecommunications*, ser. IPTComm '14. New York, NY, USA: ACM, 2014, pp. 2:1–2:8.

[30] H. Birkholz and I. Sieverdingbeck, "Link-failure assessment in redundant ICS networks supported by the interconnected-asset ontology," in *2014 IEEE International Workshop Technical Committee on Communications Quality and Reliability (CQR)*, May 2014, pp. 1–6.

[31] "GÉANT project website," geant3.archive.geant.net/service/cnis/pages/home.aspx, Accessed: 08/28/2016.

[32] "cNIS Data Model Schema," forge.geant.net/forge/display/CNIS30DOC/Database+schema, Accessed: 08/28/2016.

[33] M. Wolski, A. Patil, C. Mazurek, and M. Łabędzki, "common Network Information Service - Modelling and interacting with a real life network," geant3.archive.geant.net/service/cnis/Resources/Documents/cnis-presentation-05.pdf, 2009, Accessed: 08/28/2016.

[34] C. Dobre, R. Voicu, and I. Legrand, "Monitoring large scale network topologies," in *Intelligent Data Acquisition and Advanced Computing Systems (IDAACS), 2011 IEEE 6th International Conference on*, vol. 1, Sept 2011, pp. 218–222.

[35] "MonALISA Project Website," monalisa.cacr.caltech.edu/monalisa.htm, Accessed: 08/28/2016.

[36] A. Carpen-Amarie, J. Cai, A. Costan, G. Antoniu, and L. Bougé, "Bringing Introspection Into the BlobSeer Data-Management System Using the MonALISA

Distributed Monitoring Framework," in *Complex, Intelligent and Software Intensive Systems (CISIS), 2010 International Conference on*, Feb 2010, pp. 508–513.

[37] B. Tierney, J. Metzger, J. Boote, E. Boyd, A. Brown, R. Carlson, M. Zekauskas, J. Zurawski, M. Swany, and M. Grigoriev, "perfSONAR: Instantiating a Global Network Measurement Framework," in *In SOSP Workshop on Real Overlays and Distributed Systems (ROADS '09)*. ACM, October 2009.

[38] "perfSONAR Project Website," www.perfsonar.net/about, Accessed: 08/28/2016.

[39] R. Deraison, "NASL Reference Guide," virtualblueness.net/nasl.html, Accessed: 09/02/2016.

[40] G. Lyon, *The Offical Nmap Project Guide to Network Discovery and Securtiy Scanning*. Nmap Project, 2009, Accessed: 02/09/2016.

[41] C. Diekmann, L. Hupel, and G. Carle, "FFFUU," www.github.com/diekmann/Iptables_Semantics/tree/master/haskell_tool, 2016, Accessed: 22/10/2016.

[42] C. Rudolf and M. Dorfhuber, "INSALATA," www.github.com/tumi8/INSALATA, 2016, Accessed: 24/11/2016.

[43] Python Software Foundation, "Python," www.python.org, Accessed: 22/10/2016.

[44] Citrix Systems, Inc, "Citrix XenServer Management API," docs.citrix.com/content/dam/en-us/xenserver/xenserver-65/xenserver65sp1_management_api_guide.pdf, Accessed: 15/11/2016.

[45] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, "Xen and the Art of Virtualization," in *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, ser. SOSP '03. New York, NY, USA: ACM, 2003, pp. 164–177.

[46] M. DeHaan, "Ansible," www.ansible.com, Accessed: 07/09/2016.

[47] Zabbix LLC, "Zabbix," www.zabbix.com/, Accessed: 22/10/2016.

[48] A. Dubkov, "py-zabbix," py-zabbix.readthedocs.io/en/latest, Accessed: 22/10/2016.

[49] J. Forcier and R. Pointer, "Paramiko," www.paramiko.org/, Accessed: 22/10/2016.

[50] I. Etingof, "PySNMP," pysnmp.sourceforge.net/, Accessed: 22/10/2016.

[51] M. Faassen and S. Behnel, "lxml," lxml.de, Accessed: 15/11/2016.

[52] J. Hermann and R. Dennis, "configobj," www.github.com/DiffSK/configobj, 2016, Accessed: 15/11/2016.

[53] D. P. D. Moss and S. Nordhausen, "netaddr," www.pythonhosted.org/netaddr, Accessed: 15/11/2016.

[54] V. Gropp, “bwm-ng(1) – Linux man page,” www.linux.die.net/man/1/bwm-ng, Accessed: 04/12/2016.