CZECH TECHNICAL UNIVERSITY IN PRAGUE
FACULTY OF INFORMATION TECHNOLOGY

ASSIGNMENT OF MASTER'S THESIS

Title:	Unified configuration interface for NEMEA collectors
Student:	Bc. Matěj Židek
Supervisor:	Ing. Tomáš Čejka
Study Programme:	Informatics
Study Branch:	Web and Software Engineering
Department:	Department of Software Engineering
Validity:	Until the end of winter semester 2018/19

Instructions

Study the Yang modeling language [1] for description of configuration data models and the sysrepo configuration system [2,3] that is used for validation, storage, and manipulation with configuration data according to data models.
Study the NEMEA system [4,5] (a set of software modules for network traffic analysis and anomaly detection) and its related tools of the STaaS [6] collector.
Design a configuration data model for the NEMEA system and STaaS that is needed for sysrepo.
Implement web GUI and web API that allow users to configure NEMEA and STaaS using sysrepo (the complete list of configuration and state information will be provided by the supervisor).
Generally, the GUI and API must support access to the configuration and state data and modification of the configuration data that is stored in sysrepo.
Test the implemented GUI and API.

References

[1] https://tools.ietf.org/html/rfc6020
[2] http://www.sysrepo.org/
[3] http://www.sysrepo.org/sysrepo-datastore?action=AttachFile&do=get&target=poster.pdf
[4] T. Cejka, V. Bartoš, M. Svepes, Z. Rosa, and H. Kubatova, "NEMEA: A Framework for Network Traffic Analysis," in *12th International Conference on Network and Service Management (CNSM 2016)*, Montreal, Canada, 2016.
[5] https://github.com/CESNET/NEMEA
[6] https://github.com/CESNET/STaaS
[7] Alexa, David. Rozšíření grafického uživatelského rozhraní NetopeerGUI. Master thesis. Czech Technical University in Prague, Faculty of Information Technology, czech language, 2015.

Ing. Michal Valenta, Ph.D.
Head of Department

prof. Ing. Pavel Tvrdík, CSc.
Dean

Prague September 15, 2017

FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Master's thesis

Unified configuration interface for NEMEA collectors

Bc. Matěj Židek

Department of Software Engineering
Supervisor: Ing. Tomáš Čejka

May 9, 2018

Acknowledgements

I would like to thank my dearest Mgr. Jolana Rubešová for all the patience and help. Thanks go of course also to my supervisor Ing. Tomáš Čejka for his time and guidance and lastly, I thank Bc. Hedvika Pohlová and Václav Bartoš for their assistance.

Declaration

In Prague on May 9, 2018 . . . .

Citation of this thesis

Žídek, Matěj. *Unified configuration interface for NEMEA collectors.* Master's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2018.

Abstrakt

V této práci se zabývám novým způsobem konfigurace modulárního systému NEMEA, který slouží pro analýzu provozu a detekci anomálií v počítačových sítích. Tento nový způsob spočívá ve využití projektu sysrepo jako úložiště konfigurace a prostředníka pro komunikaci se spuštěným NEMEA systémem.

Problém jsem vyřešil implementací nové verze démona, který je v rámci NEMEA projektu odpovědný za správu běhu všech modulů.

Nový démon je navržen, aby používal sysrepo. To umožnilo přímo nad sysrepem postavit webové uživatelské rozhraní, které dovoluje běžící NEMEA systém spravovat.

Celé řešení jsem pro snadné nasazení sestavil v prostředí vytvořeném technologií Docker.

Klíčová slova NEMEA, sysrepo, supervisor, démon, angular

Abstract

The thesis focuses on creating a new method for configuring the modular NEMEA system that is used to analyze computer network traffic and detect anomalies that may occur. The new solution is based on using the sysrepo project as a configuration datastore and as a medium for communication with the running NEMEA system.

The issue was solved by an implementation of a new version of its daemon that manages runtime of all the modules within the NEMEA project.

The new daemon was designed to use sysrepo. This approach facilitated building a new web graphical user interface on top of sysrepo allowing to manage the running NEMEA system.

To make a deployment easier, the solution was built in an environment created with the Docker technology.

Keywords NEMEA, sysrepo, supervisor, daemon, angular

Contents

List of Figures

List of Tables

Introduction

NEMEA is a modular system used for network traffic analysis and anomaly detection developed in the CESNET association. Each of its modules is independent and takes care of different tasks within NEMEA. However, there is a need for a unified configuration interface in the case of larger deployments. For this purpose, NEMEA contains a supervisor daemon that allows managing all the modules from one place. Originally, the NEMEA Supervisor was based on manually written XML files and templates. This approach was time-consuming and not user-friendly. The new concept presented in this thesis uses the sysrepo project as a configuration datastore instead of raw files. In addition, sysrepo provides many useful features that were only partially covered or completely missing in the previous version. Sysrepo handles validation of the user's configuration data and therefore there is no need to implement complex logic related to handling XML files and semantic relations in the configuration data. To manage the system even more conveniently, a new graphical web user interface (GUI) for the NEMEA system administration is going to be built on top of sysrepo.

First, I provide the readers with a broader view of the whole topic by introducing the NEMEA system and the STaaS environment in which the NEMEA system is used. I then continue in the introduction by describing the existing configuration subsystem of NEMEA and technologies that are more convenient for use with the new subsystem. Next, the analysis of the new configuration solution follows. The analysis can be divided into three main parts — the data model, the new Supervisor and the design draft and technologies used in the graphical user interface. Since the entire solution is constituted by many components, I also pay attention to a platform that is used for its easier deployment. After that, a detailed description of the implementation of the whole solution is provided. Finally, the testing methods used during the development along with the assessment of the solution are discussed. The final evaluation and several suggestions for the possible future work can be found there as well.

Chapter 1

Introduction to the Topic

Since the purpose of this thesis project is all about making the configuration of the NEMEA system less painful, in this chapter I would like to first introduce the system to the readers. Afterward, it should become more obvious what are the shortcomings of the previously used configuration approach and what could be improved. The introduction focuses on explaining the concepts of the YANG configuration modeling language. It will cover its usage in the sysrepo configuration datastore and it will assess the suitability of those concepts for the NEMEA's needs. Next, the chapter presents NETCONF as a protocol for remote configuration and the Netopeer project that integrates both sysrepo and NETCONF and also provides a graphical user interface that helps manage NETCONF enabled applications. The last part of this chapter is dedicated to Security Tools as a Service (STaaS) as the environment in which the new configuration of NEMEA should be applied.

1.1 NEMEA

NEMEA is an open source modular system for network traffic analysis and anomaly detection based on IP flows [1]. It consists of three main parts: modules, framework and Supervisor.

1.1.1 Modules

NEMEA modules are separate programs that are interconnected using an interface mechanism. We can imagine the input interfaces as clients and the output interfaces as servers. Therefore, in a scenario of one-way communication between modules A and B, the module A creates an output interface while the module B only connects to it. In the NEMEA terminology, this connection is referred to as its input interface. There are multiple types of NEMEA's interfaces for both communication on the same host (e.g. UNIX socket) and across a computer network (e.g. TCP socket).

Although each module is different, we can put most of them into one of four main categories: *data sources*, *detectors*, *reporters* and *loggers*. Data sources, typically IPFIXcol[1] (flow collector) or flow_meter[2] (flow exporter from NEMEA project), are converting network traffic data into an UniRec[3] format that is used in NEMEA. UniRec messages are afterward sent via output interfaces from which detectors take data and try to detect network anomalies or attacks. Detectors then take the data from the output interfaces and try to detect any network anomalies or attacks. The detectors provide results of what they compute and put it on their output interfaces for reporter and logger modules that can then notify users, log into a file or send intrusion detection extensible alert (IDEA) [2] messages to other security analysis or anomaly detection systems.

1.1.2 NEMEA Framework

The NEMEA framework[4] is a collection of libraries providing most of the shared functionality for modules. The most significant of them is libtrap[5], a library providing means of communication between modules, and UniRec that is a key-value data format for exchanging messages across modules.

1.1.3 The NEMEA Supervisor in Depth

One of NEMEA's advantages is its modularity that follows the UNIX philosophy [3] prescribing each module a single purpose that should be fulfilled to the highest degree possible. That, however, can be quite inconvenient when we want to use multiple NEMEA modules. For this case, the NEMEA Supervisor comes in to manage all the modules from one place. This daemon service takes care of starting and stopping of modules, delivering configuration (command line arguments), gathering state data about modules' interfaces and collecting information about the usage of system resources.

1.1.4 Configuration

The Supervisor takes care of providing configuration to each module it manages. This configuration model uses XML files with its own templating system. It consists of one template file that includes statements to import settings of individual modules from separate files with the `.sup` extension. The `.sup` files are also using the XML format and they are included into the template at runtime by the Supervisor to form one big parsable XML.

[1] `https://github.com/CESNET/ipfixcol`

[2] `https://github.com/CESNET/Nemea-Modules/tree/master/flow_meter`

[3] `https://github.com/CESNET/NEMEA-Framework/tree/master/unirec`

[4] `https://github.com/CESNET/NEMEA-Framework/tree/master`

[5] `https://github.com/CESNET/NEMEA-Framework/tree/master/libtrap`

Figure 1.1: Typical directory listing of the original NEMEA Supervisor.

```
main-configuration-directory
├── data-sources
│   ├── flow_meter.sup
│   └── ipfixcol.sup
├── detectors
│   ├── amplification_detection.sup
│   ├── ...
│   └── vportscan_detector.sup
├── loggers
│   ├── amplification_logger.sup
│   ├── ...
│   └── vportscan_aggr_logger.sup
├── others
│   ├── ipv6stats.sup
│   └── traffic_repeater.sup
└── supervisor_config_template.xml.in
```

To help manage multiple modules at once, they can be grouped into profiles so that the administrator can control all the modules as a whole. Profiles are represented as folders inside the Supervisor's configuration directory.

In the directory listing in figure 1.1, we can see an example of how the content of such directory may look like to help a reader gain a better insight.

Now, we should have an idea of how the configuration gets to the Supervisor, but how does it get to individual modules? Again to better explain it I provide the reader with an example featuring a content of configuration for *voip_fraud_detection* module in a listing 1.1. We can see the element `<trapinterfaces>` from which the Supervisor creates a command line parameter understandable to libtrap [4] and adds it to the rest of the parameters in a `<param>` node to form the following string that is passed on on module startup: `-i "u:voip_data_source,t:12003" -o -w -l /data/log`. Although this custom templating model of splitting one configuration file into multiple smaller ones is better readable, it does, on the other hand, make the Supervisor's code more complex by adding extra logic.

```
<module>
  <name>voip_fraud_detection</name>
  <enabled>true</enabled>
  <path>/usr/bin/nemea/voip_fraud_detection</path>
  <params>
    -o -w -l /data/log
  </params>
  <trapinterfaces>
    <interface>
      <type>UNIXSOCKET</type>
      <direction>IN</direction>
      <params>data_source</params>
    </interface>
    <interface>
      <type>TCP</type>
      <direction>OUT</direction>
      <params>12003</params>
    </interface>
  </trapinterfaces>
</module>
```

Listing 1.1: XML configuration of *voip_fraud_detection* NEMEA module in the original NEMEA Supervisor.

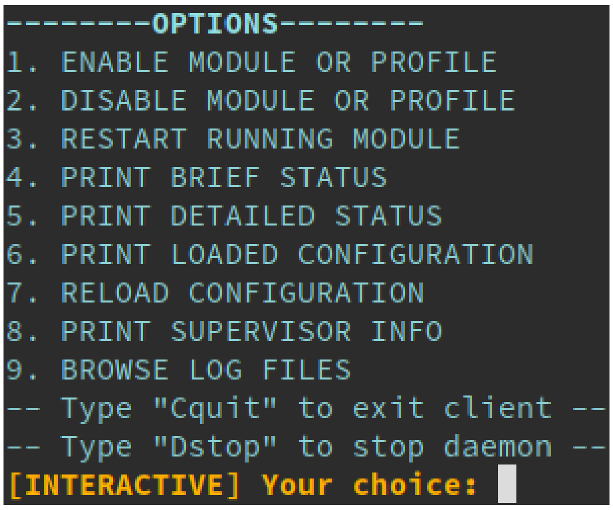
1.1.5 Runtime Management

The Supervisor (generally) takes care of starting, stopping and restarting all modules. Modules that are enabled at the Supervisor's startup are launched. In case they fail to start, the Supervisor tries to relaunch them again. This is repeated as many times as indicated by the given limit number of restarts per minute for the specific module. If the number of attempts exceeds the given limit number, the module goes into a disabled mode and the Supervisor no longer tries to start it.

Starting, stopping and restarting the module can be done even while the Supervisor is running. These operations can be triggered by the Supervisor client as described later in section 1.1.7.

1.1.6 Runtime Statistics

Modules have the option to use libtrap's helper macros and functions to make using their NEMEA interfaces easier. The libtrap library also creates a new UNIX socket on their behalf that a program can connect to and read statistics about data flowing through module's interfaces. The Supervisor connects to



```
--------OPTIONS--------
1. ENABLE MODULE OR PROFILE
2. DISABLE MODULE OR PROFILE
3. RESTART RUNNING MODULE
4. PRINT BRIEF STATUS
5. PRINT DETAILED STATUS
6. PRINT LOADED CONFIGURATION
7. RELOAD CONFIGURATION
8. PRINT SUPERVISOR INFO
9. BROWSE LOG FILES
-- Type "Cquit" to exit client --
-- Type "Dstop" to stop daemon --
[INTERACTIVE] Your choice:
```

Figure 1.2: NEMEA Supervisor client.

this so-called service interface of each module and collects statistics about the count of messages passed through the given interface, and more.

Besides collecting state data about interfaces, the Supervisor also reads memory and CPU usage statistics from the system.

1.1.7 Supervisor Client

The figure 1.2 depicts what is presented at the command line when a user starts the Supervisor client. Under the hood, there is a thin client that connects to the UNIX socket provided by the Supervisor daemon. The client receives a text to be printed from the UNIX socket and also uses it to send commands.

1.1.8 YANG

Before I discuss the sysrepo datastore in detail, I would like to present the language it uses to describe data models — YANG — as it is later used to define the new data model for NEMEA. The initial version[6] was defined in RFC 6020 [6] in the following words: "YANG is a data modeling language used to model configuration and state data manipulated by the Network Configuration Protocol (NETCONF), NETCONF remote procedure calls, and NETCONF notifications.". Although YANG was created for NETCONF, it can still be used without it as we will see later in section 1.2.

[6]There is also YANG version 1.1 defined in RFC 7950 [5].

1.1.9 Data Models

A single configuration data model in YANG is called a *module*. It is a hierarchical structure and its nodes can be made of leaves (e.g. *leaf* statement), array of nodes (e.g. *list* statements), group of other nodes (e.g. *container* statement) and others. There are many features for creating data models in YANG. For example, nodes in this tree structure can be available only under a certain condition and values of data nodes can be restricted to some values only. To evaluate references between nodes and to query data modeled by YANG, the XML Path Language (XPath) in version 1.0 [7] is used. Each data node has its own data type. It can either be one of the YANG's built-in types (e.g. *uint16*, *string*, *boolean*) or it can be extended from those. A module itself is also extensible and can load data from other modules or submodules (*submodule* statement).

This data model does not only provide configuration data, it also contains definitions of how operational data should look like. Operational data[7] are just ordinary nodes in the YANG's tree structure, the only difference is that they have a property *config* with a value of *false*.

1.1.10 YIN

There is another representation of the data models called YIN (YANG Independent Notation). YIN is essentially YANG in the XML syntax. That is convenient because opposite to YANG, many parsers for XML exist. From this fact we can also see why we can use the XPath describe the path to nodes in YANG.

1.1.11 Acceptable Data Formats

Until now only data model has been discussed. However, it is also important to mention the formats of the data that match the model. Originally, only the XML encoding of the data was supported. Later on, the JSON encoding was also introduced in RFC 7951 [8].

1.2 sysrepo

To improve the NEMEA's configuration model, the sysrepo open source project was chosen as an alternative to the existing solution of NEMEA Supervisor (storage of plain XML files merged at runtime). At first sight, it may seem that sysrepo works in a similar manner since it also accepts the XML format as an input. However, one should not be mistaken by that as it adds many more features than those that are currently implemented in NEMEA. It's due

[7]The terms operational data, state data and runtime statistics are used in this thesis interchangeably.

to a fact that sysrepo is a datastore for the YANG-modeled configuration and state data. Therefore, each value we load into sysrepo is validated against a predefined YANG model.

1.2.1 Integration

There are two ways of integrating sysrepo. It can be done either by using its C shared library or using bindings for other languages (C++, Python, ...) that are provided by SWIG (Simplified Wrapper and Interface Generator) [8].

Applications using sysrepo first need to connect and then create a session. Each application using the sysrepo library internally creates its own sysrepo engine for accepting its own connections. However, we can also create a single engine shared by all the applications in the system by running a standalone sysrepo daemon, which saves system resources.

1.2.2 Obtaining Values

Configuration values can be queried and set using XPath. With the runtime statistics, it is a bit more complicated as those are not known by sysrepo and can only be set by the application providing them. Sysrepo works as a proxy in the following manner: The application A subscribes to requests for specific statistics data in the YANG model by registering its callback function. When another application (an application B for this purpose) requests the data from sysrepo, sysrepo calls a registered function of A and passes on the data to B. Passing the data through sysrepo ensures that B gets the data in the format defined in YANG. The whole process, of course, only works if A's callback function is subscribed and thus A is running.

1.2.3 Types of Datastores

Datastore in the context of sysrepo is a repository where data can be stored. A single YANG model installed in sysrepo can have up to three datastores, each of a different type. Those types are as follows:

- **Startup**: Provides configuration data that should be available to an application at startup.
- **Running**: Is created at the time an application subscribes for state data requests or configuration data changes. In the case of no subscription, it behaves as if there were no data. Data in this datastore are removed once all subscriptions are dropped.
- **Candidate**: Exists as an ephemeral playground where changes are validated only and not propagated to any other datastore. There is a unique datastore for each sysrepo session.

[8]`http://www.swig.org/`

1.2.4 Tools

Sysrepo comes with two tools to manage our installation:

- **sysrepoctl**: Allows adding/removing/modifying YANG modules.
- **sysrepocfg**: Manages data saved in sysrepo datastores.

1.3 NETCONF

NETCONF is a protocol that was defined in RFC 4741 [9] and later redefined in RFC 6241 [10]. The text below is focused on the more recent RFC version that describes NETCONF as follows: "The NETCONF protocol defines a simple mechanism through which a network device can be managed, configuration data information can be retrieved, and new configuration data can be uploaded and manipulated.". I describe it here as an option that can be added on top of sysrepo and used for improved configuration of NEMEA.

1.3.1 Layers

The NETCONF protocol is comprised of four layers:

1. **Secure transport layer**: Takes care of transport of messages between a server (configured device) and a client (administrator). As long as conditions laid down in RFC 4741 [9] are observed, any transport protocol can be used. However, there also exist particular RFCs containing usage of NETCONF over SSH (in RFC 4742 [11] and 6242 [12]. SSH is a protocol that any implementation must be able to manage. Other transport protocols are for example SOAP (in RFC 4743 [13]) or TLS (in RFC 5539 [14] and 7589 [15]).
2. **Messages layer**: Serves as a wrapper layer for encoding of remote procedure calls (RPC) and notifications (messages sent from the server without the client asking).
3. **Operations layer**: RFC 6241 [10] states it clearly: "The Operations layer defines a set of base protocol operations as RPC methods with XML-encoded parameters."
4. **Content layer**: Represents configuration data and information about a state of a device.

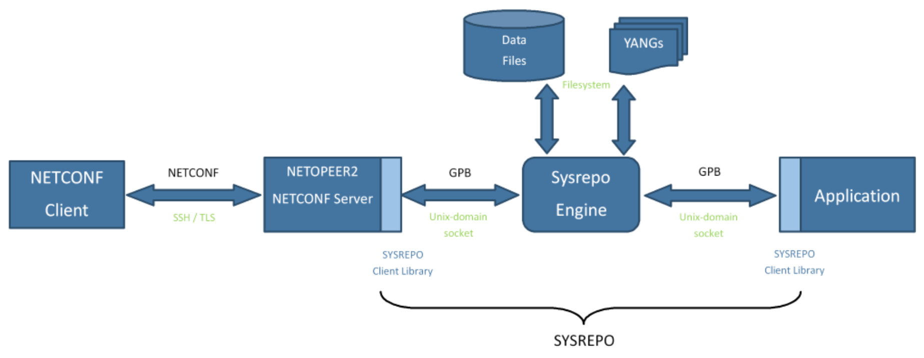


Figure 1.3: Usage schema of NETCONF, Netopeer2, sysrepo and YANG for an application. GPB is protocol Google Protocol Buffers used by sysrepo in socket communication. (Source: sysrepo project homepage, `https://www.sysrepo.org/Sysrepo?action=AttachFile&do=get&target=high_level_architecture.png`, downloaded at 2018-05-04)

1.3.2 Netopeer Project

In the NETCONF Wiki by IETF[9] there is a list of the NETCONF protocol implementations. There are not many of them and in case we require support of an open source configuration datastore that supports YANG, we end up at the Netopeer project by the CESNET association.

Netopeer is an implementation of the NETCONF client and server based on libnetconf. In this version of Netopeer, transAPI [10] was used to access a configuration. Aside from a CLI client, also a GUI client existed that was created within a diploma thesis written by David Alexa [16].

Recently, the second generation of Netopeer — Netopeer2 — was developed. The version contains the client and server based on libnetconf2. The library in Netopeer2 does not directly implement datastore anymore but uses sysrepo instead. For the new generation of the whole project, even the new NETCONF client user interface called Netopeer2GUI was created. The Netopeer2GUI also uses the current version of the library.

To get a better understanding of terms like NETCONF, Netopeer, sysrepo, YANG, configurated application, and their cooperation in practice, the figure 1.3 can be used. It is depicting Netopeer2GUI[11] only as the NETCONF client.

[9] `https://trac.ietf.org/trac/netconf`

[10] `https://rawgit.com/CESNET/libnetconf/master/doc/doxygen/html/d9/d25/transapi.html`

[11] `https://github.com/CESNET/Netopeer2GUI`

1.3.3 Netopeer2GUI

This web application is a module for Liberouter GUI and serves to manage multiple NETCONF enabled devices that can be any NETCONF server.

At the time Netopeer2GUI was analyzed (December 2017), it supported only basic operations such as connecting to NETCONF servers, managing YANG models and displaying configuration data and state data.

1.4 Security Tools as a Service (STaaS)

The name stands for an environment in which the whole NEMEA system is used. It is a single ecosystem for network monitoring and analysis from the CESNET association and it is available in a form of Ansible[12] playbook. The playbook is a set of configuration files and rules written in the YAML[13] format. Those together help automate provisioning and deploying of systems on hosts. In the case of STaaS, it allows one to easily deploy the whole environment on a specific host. A simple option for getting such host is using the included configuration file for Vagrant[14] which sets up a new virtual machine and applies the Ansible playbook of STaaS.

The current version of STaaS contains the following software relevant for this work:

- NEMEA system, including the NEMEA Supervisor, set of its modules and IPFIXcol as a data source provider (already discussed in section 1.1)
- STaaS GUI as part of Liberouter GUI (discussed below)
- NEMEA Dashboard and NEMEA status (discussed below)

1.4.1 Liberouter GUI (lgui)

Since the CESNET organization has many projects needing a graphical user interface, lgui was designed to provide a single environment that allows an easy integration of other new GUIs primarily created in CESNET. The advantage of this solution is a prepared design in the frontend and a session handling with authorization in the backend.

The frontend of lgui is written in the Angular framework (described later in section 2.3.3.1) and backend web API is written in the Flask microframework for Python (described later in section 2.3.4).

Just as the NEMEA system, lgui is also modular and allows to have multiple modules integrated with it. A single module consists of frontend and

[12]`https://www.ansible.com/`
[13]`http://yaml.org/`
[14]`https://www.vagrantup.com/`

backend parts. Details on the lgui development can be found on the official wiki page on the project's Github[15].

1.4.2 STaaS GUI

Components contained in STaaS have their GUIs developed as modules of Liberouter GUI. The most important module for this thesis project is NEMEA Dashboard, for it was used as a foundation for NEMEA GUI lgui module created in this thesis. The complete list of modules that STaaS GUI provides can be found in the official GitHub repository[16].

1.4.3 NEMEA Dashboard

The NEMEA Dashboard is the only existing GUI for NEMEA. There are two versions of it in its GitHub repository[17]. The first one can be run without Liberouter GUI and the second one is its module. As the former is the older version of the NEMEA Dashboard, only the latter that is used throughout this thesis will be discussed. There are currently four parts to it:

- **Dashboard**: Displays detected security events in graphs.
- **Events**: Displays detected security events or tables.
- **Status**: Displays currently running NEMEA modules with statistics from their interfaces and system resources usage. It takes the information about modules from NEMEA Supervisor client.
- **Reporters Configuration**: Allows configuration of NEMEA reporters.

[15]`https://github.com/CESNET/liberouter-gui/wiki`
[16]`https://github.com/CESNET/STaaS-GUI`
[17]`https://github.com/CESNET/Nemea-Dashboard/tree/liberouter-gui`

Chapter 2

Analysis and Design

NEMEA is a modular system, which is a convenient attribute when we want to use its individual parts separately. However, it stops being so convenient when we want to configure the entire system at once. A solution to this problem already exists in the form of the NEMEA Supervisor that enables such configuration. So if we want to use sysrepo for saving the configuration of the NEMEA system within STaaS, just adding the support of sysrepo into the Supervisor will do. The thesis project is based on this approach, as it is the most straightforward way to deal with the issue. After that, just building GUI on top of sysrepo used by the new version of Supervisor remains to be done to make the user experience of the administrator using NEMEA more pleasant.

This chapter first introduces the YANG model which will be used by the new NEMEA Supervisor. Due to the fact that the results of this thesis should be applied in practice, I describe the designed data model more extensively, so the genesis of the whole solution would be captured for possible future work. Afterward, a Supervisor daemon is described including the reasons leading to the decision to rewrite it from the beginning. The text explains how it differs from the first version and defines its required functionality. After that, the idea of a suitable graphical user interface of the entire solution, built on top of sysrepo, is presented. Finally, the chapter introduces the Docker environment that provides easier accessibility of the entire solution for testing purposes due to the fact that the user does not have to install all the dependencies manually.

2.1 YANG Data Model for NEMEA

In the NEMEA Supervisor repository at GitHub.com, we can notice that there have been certain attempts at modeling the NEMEA configuration in YANG[18]. The solution used libnetconf and as of today, it is not working. To

[18] `https://github.com/CESNET/Nemea-Supervisor/blob/master/ncnemea/nemea.yang`

include the characteristics of the Supervisor created in this thesis, I designed a new data model.

Since the original version of the Supervisor is already configured using XML and data from the XML files can be loaded to sysrepo, a solution suggests itself to design a data model in which the configuration data from the current version of the Supervisor would fit. I consulted the draft of the model with the members of NEMEA team including my supervisor, who have the biggest experiences with NEMEA and we came to the decision that the model should be simplified.

In the previous chapter, I have described that the NEMEA Supervisor calls individual programs it controls modules. The modules then fall into groups called module profiles that are formed according to their purpose. Module profiles served primarily for starting/stopping/restarting a group of modules and that turned out to be a functionality that is seldom used. It is therefore needless to include a grouping of modules directly in the data model when it can be easily and more suitably implemented on a different layer — in the user interface. This way, the user can select several modules and pick an action for all of them simultaneously.

Another change was made for the sake of the NEMEA modules, such as for instance *logger*[19], that can be executed several times with various parameters. In such case, the administrator must create several `<module>` elements in XML with the same path to executable files and different startup parameters. To spare the user a necessity of entering the same path twice, I have added one more object into the existing model — *instance*, which should also decrease the risk of making mistake during repeated entering of data. The *instance* object is one specific instance of a started module (a system process). Multiple instances can exist depending on which parameters is the module started with.

Now we have two basic lists of elements: `<available-module>`[20], which is the list of modules that are able to be started, and `<instance>` as a list of instances of such modules. Every instance is linked to a corresponding available module using *leafref* YANG statement.

I describe this model in listing 2.1. Although the extent of the description is larger, I still leave out some specific details, such as for example the restrictions for values of the particular *leaf* elements in order to be brief. The whole YANG model is naturally accessible in folder `src/nemea-supervisor/yang/` on the enclosed CD.

[19]`https://github.com/CESNET/Nemea-Modules/tree/c84a0c3e1e6300e9491bdd7f0a221fd30a113a93/logger`

[20]This element was renamed to prevent confusion, since in the original configuration of Supervisor, `<module>` represented an instance element.

```
module nemea {
  yang-version 1.1;
  namespace "urn:ietf:params:xml:ns:yang:nemea";
  prefix nemea;
  include trap-interfaces;

  container supervisor {
    list available-module {
      key name;
      uses nemea-key-name;

      leaf path { type string; mandatory true; }
      leaf description { type string; mandatory true; }
      leaf trap-monitorable { type boolean; mandatory true; }
      leaf trap-ifces-cli { type boolean; mandatory true; }
      leaf is-sysrepo-ready { type boolean; mandatory true; }
      choice if-is-sysrepo-ready {
        when "is-sysrepo-ready = 'true'";
        leaf sr-model-prefix { type string; mandatory true; }
      } // end choice if-is-sysrepo-ready
      choice if-trap-ifces {
        case yes {
          when "trap-monitorable = 'true' or trap-ifces-cli = 'true'";
          leaf in-ifces-cnt { type string; mandatory true; }
          leaf out-ifces-cnt { type string; mandatory true; }
        }
      } // end choice if-trap-ifces
    } // end list available-module

    list instance {
      key name;
      uses nemea-key-name;
      uses trap-ifcs-list; // included from trap-interfaces module
      uses nemea-instance-stats;

      leaf module-ref {
        type leafref { path "../../available-module/name"; }
        mandatory true;
      }
      leaf enabled { type boolean; }
      leaf max-restarts-per-min { type uint8; default 3; }
      leaf last-pid { type uint32; }

      choice if-is-sysrepo-ready {
        when "deref(.)/../is-sysrepo-ready = true";
        leaf use-sysrepo { type boolean; default false; }
      }
      choice if-isnt-sysrepo-ready-or-doesnt-use-sysrepo {
        when "deref(.)/../is-sysrepo-ready = false or \
                   use-sysrepo = false";
        leaf params { type string; }
      } // end choice is-module-sysrepo-ready
    } // end of list module
  } // end container nemea-supervisor

  grouping nemea-key-name {
    leaf name { type string; mandatory true; }
  }
  grouping nemea-instance-stats {
    container stats {
      config false;
      ...
    }
  } // end grouping nemea-instance-stats
} // end module nemea
```

Listing 2.1: YANG model for NEMEA.

Now, I would like to examine individual parts of the data model according to the listing above. We can skip the introductory “header” on the first four lines, because it contains only a definition of the YANG module. On the fifth line there is an inserted submodule *trap-interfaces* describing the interfaces of the libtrap library in independent YANG file and I will focus on it later. After that, the *typedef* statement follows that works similarly as it would for instance in a C language. By that I define a new data type for entering a UNIX file system path where I set restrictions (those are left out of the listing) for the path.

Two *grouping* statements on the lines 56-64 are just reusable blocks containing other YANG statements. First *grouping* statement — *nemea-key-name* — allows us to define a *leaf* node, which serves as a key for locating all lists used throughout our model in one place, where we can define restrictions common for all keys. The second grouping statement — *nemea-module-stats* — defines container stats including runtime statistics of an instance (memory and CPU usage). I left out the leaf nodes for brevity in listing but we can see the *config* statement with a value of *false* telling us that this container is only available at runtime because its values are not configuration but state data. I would like to add that this second grouping element exists only so the list of instances would be more legible, *nemea-instance-stats* is not used anywhere else.

The model itself begins with the statement *container* on the line 7 that is only a wrapper element to contain the Supervisor configuration that includes our two lists with all available modules and instances.

2.1.1 Available Modules

In this section, I would like to go through configuration options contained in the *available-module* element. For now, I am going to leave out the elements related to sysrepo. The options are as follows:

- *name* string leaf: Unique name of a module inserted using *nemea-key-name.*
- *path* string leaf: Full path to an executable file to start a module.
- *description* string leaf: Brief description of a module that will be displayed to the end user in GUI. It is a plan for the future that this description should be distributed along with NEMEA modules. According to this plan, there will not be a need for the administrator to enter the description manually — he will be able to gain it for example from an installation package or from a git repository of a module.
- *trap-monitorable* boolean flag: Determines whether a given module uses interfaces of the libtrap library in some unspecified manner and also

whether we want to connect to its service interface to gather runtime statistics about the interfaces. This flag is incorporated here mainly because of IPFIXcol which uses interfaces but defines their parameters in a configuration file and not through a command line as the rest of the NEMEA modules do.

- *trap-ifces-cli* boolean flag: Compared to the previous one, this flag determines whether a module supports the configuration of libtrap interfaces and so whether this configuration is supposed to be passed to it using CLI parameters in a standard format described in [4].

- *if-trap-ifces* condition: In here, I use the statement *choice* for wrapping the elements that are only supposed to exist in the model conditionally. Just as in other elements, a condition in *choice* is entered by the statement when. Condition tells us here, that it is necessary either to want to pass the interfaces' parameters over CLI while starting a module or to monitor statistics about interfaces on account of the fact that they were configured in a different way. When meeting conditions, other two elements determining a number of input and output interfaces are added. This number can be expressed by a specific number that defines an exact number of interfaces that must be set, or by an asterisk for a variable number of interfaces. This setting should mainly be used by GUI to watch for the correct count of configured interfaces.

2.1.2 Instances

Particular instances are defined in the model by the list instance. In the following text, I am going to skip the statements *trap-ifcs-list* and *nemea-instance-stats* — the former is going to be described later and the latter contains no important information. The rest of the elements is as follows:

- *name* string leaf: Unique name of an instance inserted by *nemea-key-name.*

- *trap-ifcs-list* list: Inserted list of interfaces, is examined below in section 2.1.3.

- *module-ref* reference: Refers to *name* in the list *available-modules* and by such reference it forms a relation where the particular instance belongs to a module.

- *enabled* boolean flag: Determines if the Supervisor is supposed to attempt to start an instance or not.

- *max-restarts-per-min* leaf: Enables to set a number of attempts to restart an instance in case that the instance is enabled but not running. If the number of attempts per minute exceeds the set value, the Supervisor tries to restart it no more.

- *last-pid* leaf: This value serves the Supervisor for saving a UNIX process identifier (PID) of the instance before it turns itself off. Knowledge of the PID allows the Supervisor to find out which processes are already running and therefore does not attempt to launch them again.
- *use-sysrepo* boolean flag: If the module of an instance supports sysrepo (see section 2.1.4 for more), this flag determines whether to really use a configuration provided by sysrepo or whether to use standard arguments passed through the command line.
- *params* string leaf: In case sysrepo is not used, it is possible to use leaf *params* as a place where CLI parameters for an instance should be placed, e.g., "`-v 20 -r`".

2.1.3 Interfaces

For more clarity, the interfaces of libtrap are situated in a separate submodule which is included in the listing 2.2. This is not, in fact, a particularly innovative solution but just modelling of the libtrap specification ([4]).

A little problem here caused by YANG is a necessity of using the *key* statement in the list of interfaces. From the NEMEA's point of view it is not required, because each interface is uniquely identified by its direction and position in the list. YANG, however, requires us to specify a unique key which can be created by a combination of several *leaf* nodes. Nevertheless, we cannot use for example the combination of type and direction parameters because there might exist two interfaces with the same combination. This inconvenience can be diverted from the user at least a bit by pre-generating the names of interfaces in GUI.

```
submodule trap-interfaces {
  yang-version 1.1;
  belongs-to nemea { prefix nemea; }
  import ietf-inet-types { prefix inet; }
  typedef trap-ifc-type { type enumeration { ... } }
  typedef trap-ifc-dir { type enumeration { ... } }

  grouping trap-ifcs-list {

    list interface {
      key name;
      uses nemea:nemea-key-name;

      leaf type { type trap-ifc-type; }
      leaf direction { type trap-ifc-dir; }
      leaf timeout { type string; }
      leaf buffer { ... }
      leaf autoflush { ... }
      choice type-params {
        case tcp-params { container tcp-params { ... } }
        case tcp-tls-params { container tcp-tls-params { ... } }
        case unix-params { container unix-params { ... } }
        case file-params { container file-params { ... } }
        case blackhole-params { /* Blackhole has no parameters */ }
      } // end choice type-params
      container stats { config false; ... }
    } // end list interface
  } // end grouping trap-ifcs-list
  ...
} // end submodule trap-interfaces
```

Listing 2.2: YANG submodule defining configuration of interfaces for NEMEA Supervisor module instances.

2.1.4 Custom YANG Model for Modules Using sysrepo

The main goal of this thesis is to configure the Supervisor using sysrepo and by that draft a path for further possible extension of using sysrepo in NEMEA. At present, a primary concern is so that the Supervisor would be able to receive CLI parameters passed to the modules from sysrepo. In this chapter, I would like to demonstrate that using sysrepo on a larger scale was kept in mind, the model takes it into account and that there is no need to redo the solution worked out in this thesis, only to extend it alternatively (if need be).

Most of the NEMEA modules that are to be currently found in STaaS are configured just by using CLI parameters. Some of them use also configuration files in various formats: modules reporting in IDEA format[21] use YAML[22], IPFIXcol uses XML and for example *link_traffic* module[23] uses own plaintext format. Since I am going to use sysrepo for the Supervisor, it suggests itself that the NEMEA modules should have the same option. If the modules were unified to use sysrepo instead of their own configuration files and CLI arguments, the configuration of the entire complicated system would be simplified

[21]http://nemea.liberouter.org/reporting/
[22]http://yaml.org/
[23]https://github.com/CESNET/Nemea-Modules/tree/master/link_traffic

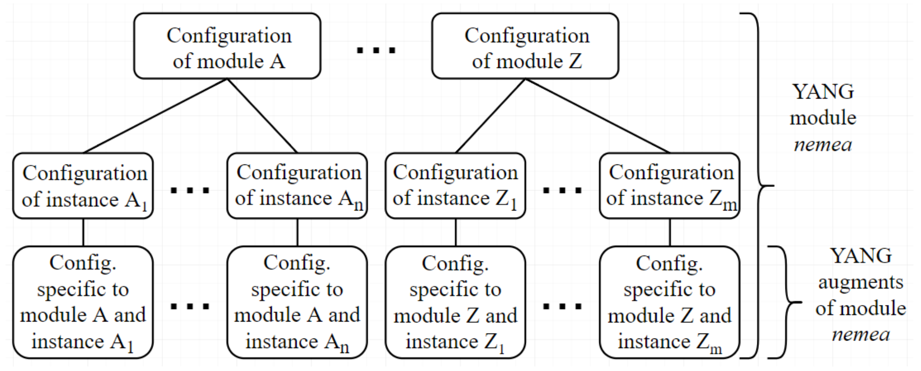

Configuration of module A
...
Configuration of module Z
YANG module *nemea*
Configuration of instance A_1
Configuration of instance A_n
Configuration of instance Z_1
Configuration of instance Z_m
Config. specific to module A and instance A_1
Config. specific to module A and instance A_n
Config. specific to module Z and instance Z_1
Config. specific to module Z and instance Z_m
YANG augments of module *nemea*

Figure 2.1: First version of YANG model for NEMEA modules using sysrepo (augmented *nemea* module)

once again. This way we would have one separate machine processable configuration model in YANG for each module. The concept will also help us in a way which would allow us to control valid configuration values of modules to a certain extent. It is because in the NEMEA model the modules that are not using sysrepo only receive CLI arguments which are not validated by that model. On the other hand, the modules using sysrepo might design their YANG module to validate their configuration data.

I designed two ways of linking the configuration model of Supervisor to the other module models. Version 1 at the figure 2.1 uses YANG feature for augmenting the modules and version 2 at the figure 2.2 allows the modules to use their own separate YANG modules, but requires to use some kind of a YANG template to which the Supervisor understands.

2.1.4.1 Version 1

YANG language enables to augment the modules by using the augment statement in some other YANG module. By that, new nodes are inserted into the existing model. If we then want to describe data, that are defined through augmenting, we add the XML namespace[24] to the elements, which is named after a prefix of augmenting module, e.g., `<prefix-of-yang-module-that-adds-augment:added-node />`. Disadvantages of this version are as follows:

- Configuration of each module cannot be saved in sysrepo separately. The model of the Supervisor will need to be installed every time, even if there would be no actual need for him. That somewhat contradicts the NEMEA's principles of modularity where each module can be run independently.

[24]`https://www.w3schools.com/xml/xml_namespaces.asp`

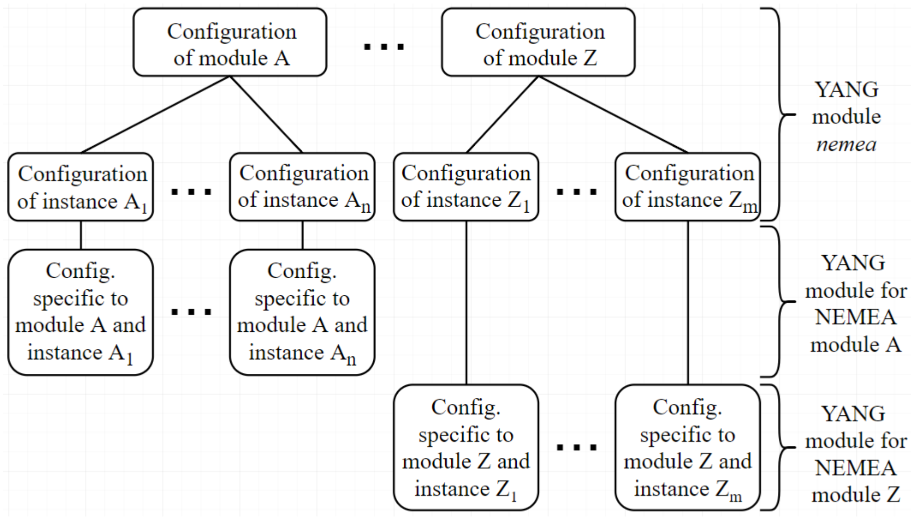

Configuration of module A
Configuration of module Z
Configuration of instance A1
Configuration of instance An
Configuration of instance Z1
Configuration of instance Zm
Config. specific to module A and instance A1
Config. specific to module A and instance An
Config. specific to module Z and instance Z1
Config. specific to module Z and instance Zm
YANG module nemea
YANG module for NEMEA module A
YANG module for NEMEA module Z

Figure 2.2: Final version of YANG model for NEMEA modules using sysrepo (own YANG module for each NEMEA module using sysrepo)

- A common model for all modules can also present a problem in the case that a new revision of the Supervisor model, in which each module supports different version, will be created. The modules can also unintentionally erase the configuration to each other.
- To manage lots of various namespaces in one XML file might not be exactly convenient.

2.1.4.2 Version 2

This version at the figure 2.2 lets the developer of a NEMEA module create the YANG model more or less his own way. There is a catch in it, however, that lies in a fact that a NEMEA module can run multiple times with different configuration and the Supervisor must somehow let the NEMEA module know, which configuration it should pick while starting. For that reason, there is a need for linking the instances in the model of the Supervisor to the configuration of instances in the NEMEA module's model. I would like to achieve such linking by forcing the developers of NEMEA module to use a template (listing 2.3) when creating their YANG model. In the listing, we can see that there as well exists the list of instances that are identifiable by their name and those names should be exactly the same as the ones that exist in the YANG module *nemea* — that is all that the template requires. Not using the YANG reference through *leafref* is on purpose here, so that the model can exist independently on whether the model of the Supervisor is installed in the system. Now I would like to mention a few disadvantages of this model:

- Due to not using *leafref* as a reference to the model of the Supervisor, a situation might occur that a name of an instance which does not exist in the Supervisor's setup will be set here. Fortunately, this is an imperfection that could be solved on another layer, for example in GUI.
- An instance with a name that does not exist in the list of instances of the Supervisor can be created in a NEMEA module as well. This is also solvable on a different layer.

```
module sysrepo-ready-module {
  namespace "urn:ietf:xml:ns:yang:sysrepo-ready-module";
  prefix sysrepo-ready-module;

  list instance {
    key name;
    leaf name { type string; }
    /* insert YANG nodes specific to sysrepo-ready-module here */
  }
} // end module sysrepo-ready-module
```

Listing 2.3: YANG template to be used by NEMEA modules using sysrepo.

2.1.4.3 Choosing the Version that Fits Better

Both versions of the implementation of custom configuration model of NEMEA modules were consulted again with the members of the NEMEA team and version 2 of the model was chosen mainly because it places smaller demands on the developers and does not force them to use the model of the Supervisor.

2.1.4.4 Elements Related to sysrepo Configuration

Now, when I have already explained how the NEMEA modules using sysrepo are supported and how this support was created, I would like to pay attention to the elements of the model of the Supervisor that are related to sysrepo and whose description was left out earlier because now they should make more sense. At first, I am going to start with the items on the list available modules:

- *is-sysrepo-ready* boolean flag: Specifies whether a module supports sysrepo and so whether it supports all the prerequisites mentioned elsewhere in section 2.1.4.6.
- *sr-model-prefix* string leaf: Defines a prefix of the YANG model that is used by a NEMEA module. This way the Supervisor and its GUI know, where to find a configuration. For example, if the prefix is *link-traffic*, we know that we should use XPath `/link-traffic:instance[name='instance-name']` for access to an instance *instance-name* of a NEMEA module.

For individual items on the list of instances there could be also set:

- *use-sysrepo* boolean flag: Even though a module defining an instance might support sysrepo, it might not necessarily use it. By this flag, we can determine whether to use a standard configuration by passing CLI parameters after all or whether a module is supposed to look the configuration up in sysrepo.

2.1.4.5 Interfaces

While reading the foregoing lines of the listing we can notice, that one particular issue has not been dealt with in the new model — interfaces. Those are still modeled only in the YANG module *nemea*. The reason for that is that the NEMEA modules usually use a macro `TRAP_DEFAULT_INITIALIZATION` of library libtrap that parses for them interfaces parameters passed on the command line by `-i` argument. Instead of forcing the developers of modules to complete an implementation of logic for loading interfaces from sysrepo, I would like to let them use the older type of logic. At some time in the future, a new form of the macro that would already support sysrepo might be created.

2.1.4.6 Summary of What Module Using sysrepo Needs to Implement

There are quite a few prerequisites that are needed to be fulfilled so that the NEMEA Supervisor would be able to configure a module using sysrepo correctly. For more clarity I would like to summarize them in the following four points:

1. The module's YANG model must implement the list of instances that are identifiable by name and the real configuration model should be included in that list (see the template in listing 2.3).

2. A module must understand the CLI argument `-x`, the parameter of which is a name of the instance and is used for the identification of the process that received this parameter. In other words, a module uses a parameter of argument `-x` for searching for the right place in the list inside its configuration and uses the configuration only for an instance with this name.

3. When starting, a module must load its configuration from a startup datastore under an instance, in which it runs at that moment.

4. A module must react to changes in a running datastore under an instance in which it runs at that moment.

2.1.4.7 Reference Module Using sysrepo

So that everything aforementioned about the modules using sysrepo would not be just an empty talk, I adjusted the module *link_traffic* as a reference module that fulfills the prerequisites enumerated in the previous section. I did not have to start from the very beginning, my solution was based on the already existing adjustment enabling the use of sysrepo by Jaroslav Hlaváč that can be found on GitHub[25].

2.2 Supervisor Daemon

Since there are already experiences with the application of Supervisor daemon in the NEMEA team and its use is quite convenient, together with my supervisor we made a decision to preserve it. Rather than try to combine two different approaches together, we agreed to create a new version from scratch but at the same time to use the code from the original version where possible. This means I implement the new version in C language.

In the original version of the Supervisor, most of the functionality was implemented in a single file comprised of nearly six thousand lines. In the new version, I would like to divide functionality into multiple files — modules. The previous version did also not contain any tests and that is why I am going to create the unit tests[26], at least where it is useful due to the creation of new functions.

Now we can look closer at the new version of the Supervisor developed in this thesis. The most demonstrative way to approach the description would be to compare the functionalities of both versions of the Supervisor, so we can also see how the Supervisor has changed.

2.2.1 Summary of Main Functionality of the Original Supervisor

At first, I provide a reader with this summary of the original Supervisor's functions:

1. Load a startup configuration of module profiles and modules.
2. Start configured modules at the Supervisor's startup.
3. Provide means for runtime management through the Supervisor client. This includes functionality to:
    a) Start (enable) or stop (disable) a configured module or profile at runtime.

[25]`https://github.com/CESNET/Nemea-Modules/tree/sysrepo/link_traffic`

[26]Unit testing is a process of testing whether individual units (in case of this thesis mostly functions) work as expected.

b) Restart a configured module at runtime.
c) Provide system usage statistics of running modules.
d) Provide libtrap interface statistics of running modules.
e) Reload whole configuration from a configuration template file defined through a Supervisor's startup CLI argument.
f) Display a currently applied configuration.
g) Provide an information about currently running Supervisor instance.

2.2.2 What Is Different in the New Supervisor Version - Functional Requirements

In the following text, I would like to describe the changes made to the new version compared to the original version. Those changes are stated in the order that corresponds with the order of the foregoing list. Most of the items in the list also represent functional requirements. To note this fact, I am going to mark such items with **F** and a matching order number.

1. (**F1**) Load a startup configuration of modules and their instances from sysrepo (profiles were removed).

2. (**F2**) Start instances of configured modules at the Supervisor's startup (this function differs from the original version in the fact that the modules alone were divided into the available modules and instances).

3. There are two options for runtime management of the Supervisor in the new version — using web API/GUI, which will be described later, or using sysrepo and its tools. The new version will contain these following options for runtime management or adjustments compared to the original version:

   a) (**F3**) Start (enable) or stop (disable) a configured module, which means all of its instances or just specific running instance through sysrepo.
   b) Restart of an instance or module should be currently possible only by its disabling and consequently enabling. The Supervisor does not natively support this function.
   c) (**F4**) Provide system usage statistics of running modules through sysrepo.
   d) (**F5**) Provide libtrap interface statistics of running modules through sysrepo.
   e) (**F6**) Ability to change a runtime configuration will be provided through the sysrepo's running datastore. The Supervisor reacts to those changes while running and according to what has changed, it restarts just the modules or instances involved.

f) The Supervisor client is not going to be implemented. The applied setup can be printed/displayed through sysrepo or in web API/GUI.

2.3 Graphical User Interface

In the first chapter, I have described Netopeer2GUI as an user environment for the management of NETCONF enabled applications. By using sysrepo, the Supervisor turns into such kind of application and so the use of Netopeer2GUI is possible. That gives me two options: either to help with the development of that user environment and then use it for management of the NEMEA Supervisor, or to design my own solution that will not be dependent on NETCONF.

In the first chapter, I have also already touched on the fact that Netopeer2GUI supported only the basic functionality at the time the analysis was being made and the suitable solution was being chosen. Netopeer2GUI is also aimed at managing multiple NETCONF devices, while in our case we need to typically configure just a single server where STaaS is running. GUI created in this thesis should be tailored to the needs of NEMEA most of all, so we can make the system accessible to the new users by providing them with a good user experience. It is evident that general solution for the management of multiple devices cannot be — in terms of user experience — compared to the solution designed for a specific use case. Although Netopeer2GUI is planned to be extended for the specific scheme through the use of plugins according to its GitHub page [17], this characteristic has not been implemented yet. Finally, I would like to add that along with the use of NEtopeer2GUI, the Netopeer server would come into play as another software dependency which is supposed to be maintained in STaaS.

For the aforementioned reasons and also on account of the fact that I spent a larger amount of time than expected working on the Supervisor's implementation, I decided to choose the simpler solution and not contribute to Netopeer2GUI development. Instead of that, I have designed a brand new user interface constructed in Liberouter GUI whose backend communicates with sysrepo directly with no need for involving NETCONF. I implement new GUI inside the existing NEMEA Dashboard lgui module so that NEMEA GUIs are not shattered. Because NEMEA Dashboard including my solution contains more than a single dashboard I call this connection NEMEA GUI and the specific GUI part developed in this thesis NEMEA Supervisor GUI.

In the rest of this section I am going to focus on the requirements for GUI functions, the actual design of the GUI and description of technologies determined by lgui.

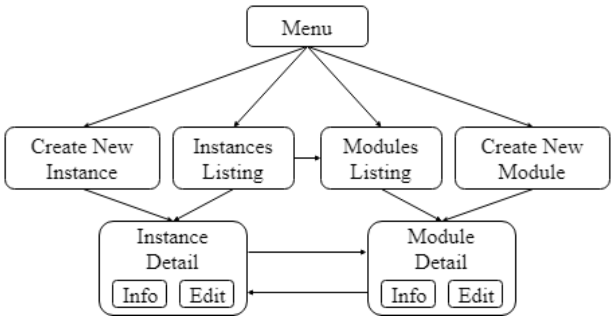


Figure 2.3: Visual representation of the sitemap for the NEMEA Supervisor GUI.

2.3.1 Functional Requirements

Functions common to the API and the graphical web user interface (frontend):

1. Option to create a module or an instance.
2. Option to adjust a configuration of a module or an instance.
3. Option to display a configuration of a module or an instance.
4. Option to display a current system resources usage of an instance.

Functions exclusive for the API:

1. Runtime statistics of libtrap interfaces.

2.3.2 GUI Design

In figure 2.3 we can see a design of sitemap for the web user interface that depicts the individual websites and the options of passing among them. For this sitemap, website wireframes were created that we can find in the appendix B. The wireframes were hosted for review by the NEMEA team members in a service InVision[27]. Such service enables to upgrade static wireframes by selecting clickable spots that take the user to another website. By clicking on them, a correct wireframe picture displays. This way we get a simple prototype without any labor lost.

2.3.3 Frontend Technologies in Liberouter GUI

Liberouter GUI is based on a concept of single page application (SPA) using Angular. That essentially means that if we perform any action in a web application, an already loaded website will not be reloaded in a browser (the

[27] `https://www.invisionapp.com/`

browser will not request new HTML). If needed, the browser will only request data by using XMLHttpRequest[28]. JavaScript loaded in the browser then redraws only that part of the website that should be really changed. This is achieved by the fact, that when we visit SPA, we download an entire bundle of all the necessary JavaScript, and possibly HTML, which are later used for rendering of all parts of an application. Thanks to that, the user is provided with a better experience, because the amount of time spent on redrawing the part of a site is typically not as large as when requesting and rendering the whole site — this way we get closer to the speed of classic desktop applications.

The bundles mentioned above are generated on the side of a server from many other technologies. The point is, that for the web development there are no longer used only classic frontend technologies like HTML, CSS, and JavaScript, which a majority of browsers understands. Instead, we can see, that the other various languages and template systems, that are built on top of them, are used more often. So that the browsers would be able to understand them, it is necessary to transpile a code before it is delivered to the browser. To transpile means to go through a process, that transfers the code from one representation to another while preserving a functionality. This process is mostly mediated by transpilers running in node.js[29] which is a desktop runtime environment for JavaScript. Owing to this process, the modern technologies, I briefly describe in the following text, could be used in lgui.

2.3.3.1 Angular 4

Angular is a SPA framework developed by Google. It uses Typescript[30] that is a superset of JavaScript developed by Microsoft in which the main mentionable differences are static typing (which quality can be already deduced from its name) and support for classes making the object-oriented programming in JavaScript easier.

In Angular, there exists a concept of modules, that correspond to the lgui modules, or to their frontend part to be more precise. A module is, however, only an encapsulation of one single workflow or of other enclosed part of a website. It consists of components that we can imagine as a single view. In wiki of Liberouter GUI[31] it is recommended to use the components comprised of three files. For instance, the component `inst-detail` consists of:

- `inst-detail.component.html`: Describes HTML of the component and by a special template language of Angular determines conditions defining when and how data from a `.ts` file (see below) should be displayed. It also, for example, determines events that should be handled and using which functions should they be handled by.

[28] `https://developer.mozilla.org/en-US/docs/Web/API/XMLHttpRequest`
[29] `https://nodejs.org/en/`
[30] `https://www.typescriptlang.org/`
[31] `https://github.com/CESNET/liberouter-gui/wiki`

- `inst-detail.component.ts`: This file defines a class of this component and inside it contains functions for processing events in a `.html` file and also data that are supposed to be displayed. Such data might be gained from other components or from another special building blocks of Angular — *services*. Those are classes from which we can provide functions or data shared among the components. Their most common use is nevertheless to define XMLHttpRequests for gaining data from a backend.

- `sup-gui.component.scss`: The last file serves for defining styles for this component. In lgui there is used an augmented version of classic CSS called SASS[32].

2.3.4 Backend Technologies in Liberouter GUI

As it has already been said, the server side of lgui uses Python language with the Flask framework. This framework calls itself "micro" because it contains only a web server gateway interface (WSGI)[33] and a template engine. It does not contain, for example, any facilitation of communication with databases through an object-relational mapping (ORM) which is a feature typical for web backend frameworks. In case that we need to create a simple API (which is exactly our case), this framework is going to be sufficient.

In lgui it is used in a way, that backend of the lgui module will hook its functions to a combination of URL paths and HTTP methods. After receiving a request with the fitting path and method combination, those functions are called.

Although the Flask is WSGI, so we can run it on the command line without anything else and it would work, for a production deployment, however, it is better to use it behind some kind of web server such as for example Apache[34].

Using the Flask we want to create an API which will then communicate with sysrepo. For this job we have three options in Python:

1. Using sysrepo tools sysrepoctl/sysrepocfg: The first option of communicating with sysrepo is by using its CLI tools described in section 1.2.4. Calling other programs and processing their results can be overly complicated, but despite that, it is probably the simplest way.

2. Python bindings in SWIG: Even though there seems to be a plan to create native client libraries on sysrepo's GitHub page [18], they do not exist yet. Instead of that, we have an option to use SWIG bindings to languages different from C, from which we will focus here on bindings to Python version 2 and 3, that are supported by sysrepo. SWIG is

[32]`https://sass-lang.com/`
[33]`https://www.python.org/dev/peps/pep-0333/`
[34]`https://httpd.apache.org/`

a system that enables generating bindings to several languages with minimum effort. The price for that is that a manipulation with such library might not be as pleasant as with the library that would be written in a particular language.

3. Writing our own library: The last and the most demanding option is to write our own library in Python. Such library could be tailored exactly to our needs, but represents the most time-consuming solution.

2.4 Reproducible Environment

In this thesis I use quite a lot of various technologies: the Supervisor is written in C and requires sysrepo and libtrap which are both libraries written in C as well and so there is a need to compile them all. Lgui uses Angular, Python and it also requires web server for the production deployment. That is a lot of dependencies and if someone was forced to install them all one by one, he might get discouraged by it. As I have already explained in the first chapter, STaaS uses Ansible, alternatively in combination with Vagrant, for automatization of this process. I decided to choose Docker[35] in this thesis as a technology replacing virtualization of Vagrant virtual machine provisioned using Ansible. In Docker, a virtualized system is called a container and on Linux it uses a resources isolation provided by kernel, by which it can have a lesser footprint compared to a virtualization of the entire hardware (on Windows and Mac there is a need to virtualize Linux first and then to use Docker in it). The lesser requirements on system resources are the reason why I chose it.

For the users of my solution it will be sufficient to install Docker, to download or to build appropriate Docker image, which stores file system of a container, and by one command the pre-prepared environment, containing the new NEMEA Supervisor and the GUI, will launch.

[35]`https://www.docker.com/`

Chapter 3

Implementation

In this chapter, I would like to focus in more detail on the implementation of the whole solution. In figure 3.1 we can see in outline, how and by using which technologies the individual components of the solution communicate with each other. In the following text, I am going to describe these components one by one, except sysrepo, which was not developed as a part of this thesis. In the end of the chapter, there are included information about implementation of the reference NEMEA module using sysrepo and about the reproducible environment for the whole solution.

3.1 Supervisor Daemon

In this section I would like to give a detailed description of the Supervisor's implementation from two different points of view — first from the viewpoint of high-level architecture, then from the viewpoint of meeting the functional requirements enumerated in section 2.2.2. Source codes of the Supervisor along with the instructions on how to build it and documentation for developers are available on the enclosed CD or on GitHub[36].

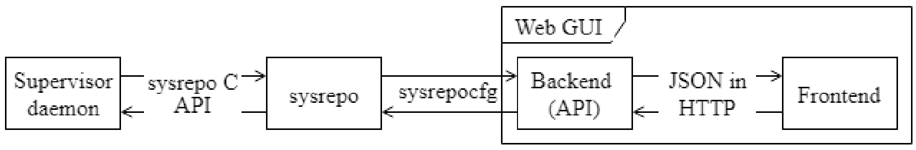


Figure 3.1: Communication overview of the whole solution.

[36]http://github.com/zidekmat/nemea-supervisor-sysrepo-edition

3.1.1 High-Level Architecture Overview

When we look at the Supervisor solely from the perspective of threads, it might give us an impression of being a simple application using just a single thread. This changes the moment we realize, that callbacks are registered to sysrepo and that sysrepo creates a new thread for calling these callbacks. The whole lifecycle diagram, that should give a comprehensive idea of asynchronous callbacks in daemon, can be seen in figure 3.2. Actors A, B and C represent here either the users or programs. It is not necessarily required that all the flows would be initiated by a single actor, so they are separated in the diagram. The role of the actor A is just to launch the Supervisor. The Supervisor then loads the configuration, registers callbacks and launches the main thread with its daemon routine, where the Supervisor runs until it is stopped. The actors B and C can represent for example the backend web API (which I describe later) and their requests are processed each in its own thread under the Supervisor.

3.1.2 Loading Startup Configuration (F1)

Sysrepo C API gives us several options for loading data under specific XPath. Either we can load more data leaves, for instance in an entire list, by specifying the path to a list root, or we can load the data from a whole exact path to one data leaf. I use a combination of both approaches, where I initially load the lists and according to whether it is an available module, an instance or an interface I then extract individual data leaves of a specific object by using exact XPath to leaves.

3.1.3 Daemon Routine (F2, F3)

The main part of the Supervisor is its daemon routine in the function `supervisor_routine`. A concept of this loop is borrowed from the original version of Supervisor, while at the same time I adjusted the code and, hopefully, made it more organized and lucid. In the loop, the following actions are being executed over and over again (this list does not express an order of such actions):

- All instances, that do not run even though they should, will be started. A configuration variable *max-restarts-per-min* is used here. This value is convenient in case we have an instance that functions correctly only after some other instance/instances, that is/are launching slowly, is/are started. When we set a larger value for such instance, the Supervisor will try to start it more times until it works. Such approach surely is not very elegant, but it definitely is much simpler than to implement a dependency logic because of the few edge cases. This functionality existed also in the original version of Supervisor and it functioned sufficiently.

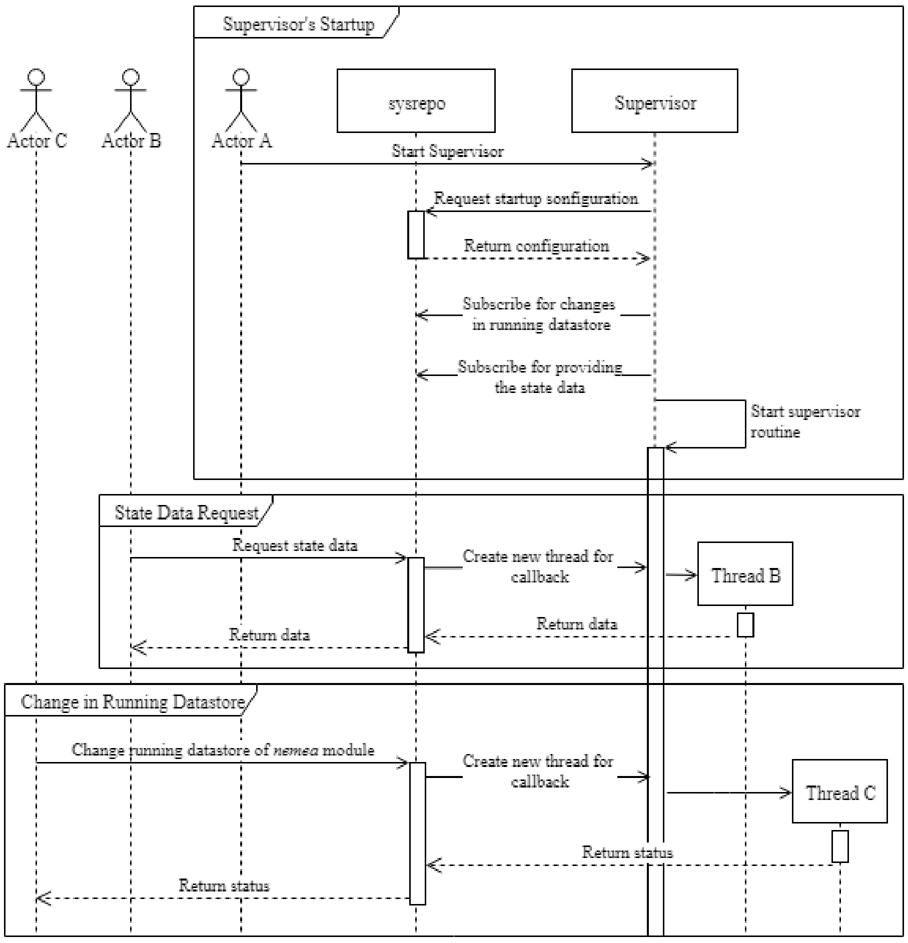

Supervisor's Startup
Actor C
Actor B
Actor A
sysrepo
Supervisor
Start Supervisor
Request startup sonfiguration
Return configuration
Subscribe for changes in running datastore
Subscribe for providing the state data
Start supervisor routine
State Data Request
Request state data
Create new thread for callback
Thread B
Return data
Return data
Change in Running Datastore
Change running datastore of nemea module
Create new thread for callback
Thread C
Return status
Return status

Figure 3.2: High-level architecture overview of Supervisor's implementation.

- All instances, that do run even though they should not, will be stopped.
- It is verified which instances do run using standard function `kill`[37].
- Up-to-date information about use of system resources used by running instances are gained from the special files `/proc/{PID}/stat` and `/proc/{PID}/status`, where `{PID}` is a process identifier of an instance's process.
- If an instance has monitorable libtrap interfaces then they will be connected to its service interface (a UNIX socket created by libtrap).
- Runtime statistics about instances' interfaces are downloaded from connected service interfaces. Those data come in JSON format and are

[37]`https://linux.die.net/man/3/kill`

checked whether a number of interfaces in them corresponds to the number known by the Supervisor (or in other words a number of interfaces set in the sysrepo running datastore). This is why it is necessary to set interfaces for all instances, whose module has a flag *trap-monitorable* set to *true*, even if their configuration is not transferred through CLI.

3.1.4 Providing Runtime Statistics (F4, F5)

Runtime statistics are provided by callbacks subscribed at two XPaths. For the sake of completeness, I would like to add which state data elements they are. At first, I am going to describe statistics about usage of system resources. For gaining such statistics, I have reused functions of the original Supervisor. We can load them for all instances through sysrepo at XPath `/nemea:supervisor/instance/stats`. For a single instance those data are:

- `/running`: A boolean value stating whether the instance is running.
- `/restart-counter`: A number of restarts in a last minute.
- `/cpu-user`: A percentual CPU usage in user mode or 0 in case the instance is not running.
- `/cpu-kern`: A percentual CPU usage in kernel mode or 0 in case the instance is not running.
- `/mem-rss`: A value of resident set size (RSS) portion of random-access memory (RAM) used by the instance process or 0 in case the instance is not running.
- `/mem-vms`: A value of virtual memory size, which is all the memory the instance process can access, meaning all swapped memory, all allocated memory and size of memory of the shared libraries. In case the instance is not running, 0 is returned.

Methods for loading the statistics from a libtrap's service interface are also reused from the original version of Supervisor. Through sysrepo we can find those statistics for all interfaces of specific instance *inst1* by making request for data under XPath
`/nemea:supervisor/instance[name='inst1']/interface/stats`. There are two different sets of statistics according to a direction of an interface. For input interfaces they are:

- `/recv-msg-cnt`: A number of received messages.
- `/recv-buff-cnt`: A number of received buffers. Buffers are blocks for sending messages in larger packs and their usage can be set individually for each interface.

For output interfaces they are:

- `/sent-msg-cnt`: A number of sent messages.
- `/sent-buff-cnt`: A number of sent buffers.
- `/dropped-msg-cnt`: A number of dropped messages.
- `/autoflush-cnt`: A number of autoflushes. Autoflushes are events that occur once an every predefined time period, during which a buffer is sent even when it is not full. The behaviour of autoflushes can be set individually for each interface.

3.1.5 Changes in the Running Datastore (F6)

Processing of changes in the sysrepo's running datastore is an issue that I made the most efforts to deal with and it also cost me the most of my time. Thanks to that I was able to ensure, that the Supervisor reacts adequately only to particular changes of the configuration. That means there is no need to load the entire configuration in case of any unspecified change occurs and also that only those instances, that are truly needed to be restarted, are restarted.

The whole process unfolds in the following manner: before the Supervisor jumps in its daemon routine, it subscribes its callback function `run_config_change_cb` to sysrepo, so it would be called in case a change occurs in the `/nemea:supervisor` subtree. The moment that the change occurs, the sysrepo library creates a new thread within the Supervisor, calls a subscribed function in it and hands over the changes into it. In this array of changes every item has the following information:

- An exact XPath of an element that has changed.
- A change operation type, which can be: created, modified, deleted or moved. The type moved does concern only lists, which we currently do not need to sort. That is why the changes of this type are ignored by the Supervisor.
- What was a value before the change occurred (if there was any value).
- What is a new value after the change happened (if there is any value).

3.1.5.1 Creating Runtime Changes (Runches)

Gained array of changes is processed in sysrepo in a way that each change is transferred into its own format. From an XPath and a type of single change, the Supervisor deduces which of his elements have changed and which action is supposed to be executed in reaction. In the code I decided to store this representation in my own data type `run_change_t` (runtime change).

In the remaining text I would like to refer to it as *runch* (plural *runches*). Depending on the processed XPath, a runch can be of one of the following types:

- A change of an instance's root element, e.g.
  `/nemea:supervisor/instance[name='inst1']`
- A change of an instance's child node, e.g.
  `/nemea:supervisor/instance[name='inst1']/enabled`
- A change of an available module's root element, e.g.
  `/nemea:supervisor/available-module[name='mod1']`
- A change of an available module's child node, e.g.
  `/nemea:supervisor/available-module[name='mod1']/path`

Although there are four aforementioned types of runches, when it comes to an action that is supposed to be made in reaction to a change, we are interested solely in whether the change is a change of an instance or of an available module. The reason is that for those changed objects, managed by the Supervisor, actions that are then executed are derived from the types of change operations in sysrepo. Explanation of their derivation is going to be provided further down. Right now, I am going to describe types of actions in runches:

- **Delete**
  - For a module: Stop all instances of a deleted module and remove associated internal structures.
  - For an instance: Stop an instance and remove internal structure of this instance.
- **Restart**
  - For a module: If there are any instances in the Supervisor and they are running, stop them. Then load a new configuration of the module and of its instances from the running datastore (and eventually start the module's instances in daemon routine thread).
  - For an instance: If an instance is already in the Supervisor and is running, stop it. Then load a new configuration of this instance from the running datastore (and eventually start the instance in daemon routine thread).
- **None**: Take no action, ignore this sysrepo change.

Derivation of action for a runch is carried out according to the following table 3.1.

Type of runch	Change operation type	Action to assign
Module's or instance's root element	created, modified	restart
Module's or instance's root element	deleted	delete
Module's or instance's child node	created, modified	restart
Module's or instance's child node	deleted	restart

Table 3.1: Schema of runch type derivation.

3.1.5.2 Creating List of Registered Changes (Runches)

Delete and restart actions of a runch are both relatively time-consuming. For example, in case we will obtain more runches in a callback that are of instance's child node type, we would not want to restart the instance multiple times when once is just enough (during first restart a new configuration containing all changes is already loaded). To deal with this problem, we need a mechanism by which runches will be "merged" so that the same action would not be repeatedly executed and only a minimum of time-consuming actions would take place. This is achieved by adding the runch, that already has its type and action assigned, to a list of registered runches, which signifies that actions of those runches remain to be processed (except from runches of a *none* type, those will not be added). During the process of adding it is checked whether the runches are not supposed to be merged or otherwise replaced, so we can execute as little actions as possible. In more detail I describe the mechanism of adding in the list below. I mark each condition, under which merging or replacing of a runch happens, with a particular number. If an incoming runch does not match with any undermentioned condition, it is added to the list of registered runches.

1. Is there a runch of the same type that also points to the same object (identified by name) in the registered runches list?

    a) And does the incoming runch concern a change of a root element?
    Action to take: Replace an action in a runch, that is already registered, with an action in the incoming runch.
    Explanation: This solves the following scenario. We have a runch of module M with an action A_1 and after that, we receive the second runch for the same module M with an action A_2. We only care about the action A_2 because it always overwrites an effect of A_1, therefore there is no need to process A_1. The same applies to

instances.

b) And does the incoming runch concern a change of a non-root element or in other words a change of a child node?
Action to take: Ignore the incoming runch.
Explanation: The incoming runch can be ignored because by changing a node of an instance or a module we cause a restart of that object. Because we have already registered one restart for such object, there is no need to add another.

2. Is the incoming runch intended for an instance? And does there already exist a runch of a module, that contains an instance specified in the incoming runch?
Action to take: Ignore the incoming runch.
Explanation: The instance specified in the incoming runch will be deleted or restarted anyway by an action within an already registered runch.

3. Is the incoming runch intended for a module? And do there already exist runches of instances that belong to a module specified in the incoming runch?
Action to take: From the registered runches list, delete those runches of an instance type, that belong to the module specified in the incoming runch.
Explanation: Those registered runches will be deleted or restarted by an action within the incoming runch of a module. That is why there is no need to keep them in the list.

3.1.5.3 Processing Registered Runches

Before the registered runches begin to be processed one by one, the last step remains to be done. The array of runches needs to be sorted in a manner that the modules will be placed at the end of the list. This is important in the case that the new module and its instance are added to the registered runches list within the same callback function call. If the Supervisor has not loaded the module yet, it would not know, that the newly added instance belongs to it and as a consequence of that it adds both changes to the registered runches list. If the instance was processed before processing of its module, the Supervisor would not load the instance, because it would not be able to match it with a corresponding module (for it has not been loaded yet). That is why I sort the registered changes so there would be instances in the beginning and modules in the end of the list. The reason for that is the processing proceeds from the end and for that reason, instance will not be loaded before module. For sorting I use plain bubble sort since the registered changes list is of a small extent typically and so its worst-case performance of $O(n^2)$ does not have to bother us.

3.1.6 Components

Compared to the original implementation, I have divided the Supervisor's code into multiple C files, each of which has its own header file. Every pair of such files corresponds to one block of the Supervisor's functionality in the following way:

1. `conf`: Serves for loading configuration from sysrepo.
2. `inst_control`: Takes care of starting and stopping of instances.
3. `main`: Entry file containing `main` function, where CLI arguments are processed and where the Supervisor is launched.
4. `module`: Contains internal structures of all available modules and instances, functions for allocation/release of internal structures and also dynamic arrays that contain lists of available modules and instances that are part of the running configuration.
5. `run_changes`: Exports just one function that is called in case a change of `/nemea:supervisor` subtree in sysrepo's running datastore occurs.
6. `service`: Contains functions for connecting and gaining statistics about interfaces from a service interface created by libtrap.
7. `stats`: Provides callbacks for sysrepo for the purpose of gaining statistics about instances and their interfaces.
8. `supervisor`: This is the core of the whole daemon where the daemon routine and its related functions are implemented.
9. `utils`: Provides helper macros and functions shared within the whole daemon, such as for example a macro for logging or implementation of a simple dynamic array for storing data about modules and instances.

3.1.7 Documentation

Functions and modules are documented in the Supervisor using Doxygen[38] syntax, so a documentation for developers can be generated in HTML. The

[38] `http://www.doxygen.org/`

generated documentation can be found on the enclosed CD, is available online[39] or can be built by the developer himself using the instructions in `README.md`[40] on the enclosed CD. For online hosting, I use a free hosting service GitHub Pages[41].

3.2 Graphical User Interface

So the already existing GUI for NEMEA would not be split, I have created the GUI part of this thesis as a component of the NEMEA Dashboard lgui module. I decided, that the linking of the GUI developed in this thesis and NEMEA Dashboard, should be called NEMEA GUI in order to express, that it stands for a combination of all graphical user interfaces for NEMEA project.

Instructions for the GUI installation and also the GUI itself are available on the enclosed CD and publicly on GitHub[42].

3.2.1 Frontend

The implementation of a GUI frontend for the Supervisor is strictly minimalistic and corresponds with the wireframes quite well. This means a greater attention could have definitely been paid to its design. It is not necessary to include examples of screenshots here, since it is, in fact, not very different from the actual wireframes that can be found in the appendix B.

The functionality of the frontend is limited only to the bare essentials, based on requirements defined in section 2.3.1. Although there are a few details in addition to it, such as a functionality of prefilling names for example during creating an instance or an interface, in terms of user-friendliness, frontend does not stand out particularly.

3.2.1.1 Architecture Overview

The implementation of frontend part of NEMEA GUI is located in a separate folder `supervisor` in the frontend part of NEMEA GUI lgui module. The specific architecture would be more clear after taking a look at the directory listing in figure 3.3. There we can see a root component `supervisor-gui` containing a menu that is common for all other components and also serves as a frame to which individual pages from the `pages` file are inserted.

The folder `models` contains Typescript classes representing JSON objects that are sent and received over HTTP. Compared to classic JSON objects, classes can have functions. I use this particular advantage of Typescript to implement serialization and deserialization functions that allow me to convert

[39] `https://zidekmat.github.io/nemea-supervisor-sysrepo-edition/index.html`
[40] File `README.md` is located in directory `src/nemea-supervisor/`.
[41] `https://pages.github.com/`
[42] `https://github.com/zidekmat/nemea-gui`

Figure 3.3: Folder structure of the NEMEA Supervisor GUI.

```
supervisor/
├── supervisor-gui.component.*
├── models/
│   └── ...
├── pages/
│   ├── shared/
│   └── ...
└── services/
    └── ...
```

classes to JSON objects and back. The need for these functions is because the API is sending variables containing a dash (-), but Typescript is not able to assign such variables to its classes. It is, therefore, necessary to transfer those variables to underscore notation using an underscore (_) instead of dash so that Typescript would understand them.

At the end of the listing in the figure only services remain, where we can find classes for HTTP communication with the backend API.

3.2.2 Backend

As a backend for Supervisor GUI I implemented RESTful[43] JSON HTTP API in Python 3. I created the user documentation in format Swagger 2.0[44]. It is available on the enclosed CD or online[45] on SwaggerHub, which is a service enabling public hosting of interactive Swagger documentation.

3.2.2.1 Functionality Overview

This API was implemented just for a use case, where we have a running Supervisor, which is, however, presumed to be a constant state of the Supervisor developed in this thesis. The API also operates on top of both running and startup datastore simultaneously, which means it executes changes in both of them, so the configuration would not differ. Since the API does not implement any locking mechanism on top of a database (in our case on top of sysrepo) it could happen, that due to a parallel processing of HTTP requests, the datastores might get into an undesired state. To prevent potential problems of this kind, there is supposed to be just one WSGI thread for backend in the production configuration of NEMEA GUI. This might have, theoretically, a certain impact on API performance, but that does not have to overly concern

[43] https://en.wikipedia.org/wiki/Representational_state_transfer

[44] https://swagger.io/

[45] https://app.swaggerhub.com/apis/nemea-supervisor-gui/nemea-supervisor-api/1.0.0

us, as the API here is a configuration tool, which means the performance is not a feature that would be of any great importance.

Now, I would like to return to the functional requirements for GUI defined in section 2.3.1. From the backend part, those were implemented mainly using CRUD[46]) operations over the configuration of NEMEA modules and their instances. Specific functions that the web API contains and offers are stated in the list below. In this list, I intentionally left out the functions used for the modules using sysrepo — since the frontend for them is not implemented yet, they can be rather considered to be a part of the future work.

1. Create a new module.
2. Retrieve a module configuration by name.
3. Retrieve a configuration of all modules.
4. Update a module configuration by name.
5. Delete a module (including its instances) by name.
6. Create a new instance for some module.
7. Retrieve an instance configuration by name.
8. Retrieve a configuration of all instances for a single module by module name.
9. Retrieve a configuration of all instances.
10. Update an instance configuration by name.
11. Delete an instance configuration by name.
12. Retrieve system resources statistics and libtrap interface statistics for a single instance by its name.
13. Retrieve system resources statistics and libtrap interface statistics for all instances.
14. Start or stop an instance by name.

[46]create, read, update, delete as described for example here: `https://en.wikipedia.org/wiki/Create,_read,_update_and_delete`

3.2.2.2 Authorization

The API supports only authorized requests. This functionality is based on the session and authorization features provided by lgui and thanks to that there is no need for us to implement it by ourselves. Due to the fact, that the API enables operations, that are dangerous from a security standpoint, such as the launching of given application, the requirements for the API are processed only in case the authorized user has the highest access rights — role *admin*.

3.2.2.3 Architecture Overview

The API implementation is a part of NEMEA GUI lgui module, where the part related to the Supervisor is situated in a separate folder (Python module). The API is architecturally quite simple — it contains two building blocks: *controllers* and *models*. In controllers, we can find functions, that take care of processing HTTP requests, which are identifiable according to RESful principles using URL path and HTTP method. The controllers also secure an implementation of individual API functions and a manipulation with data that represent the Python objects directly. The models are responsible for loading data from sysrepo to Python objects and back.

3.2.2.4 Communication with sysrepo

In the models, there were three possible ways of dealing with communication with sysrepo, that are listed in section 2.3.4. From those options, I chose using sysrepocfg. Running of CLI tool from programming language might not seem as a proper solution at first glance, but more thorough analysis and extensive testing of possible solutions had shown, that this is the simplest approach, that does not carry any larger imperfections.

The simplicity of the sysrepocfg usage lies mainly in the fact it can communicate using JSON, as well as the API does. This is why this solution requires the format to be only slightly corrected and also a few checks to be done (in terms of a possible incorrect data manipulation). In other words, it needs to be checked, whether the data are not being written to places where they are not supposed to be written to or whether the data are not being read from the wrong places. In contrast to sysrepocfg, SWIG API requires all values to have their XPath defined while communicating with sysrepo. Therefore we would need to convert every key in a key-value JSON pair to XPath and back. On the other hand, using sysrepocfg means that we do not have a support of a candidate datastore for data validation. This shortcoming is eliminated by the API operating and executing changes in both startup and running datastore. This enables us to use startup datastore as candidate datastore. The running Supervisor is not interested in changes in startup datastore and if they pass and are saved, there should not be any other problems left and the startup configuration can be synchronized to the running datastore.

Although it would certainly be the best fitting solution, I decided not to choose the option of creating my own library in Python. Considering that amount of work related to the implementation part of this thesis was not exactly small, this solution with its excessive demands on time would pose a significant complication to the whole implementation process.

3.2.3 Inability to Change List Key in YANG

Finally, I would like to mention one specific issue, that the reader could be surprised by. The libyang library, used for YANG-related operations by sysrepo, does not enable to change values of nodes, that represent the keys in the YANG lists. This means that when we want, for example, to change a name of an instance, we must create a new instance with a different name and similar values and erase the original instance.

3.3 Modules Using sysrepo

In the previous chapter, I have designed the data model that can be used also for an implementation of modules using sysrepo. I have fulfilled the main goal of the thesis — configuration of the Supervisor by using GUI — but, unfortunately, there was no time left to implement GUI for configuration of those modules. At least the reference module *link_traffic* that I have already mentioned has been implemented and is available on the enclosed CD or on GitHub[47].

3.4 Reproducible Environment

For the implementation of the reproducible environment, I have created two demo environments in Docker.

The first version has a setting similar to STaaS, where the original Supervisor is replaced by the version developed in this thesis. The docker image can be either downloaded from Docker Hub[48] (hosting service for Docker images) or built by the user himself from Dockerfile on the enclosed CD.

The second version is based on the first one and adds the GUI developed in this thesis to the fully-fledged testing version of STaaS with sysrepo. It is also available on Docker Hub[49] or on the CD for purposes of building the Docker image on local host.

[47]`https://github.com/zidekmat/Nemea-Modules/tree/master/link_traffic`
[48]`https://hub.docker.com/r/zmat/nemea-supervisor-sysrepo-edition/`
[49]`https://hub.docker.com/r/zmat/nemea-gui-sysrepo-edition/`

Chapter 4

Testing and Evaluation

This chapter describes methods and procedures, that I used for testing the solution that was developed in this thesis. In the end of the chapter, I am going to look back at the whole development from the perspective of its imperfections to suggest ways of the future improvement.

4.1 Supervisor Daemon

In the process of developing the Supervisor daemon, I used mainly automated testing for verification of its correct behavior. Besides that, I naturally subjected the daemon also to the manual testing. For these purposes, I created three sets of the Supervisor configuration having 3 to 33 NEMEA module's instances. With all configurations the Supervisor behaved as expected.

4.1.1 Automated Testing

Since in C language the programmer must manage its dynamically allocated memory by himself, mistakes might occur, causing instability of the whole program. To eliminate such mistakes and prevent the consequent problems, I used Valgrind[50] for testing a possibility of incorrect memory handling during the development. Owing to this tool, I was able to find out the location of the problems and to remove them.

As another method for automated testing, I implemented unit tests that secure a proper functionality of the key parts of the Supervisor. Altogether, I implemented 35 of such tests using the framework cmocka[51] that are available in the folder `src/nemea-supervisor/tests` on the enclosed CD. All tests can be launched at the same time by using the script `run_tests.sh`, which takes care of creating and destroying the testing environment. I describe this procedure in more detail in the following text.

[50] http://valgrind.org/
[51] https://cmocka.org/

4.1.1.1 Own YANG Module for Unit Testing

After the `run_tests.sh` is launched, the current YANG model used by the Supervisor is copied and renamed to *nemea-test-1*. By loading this model to sysrepo we create an environment for testing data that can be changed while the unit tests are running. The reason for that is the name of the YANG module, that the Supervisor is supposed to use, is defined in its source code by C preprocessor (described in [19] page 90) for a regular compilation as the constant string *nemea*. When script compiles the tests, the constant is replaced with *nemea-test-1* and thanks to that the unit tests will use data in the YANG model designated for testing purposes, not in the production YANG module *nemea*.

4.1.1.2 The Need for Helper Applications

For most of the tested functions in the Supervisor it was enough to mock objects used by those functions, however, there were also a few exceptional cases that required specific treatment and implementation of special helper applications:

1. `intable_module.c` The Supervisor starts and stops independently running applications. This helper program was created to enable the unit tests to test such functionality.

2. `async_stats.py` Python script using sysrepo's SWIG bindings for Python, that runs asynchronously with the unit tests and serves for testing functions providing state data.

3. `async_change.py` Python script using sysrepo's SWIG bindings for Python, that runs asynchronously with the unit tests and serves for testing reactions to changes in the running datastore used by the Supervisor.

4.2 Graphical User Interface

Similarly to the Supervisor testing, testing of the GUI was also partly automatized. Namely, I used automated testing for the API testing. The functionality of the frontend was tested by the manual approach and then by walkthrough with users.

4.2.1 Automated Testing of API

For automated testing of API, I decided to use the framework that is a part of standard Python library — *unittest*. I did not test all the functions — the testing was limited only to the functions in *controllers* that are responsible for a correct answer to a HTTP request. I worked on the assumption that if those functions work correctly, the rest of functions will most likely work as well, given the fact that they are used by the tested ones. This way we get the guaranty of at least some level of a correct behaviour.

The same way as the Supervisor testing, automated testing of API could not be made without the helper programs. This time it was sufficient to use just two of them. The first, `stats_provider.c`, is used to provide dummy data so there is an application connected to sysrepo that provides state data without the need of starting the Supervisor. The second, `subscriber.py`, is used only to subscribe to the changes in the running datastore, so the running datastore would be filled with data and the API functions would be able to work with it.

4.2.2 Manual Testing of Frontend

The functionality of the GUI frontend was tested manually. I also tested the first working version by application walkthrough with users, namely with the members of NEMEA team including my supervisor. This walkthrough focused mainly on the user experience. In this case, the tested version of the frontend was not yet integrated to the final version of API and Supervisor daemon, so only dummy data, not the real data from sysrepo and the Supervisor, were involved. As a result of this testing, the specification of requirements for the future work and improvement was created. This specification is available in the appendix C.

4.2.3 Future Work/Evaluation

In hindsight, it is obvious that most of work and time was dedicated to the modeling of the configuration and implementation of the Supervisor daemon. That resulted in developing only the basic GUI of the whole solution that would be useful to further enhance. It should be admitted that better approach would be to create the basic version of the entire solution first and then to improve it step by step as a whole. Nevertheless, the developed system is going to be deployed and therefore tested more extensively.

Speaking from experience gained through development, I would like to point out the imperfections of the solution that I have not managed to fix in time and that would thus be convenient to eliminate by future work.

4.2.4 Issues in YANG Model

1. **RPC for Restart of an Instance**
   At the moment, the Supervisor does not natively implement an ability to restart an instance. The GUI implements it merely by setting its running configuration to disabled mode, waiting a bit and then re-enabling it. This solution requires more work on the side of the client requesting restart and it might have been a much cleaner solution to implement

the restart functionality as a RPC in the YANG model using the *rpc* statement and to let the Supervisor do the work.

2. **Missing Selection of Revision for the Custom YANG Model**
The YANG modules generally support specification of the revision dates to provide versioning of the models. When I was designing the support for the custom YANG model used by the NEMEA modules, I did not realize this particular fact. That means I completely omitted an option for the NEMEA modules to specify the version so just the latest version is used. This issue should be taken care of by adding the YANG leaf node *sr-model-rev*, specifying the date of revision that is supposed to be used.

3. **Referencing to Available Modules from Instance**
The way it is currently implemented, an available module, which is in a relationship with an instance as its parent, is referenced from an instance using *leafref*. Let us imagine a case in which change in some instance happens and the Supervisor is currently in the process of reacting to the change. Before the Supervisor tries to add the change to the array of registered changes, it needs to lookup an available module of an instance that has been changed. The Supervisor would not have to do it if the form of the YANG model was used, where the instance list is a direct child of the available module list. If the Supervisor received a change of an instance (e.g. XPath
`/nemea:supervisor/available-module[name='m1']`
`/instance[name='i1']/enabled`) in such case, it would know right away, where it belongs, from the XPath of the change.

4.2.5 Issues with Graphical User Interface

Imperfections of the GUI are described in the already mentioned specification in appendix C. The current GUI is a very simple tool for the time being and if it is supposed to replace the existing remote access for an administrator using CLI tools, it has still a long way to go. I consider the most significant issue beyond the specification to be the missing part of GUI for the modules using sysrepo. In reality, only a single module exists so far and that is the reference module created in this thesis. It would be necessary to adjust the GUI developed in this thesis at some time in the future so it would be able to configure also those modules, that do not exist at the moment.

Conclusion

In this thesis, I have described the former version of a way of the NEMEA system configuration, introduced the new configuration model for the NEMEA system and described the technologies by which it might be improved. On the top of the YANG and sysrepo technologies, I have designed the new configuration data model that enables larger control over the configuration data and the NEMEA modules. In addition to that, I have designed the model so it would be extensible enough to support future NEMEA modules that might use sysrepo as storage for their particular data and settings. I have also adjusted one NEMEA module to be an example of the module using the configuration from sysrepo within the newly created model.

I have designed and implemented the new NEMEA Supervisor daemon that manages the whole NEMEA system. It receives the configuration data from sysrepo and at the same time, it is able to provide the state data to other applications through sysrepo. The Supervisor is also able to react to the changes made to the configuration while running and in reaction to them it executes actions with individual NEMEA modules. The Supervisor's code is covered by prepared unit tests. Integrating the Supervisor with the rest of the system was subjected to a manual testing and is functional.

I have also designed and implemented a simple web user interface for managing the Supervisor and by that the entire NEMEA system. This GUI extends the existing module for Liberouter GUI, so the graphical user interfaces for the NEMEA system would not be split. The backend API of this web application communicates with the Supervisor through sysrepo. The API contains the functional tests of all of its endpoints. I have tested the frontend of the GUI manually and also by walkthrough with users. The users provided me with a feedback, which I have integrated into the specification of the requirements for GUI improvements that is included in the appendix C of the thesis.

As it is the first version of the solution, it provides only basic functionality. It is, therefore, possible to further enhance it. Other suggestions for improvements and future work that do not originate in testing with the users are also included.

Lastly, I would like to add that I have created a reproducible environment where the whole solution is set up in a similar way as it would be in STaaS. The results of this thesis are going to be further tested in a real deployment of NEMEA and STaaS.

Bibliography

[1] Cejka, T.; Bartos, V.; et al. NEMEA: A Framework for Network Traffic Analysis. In *12th International Conference on Network and Service Management (CNSM 2016)*, 2016, doi:10.1109/CNSM.2016.7818417. Available from: `http://dx.doi.org/10.1109/CNSM.2016.7818417`

[2] CESNET. *Intrusion Detection Extensible Alert.* [cit. 2018-05-04]. Available from: `https://idea.cesnet.cz/en/index`

[3] Raymond, E. S. *The Art of Unix Programming.* First edition, 2003. Available from: `http://www.catb.org/~esr/writings/taoup/html/ch01s06.html`

[4] CESNET. *TRAP Interface Specifier.* [cit. 2018-05-04]. Available from: `http://nemea.liberouter.org/trap-ifcspec/`

[5] Bjorklund, M. The YANG 1.1 Data Modeling Language. RFC 7950, RFC Editor, August 2016.

[6] Bjorklund, M. YANG - A Data Modeling Language for the Network Configuration Protocol (NETCONF). RFC 6020, October 2010.

[7] DeRose, S.; Clark, J. XML Path Language (XPath) Version 1.0. W3C recommendation, W3C, 1999, http://www.w3.org/TR/1999/REC-xpath-19991116/.

[8] Lhotka, L. JSON Encoding of Data Modeled with YANG. RFC 7951, August 2016.

[9] Enns, R. NETCONF Configuration Protocol. RFC 4741, December 2006.

[10] Enns, R.; Bjorklund, M.; et al. Network Configuration Protocol (NETCONF). RFC 6241, June 2011.

[11] Wasserman, M.; Goddard, T. Using the NETCONF Configuration Protocol over Secure SHell (SSH). RFC 4742, December 2006.

[12] Wasserman, M. Using the NETCONF Protocol over Secure Shell (SSH). RFC 6242, June 2011.

[13] Goddard, T. Using NETCONF over the Simple Object Access Protocol (SOAP). RFC 4743, December 2006.

[14] Badra, M. NETCONF over Transport Layer Security (TLS). RFC 5539, May 2009.

[15] Badra, M.; Luchuk, A.; et al. Using the NETCONF Protocol over Transport Layer Security (TLS) with Mutual X.509 Authentication. RFC 7589, June 2015.

[16] Alexa, D. *Rozšíření grafického uživatelského rozhraní NetopeerGUI*. Brno: Computer Press, 2015.

[17] CESNET. *Netooper2GUI GitHub repository*. [cit. 2018-05-04]. Available from: `https://github.com/CESNET/Netopeer2GUI/tree/078619d71e0f2c1e06716b966e3738c8274b7ad9`

[18] *GitHub repository of sysrepo*. [cit. 2018-05-04]. Available from: `https://github.com/sysrepo/sysrepo/tree/c4fc434f0640905a3b4e3378151d2c399fba945a`

[19] Herout, P. *Učebnice jazyka C*. KOPP, 6th edition, ISBN 978-80-7232-383-8.

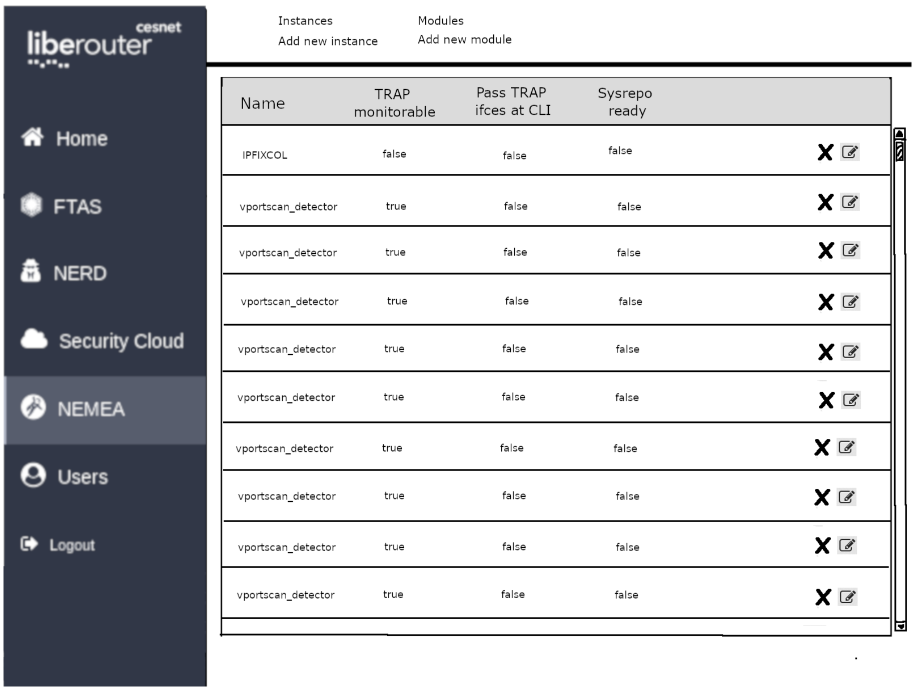
Appendix A

Acronyms

API	Application Programming Interface
CESNET	Czech Education and Scientific NETwork
CLI	Command Line Interface
GUI	Graphical User Interface
IDEA	Intrusion Detection Extensible Alert
JSON	JavaScript Object Notation
NEMEA	Network Measurements Analysis
PID	Process Identifier
RFC	Request For Comment
runch	Runtime Change
STaaS	Security Tools as a Service
TCP	Transmission Control Protocol
XML	Extensible markup language
YAML	YAML Ain't Markup Language
YIN	YANG Independent Notation
WSGI	Web Server Gateway Interface

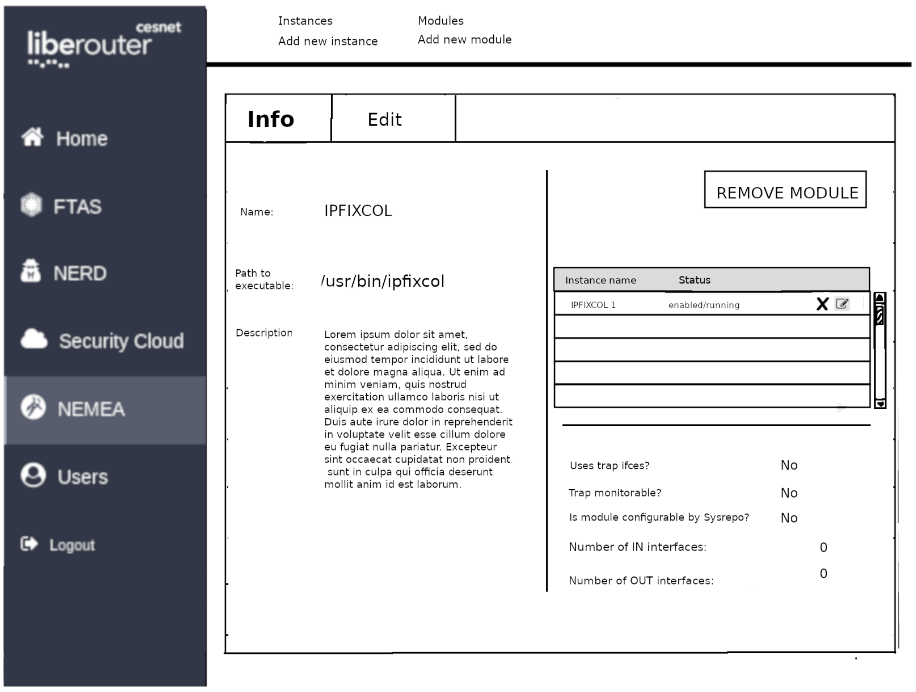
Appendix B

Wireframes



Name	TRAP monitorable	Pass TRAP ifces at CLI	Sysrepo ready
IPFIXCOL	false	false	false
vportscan_detector	true	false	false
vportscan_detector	true	false	false
vportscan_detector	true	false	false
vportscan_detector	true	false	false
vportscan_detector	true	false	false
vportscan_detector	true	false	false
vportscan_detector	true	false	false
vportscan_detector	true	false	false
vportscan_detector	true	false	false

Figure B.1: Wireframe of modules listing page.



Name: IPFIXCOL

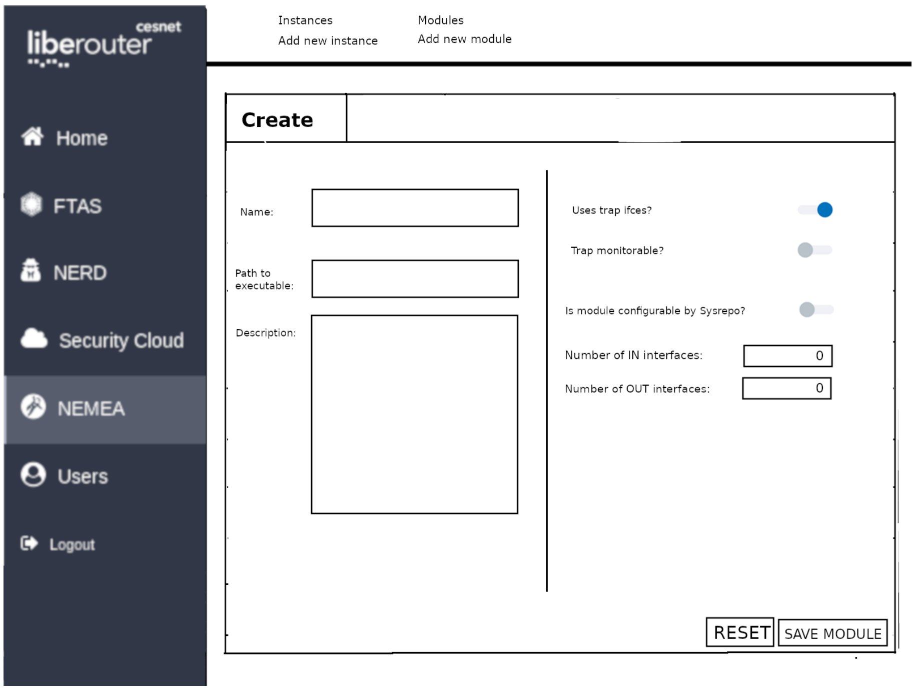
Path to executable: /usr/bin/ipfixcol

Description Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident sunt in culpa qui officia deserunt mollit anim id est laborum.

Instance name	Status
IPFIXCOL 1	enabled/running

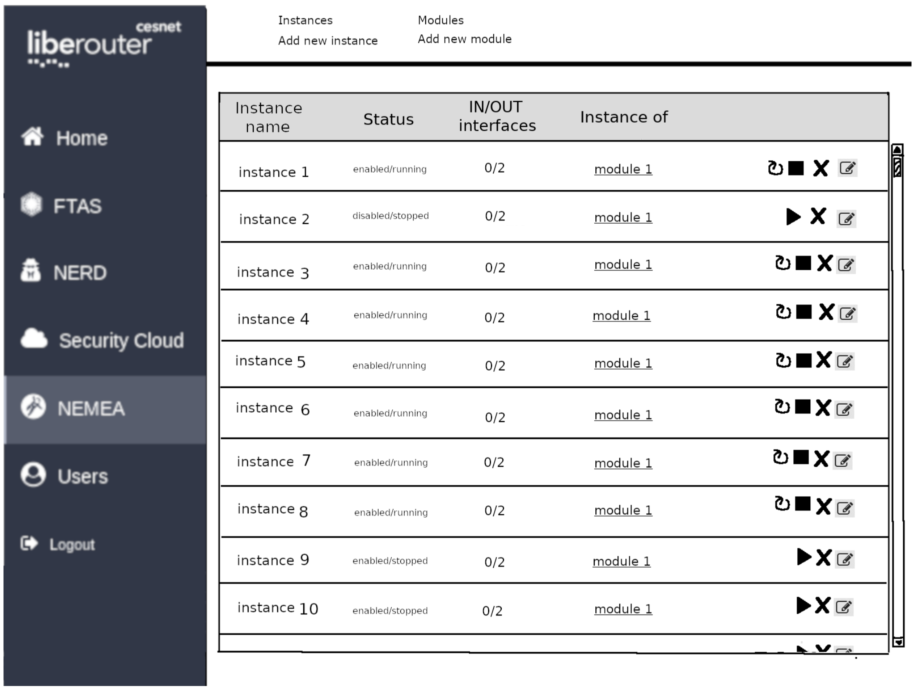
Uses trap ifces?	No
Trap monitorable?	No
Is module configurable by Sysrepo?	No
Number of IN interfaces:	0
Number of OUT interfaces:	0

Figure B.2: Wireframe of module detail page.



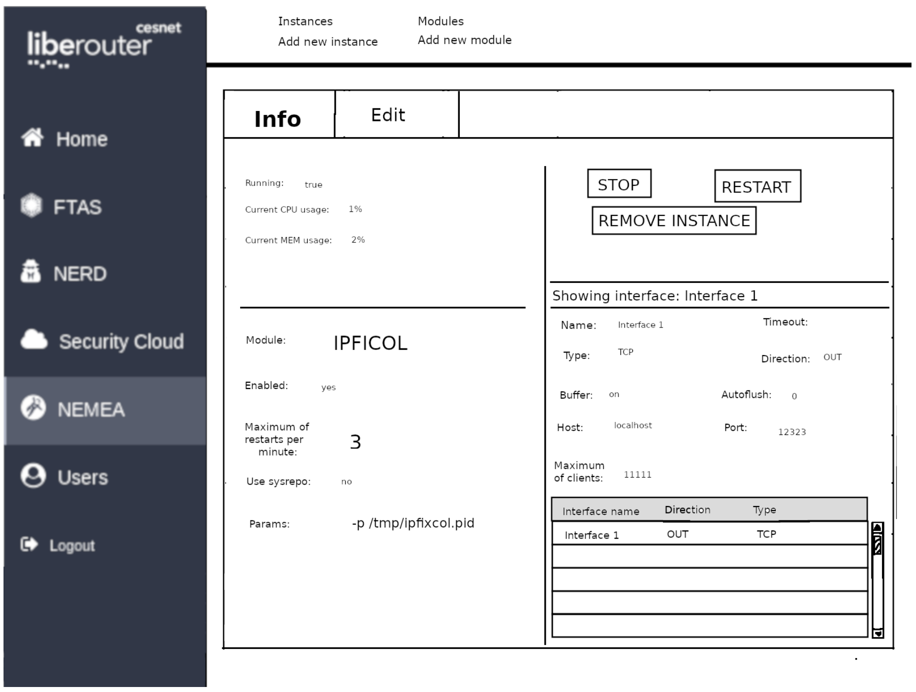
Name:	
Path to executable:	
Description:	
Uses trap ifces?	on
Trap monitorable?	off
Is module configurable by Sysrepo?	off
Number of IN interfaces:	0
Number of OUT interfaces:	0

Figure B.3: Wireframe of module create/edit page.



Instance name	Status	IN/OUT interfaces	Instance of	
instance 1	enabled/running	0/2	module 1	↻ ■ ✕ ✎
instance 2	disabled/stopped	0/2	module 1	▶ ✕ ✎
instance 3	enabled/running	0/2	module 1	↻ ■ ✕ ✎
instance 4	enabled/running	0/2	module 1	↻ ■ ✕ ✎
instance 5	enabled/running	0/2	module 1	↻ ■ ✕ ✎
instance 6	enabled/running	0/2	module 1	↻ ■ ✕ ✎
instance 7	enabled/running	0/2	module 1	↻ ■ ✕ ✎
instance 8	enabled/running	0/2	module 1	↻ ■ ✕ ✎
instance 9	enabled/stopped	0/2	module 1	▶ ✕ ✎
instance 10	enabled/stopped	0/2	module 1	▶ ✕ ✎

Figure B.4: Wireframe of instances listing page.



Running: true
Current CPU usage: 1%
Current MEM usage: 2%

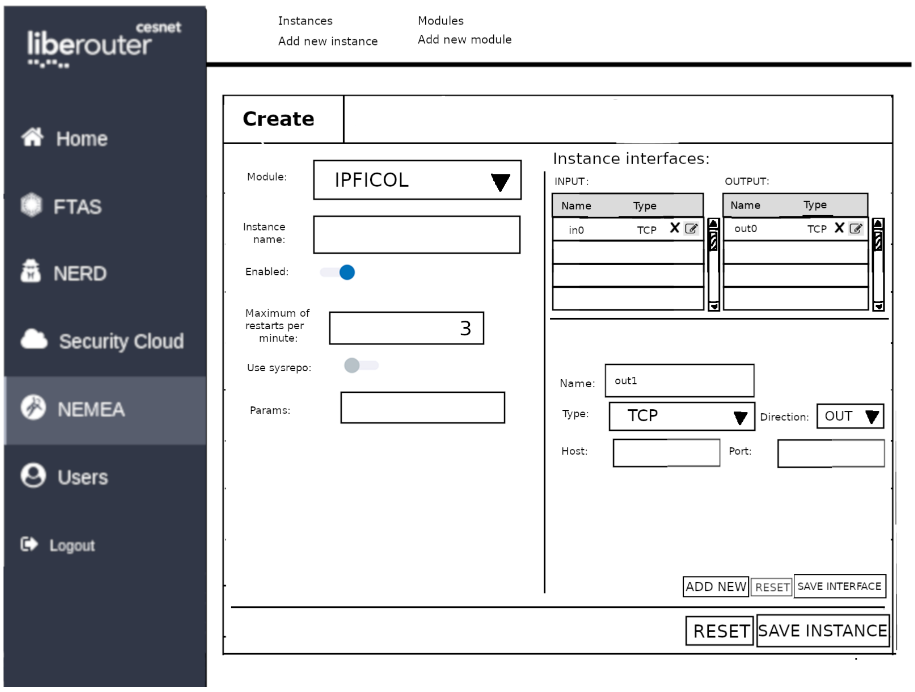
Module: IPFICOL
Enabled: yes
Maximum of restarts per minute: 3
Use sysrepo: no
Params: -p /tmp/ipfixcol.pid

Showing interface: Interface 1

Name: Interface 1
Timeout:
Type: TCP
Direction: OUT
Buffer: on
Autoflush: 0
Host: localhost
Port: 12323
Maximum of clients: 11111

Interface name	Direction	Type
Interface 1	OUT	TCP

Figure B.5: Wireframe of instance detail page.



Module: IPFICOL
Instance name:
Enabled:
Maximum of restarts per minute: 3
Use sysrepo:
Params:

Instance interfaces:

INPUT:

Name	Type
in0	TCP

OUTPUT:

Name	Type
out0	TCP

Name: out1
Type: TCP
Direction: OUT
Host:
Port:

Figure B.6: Wireframe of instance create/edit page.

Appendix C

Specification For GUI Frontend Improvements

C.1 Introduction

This document describes the specification of the requirements for improvement of the NEMEA Supervisor GUI, the web interface for management of the NEMEA Supervisor[52] and by that for management of the whole NEMEA system running on a single server. The wireframes for the original version can be found in the appendix B and the source codes with a description of how to deploy the GUI are available on GitHub[53].

A web implementation will be created on the basis of this specification as a part of NEMEA GUI, which is based on the NEMEA Dashboard[54]. The NEMEA GUI is the module of Liberouter GUI[55] and it is supposed to replace the NEMEA Dashboard as a part of STaaS GUI[56], which is a set of the modules for the Liberouter GUI. The NEMEA GUI is comprised of five parts at present:

1. Dashboard
2. Events
3. Reporter
4. Status (currently not working due to the new version of Supervisor)
5. Supervisor GUI

Part of the NEMEA GUI, described in this document, should adjust the Supervisor GUI. Besides the objective to facilitate the management for present administrators and users of NEMEA system, it is also aimed to create web and making NEMEA accessible to new users, that it why the interface should be sufficiently simple and intuitive. The web should be in English.

C.1.1 HTTP API

The web is going to use the API, that manipulates with the NEMEA configuration saved in the sysrepo datastore. The user documentation for API is available on SwaggerHub[57], where can be also found the models of individual JSON objects, that the API receives and sents. As the most important can be considered three objects, whose relationship is depicted in figure C.1, and might be described as follows:

[52]`https://github.com/zidekmat/nemea-supervisor-sysrepo-edition`

[53]`https://github.com/zidekmat/nemea-gui`

[54]`https://github.com/CESNET/Nemea-Dashboard/tree/7ebf7bd1109ef272ce967b62326e19277d3215d2`

[55]`https://github.com/CESNET/liberouter-gui`

[56]`https://github.com/CESNET/Staas-GUI`

[57]`https://app.swaggerhub.com/apis/nemea-supervisor-gui/nemea-supervisor-api/1.0.0`

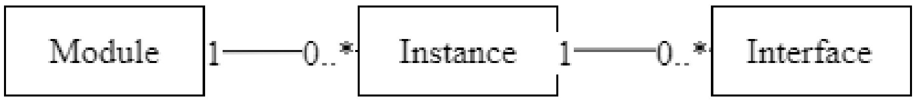


Figure C.1: Relationship between main objects in the web API.

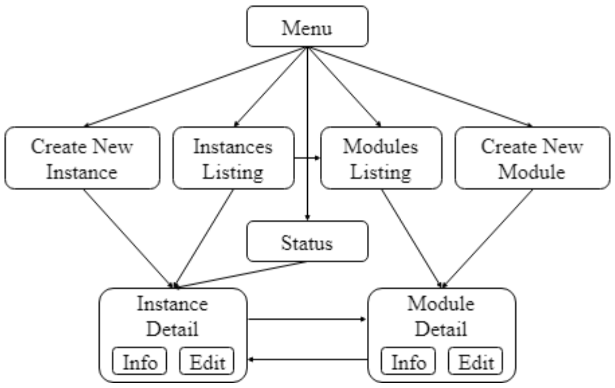


Figure C.2: Visual representation of the sitemap for the improved NEMEA Supervisor GUI.

1. **Module**: “Archetype” of an instance, program, that can be launched multiple times with different parameters, where each launch is called instance.

2. **Instance**: One unique process running in the operating system.

3. **Interface**: NEMEA interface of an instance, or in other words, the way an instance is connected to the other instances. Not every module (and thus not every instance) has to support them. Can be imagined as a file, to where a module A writes and from where a module B reads. That means that for the module A it is an output interface, while for the module B it is an input interface.

C.2 Sitemap

Sitemap of the web GUI after implementation of improvements should look like the one in figure C.2.

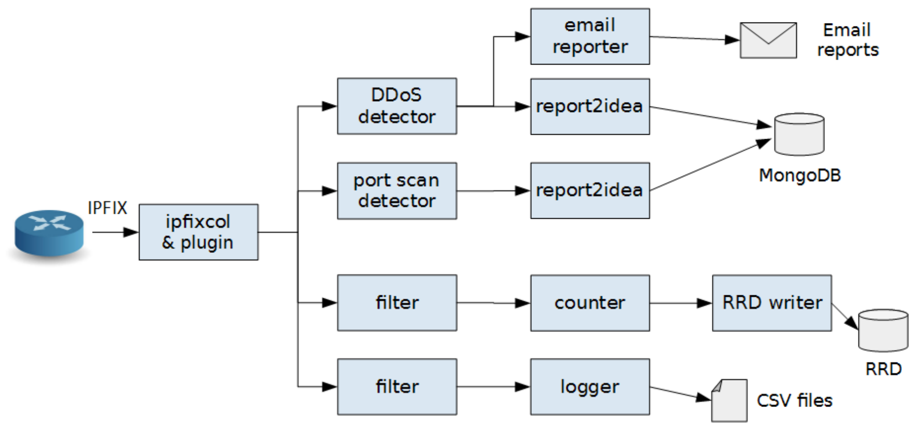


Figure C.3: Diagram of example NEMEA setup. (Source: NEMEA project homepage, `http://nemea.liberouter.org/images/nemea-scheme.png`, downloaded at 2018-05-08)

C.3 Description of Individual Pages

C.3.1 Status Page

This is the new version of NEMEA status[58], which is a part of NEMEA Dashboard. The working demo version is available online[59]. It represents a status (running/not running, CPU and RAM usage) of individual instances, what kind of interfaces they have (type, name) and also statistics from interfaces (number of sent/received messages, number of thrown away/discarded messages). In the new version, instances should have better graphic presentation by being interconnected by their interfaces in a way that can be seen in a diagram of NEMEA setup in figure C.3.

From individual nodes/instances, it should be able to click through to the page C.3.3.

C.3.2 Instances Listing Page

Here should be contained the table with instances listing according to the following schema in table C.4. Also, there should be a button for adding a new instance which should lead to Create Instance Page. The table should have a fixed header so it will stay in place if the user scrolls down. Also, it should have an effect when the user hovers over a row that colors the background of a current row and in case the user clicks on the row, it should take him to Single Instance Page View.

[58]`https://github.com/CESNET/Nemea-Dashboard/tree/7ebf7bd1109ef272ce967b62326e19277d3215d2/frontend/status`

[59]`https://staas-demo.liberouter.org/nemea-status/`

SELECT COLUMN	*name* attribute of the instance	Status (attributes *running* and *enabled*)	INTERFACES COLUMN	*module* attribute of the instance as a link to Single Module View Page	CONTROL COLUMN

Table C.1: Schema of how the content of rows in the table on an instances listing page should look like.

C.3.2.1 SELECT COLUMN

Contains a checkbox in the table header for selection of all rows and on each individual row contains another checkbox for single selection. If at least one row is selected, actions should be displayed on the page from CONTROL COLUMN (except edit option), which might be called for all selected rows. All actions should be displayed because the status can differ in each individual selected row. Action with the rows will require confirmation in the pop-up.

C.3.2.2 INTERFACES COLUMN

1. If an instance is running.
   Contains a number of IN/OUT interfaces. After hovering with the mouse pointer, a tooltip should be displayed, where the interfaces along with their types and names are listed.

2. If an instance is not running.
   Only a dash - is displayed. By hovering the mouse over the minus, a tooltip should be displayed, where the interfaces along with their types and names should be listed so that one would know, what would be defined after an instance starts.

C.3.2.3 CONTROL COLUMN

Contains buttons for:

- **Instance removal**: Requires confirmation in pop-up, notifying user that an instance on the server will be stopped.

- **Instance editing**: Serves just as a link to sec:inst-edit-tab

- **Instance restart**: Requires confirmation in pop-up.

- **Instance starting**: Requires confirmation in pop-up and is visible only if an instance is not running.

- **Instance stopping**: Requires confirmation in pop-up and is visible only if an instance is not running.

It is enough to indicate buttons using icons and describe those icons with a tooltip when the mouse hovers over the icon.

C.3.3 Single Instance View Page

This page should contain two tabs — Info and Edit.

C.3.3.1 Info Tab

Displays all attributes of an instance's model, contains buttons for restarting, starting/stopping (depending on whether the instance is running or not) and deleting of an instance.

If the layout from the original wireframes will be preserved, it is necessary to incorporate the following recommendations based on test findings:

1. It is sufficient to indicate control actions (start/stop/restart) with icons just as in Instances Listing Page and it should be required to confirm actions in the pop-up.
2. The button for removing an instance should be in the bottom right corner under both columns and it should be required to confirm the action in the pop-up.
3. To the listed attributes, the number of configured IN/OUT interfaces should be added.
4. The table with configured interfaces should be split into two tables according to the direction (IN and OUT).
5. In the table, the row of the selected interface (whose values are listed) should be highlighted.

C.3.3.2 Edit Tab

This tab contains a form to edit all attributes of *instance* model. The following text describes the specific behavior of the edit form:

1. *module* attribute, which is HTML `<select>` from available modules, the next one is *name*, which is generated from the module name by adding "`-instNUMBER_OF_NEXT_FREE_INSTANCE`", if it is not filled in yet (e.g. when *module* is changed). Example: A module has *name* IPFIXCOL, so the `IPFIXCOL-inst1` is generated.
2. An attribute *use-sysrepo* is displayed only in case the module selected for the instance has *is-sysrepo-ready* set to *true*. If *use-sysrepo* is checked, attribute *params* will not show up.

Following are the buttons with actions for the whole edit tab:

Attribute *name*	Attribute *type*	Action button for removal of the interface — only available in case instance's module has *in-ifces-cnt/out-ifces-cnt* attributes set to variable using asterisk (*).

Table C.2: Schema of how the content of rows in the table of interface on an edit instance page should look like.

- **Reset values**: resets an instance to a state, in which it was loaded, when the user came to the page (even interfaces will be renewed/restored)
- **Save instance**: sends an adjusted instance to the server, requires confirmation of restarting of an instance in a pop-up

C.3.3.2.1 Interfaces are going to be listed and at the same time adjustable only in case *trap-monitorable* in a module of an instance is set to *true*. Then they are going to be listed in two tables according to their direction (IN/OUT) similarly to the original version. The row of the selected interface should be highlighted and there should exist the ability to change an order of the rows. As in every table on this web, there should be a fixed table header, when the user is scrolling. Schema of the table is in table C.2.

If a module has *in-ifces-cnt/out-ifces-cnt* attributes set to a variable using an asterisk (*), it means that in that direction it is possible to configure a variable or an unspecified number of interfaces and, therefore, at the end of the table there should be rows with a button along with the icon + and label to add a new interface.

In case attributes *in-ifces-cnt/out-ifces-cnt* contain a number value, it means the user needs to configure an exact number of interfaces of the given direction. The GUI should react to this fact by rendering exactly the same number of prefilled interfaces so that the user does not have to add them by himself.

An interface has the three main attributes common for all types of interfaces:

- **Direction**: IN/OUT
- **Type**: TCP, TCP-TLS, UNIXSOCKET, FILE, BLACKHOLE
- **Name**: Unique identificator across the directions and types within one instance. If the user himself does not change it, it will be derivated automatically according to the type, as described in table C.3. Incremental ID is in the brackets in the table, because it should be used only in case the name is not unique (but that should not happen very often).

Attribute *type*	**Derived attribute** *name*
TCP	`t:`*`port`*`(-INCREMENTAL_ID)`
TCP-TLS	`T:`*`port`*`(-INCREMENTAL_ID)`
UNIXSOCKET	`u:`*`socket-name`*`(-INCREMENTAL_ID)`
FILE	`f:`*`file-name`*`(-INCREMENTAL_ID)`
BLACKHOLE	`b(-INCREMENTAL_ID)`

Table C.3: Schema of interface name derivation.

SELECT COLUMN	Attribute *name*	Attribute *trap-monitorable*	Attribute *use-trap-ifces*	Attribute *is-sysrepo-ready*	CONTROL COLUMN

Table C.4: Schema of how the content of rows in the table on a modules listing page should look like.

An example of a derived name for UNIXSOCKET type with *`socket-name`*`=flow_data` is: `u:flow_data`.
Other interface's attributes except the main ones mentioned above either differ in type or direction, or they are not that important and it is enough to list them in the end. Whether each attribute is supposed to be displayed is described in YANG model in GitHub repository of the NEMEA Supervisor[60]. Buttons with actions for interfaces:

- **Reset values**: reset a selected interface to values, which were there when the user came to the page

C.3.4 Create Instance Page

It basically contains Edit Tab from Single Instance View Page. The only difference is that in the forms there are no data filled yet, those are prefilled after the user takes some actions, for example, selects a module to use.

C.3.5 Modules Listing Page

Here, the table with the modules listing, defined by the following schema in table C.4, should be situated. There should also be a button for adding a new module, which should lead to Create Module page. The table should have a fixed header, so it will stay in place if the user scrolls down. Also, it should have an effect when the user hovers over a row that colors the background of a current row and in case the user clicks on the row, it should take him to Single Module Page View.

[60]`https://github.com/zidekmat/nemea-supervisor-sysrepo-edition/blob/master/yang/trap-interfaces.yang`

C.3.5.1 SELECT COLUMN

Contains a checkbox in the table header for selection of all rows and on each individual row contains another checkbox for single selection. If at least one row is selected, delete action should be displayed on the page, which might be called for all selected rows. Action with the rows will require confirmation in the pop-up.

C.3.5.2 CONTROL COLUMN

Contains buttons for:

- **Module removal**: Requires confirmation in pop-up notifying user that all instances of the module on the server would be stopped.
- **Module editing**: Serves just as a link to Single Module Edit Tab

It is enough to indicate buttons using icons and describe those icons with a tooltip when the mouse hovers over the icon.

C.3.6 Single Module View Page

This page should contain two tabs — Info and Edit.

C.3.6.1 Info Tab

Displays all attributes of a module's model, contains a button for deleting a module and shows a list of all its instances. The list of instances is a simplified table as in Instances Listing Page, where, however, are only columns with an instance's *name*, a state of an instance as described in table C.4, and instance's CONTROL COLUMN. If the layout from the original wireframes will be preserved, it is necessary to incorporate the following recommendations based on test finding:

1. Move a button for removing a module to the bottom right corner of the screen.

C.3.6.2 Edit Tab

This tab contains a form to edit all attributes of *module* model. The specific information on when the attributes should be rendered are described in the YANG model in GitHub repository of NEMEA Supervisor[61].
Following are the buttons with actions for the whole edit tab:

[61] `https://github.com/zidekmat/nemea-supervisor-sysrepo-edition/blob/master/yang/nemea.yang`

- **Reset values**: resets a module to a state, in which it was loaded when the user came to the page
- **Save module**: sends new data to the server, the user is asked before that, if he is sure about his action since a change of module means a most likely restart of its instances

C.3.7 Create Module Page

Contains basically Edit Tab from Single Module View Page. The only difference is that in the forms there are no data filled yet.

Appendix **D**

Contents of Enclosed CD

- `readme.txt` .... a file with CD contents description
- `src` ... the directory with the source codes of NEMEA Supervisor and its GUI
- `thesis` ... the directory of LaTeX source codes of the thesis and generated PDF file