Network Topology Detection & Web-UI for the Configuration of Real-Time Traffic

TIGER: **T**opology **I**dentifier **G**enerating **E**xtraordinary **R**esults

1st Marcin Bosk
Technische Universität Berlin
Berlin, Germany
marcin.l.bosk@campus.tu-berlin.de

2nd Lars Lange
Technische Universität Berlin
Berlin, Germany
lars.lange.1@campus.tu-berlin.de

3rd Jörn Strömsdörfer
Technische Universität Berlin
Berlin, Germany
joern.stroemsdoerfer@campus.tu-berlin.de

***Abstract*—With Time Sensitive Networking (TSN) Ethernet becomes vendor independently "real-time" capable. The mechanisms behind TSN do however increase the network complexity with a related extension of the needed configuration effort. For this purpose there needs to be knowledge of the network topology to configure the network devices accurately for the "real-time" traffic.**
TIGER ("Topology Identifier Generation Extraordinary Results") is a software project for detecting, displaying and editing a network topology with an additional feature for the configuration of real-time traffic, that is deployable as a complete solution to a Raspberry Pi running Raspian, i.e. the "TIGERBox".
Consisting of a back- and frontend, the detection of a network topology takes place on the server-side of the solution with a Python script. This queries LLDP information from nodes in the network via SNMP and thereby internally builds a graph representation with NetworkX. The script itself is run as a child process of a NodeJS Express server, which is handed the graph in a JSON file after completion of the detection. The server can then deliver this file to a frontend using a REST API. The frontend displays the graph information with D3 and provides an elegant interface, crafted with Bootstrap, for the modification of the network as well as the configuration of flows.

***Index Terms*—TSN, real-time networking, Ethernet, time-sensitive, LLDP, SNMP, MIB, OID, Raspberry Pi, Raspian, PM2, switches, multi-bridges, network, topology detection, Python, pySNMP, NetworkX, NodeJS, Express, REST, JSON, XHR, AJAX, jQuery, Bootstrap, PopperJS, ToastJS, D3, graph, nodes, links, flows, traffic, TIGER, TigerBOX**

I. Introduction

Ethernet is finally becoming vendor independently "real-time" capable due to Time Sensitive Networking (TSN) and offers new application scenarios in a wide range of industries ranging from the aerospace to automotive industry. As TSN puts more complexity on the network it becomes even more important to be able to easily configure the network to be "real-time" capable. A central node was introduced to configure the network from there, but for this it will need to know the network topology and this is where our project comes in. TIGER stands for Topology Identifier Generation Extraordinary Results and is a fine piece of software. Not only does it detect the network topology using a python script which mobilizes LLDP and SNMP to detect the neighbors of a device and the neighbors of its neighbors and so on, but it also offers a graphical representation of the graph in an Web-UI. This UI is based on latest web frameworks and utilizes the clients computing power to represent the graph using D3 and allows it to graphically modify the graph and save it for later uses with the REST API.

Starting point of this paper is to provide the reader with detailed background information about our iterative development process, the pains and gains we had trying to setup a network using emulated, simulated and in the end physical networks. The reader is presented a detailed report as well of how to replicate everything step by step to be able to run their own network topology detection.

The outcome of this development is a completely functioning network topology detection based on LLDP and SNMP and a graphical user interface to modify the resulting graph and provide it via a REST API to other services. Our innovative invention of the TIGERBox offers an all-in-one solution to quickly detect a network topology that already utilizes LLDP and SNMP. Using the ARM architecture we offer a low cpu and memory footprint and an affordable end product. This work can be seen as a starting point for further development using other network management protocols or to be used as a microservice for TSN to detect the topology for the central node.

First, in the chapter "Technologies" we will briefly cover the technologies we used. The next chapter will cover the how to get started guide in-depth and give you a perfect step by step tutorial of how to reproduce our results by setting up your own network based on the modified Raspbian image. The architectural chapter is split into 3 subsections covering in detail the frontend, backend and the python script for the actual topology detection. The development process gives a great overview about our iterative development and the odyssey we went through to get it working. Finally, we come up with a conclusion as well as a brief outlook based on our development we have done.

II. Technologies

In the following chapter, we will briefly go through the technologies we used and describe them to give you an overview about the technological aspects of the project.

A. LLDP

The Link Layer Discovery Protocol (LLDP) is a reliable method to determine the exact network topology of a connected network. LLDP uses its own Management Information Base (MIB) based on the standard introduced by the Internet Engineering Task Force (IETF). The MIB contains the information about the local LLDP agent but also the information about the detected neighbors agent. LLDP packets are usually sent in an interval of 30 seconds on all physical interfaces and can be seen as one-way function using multicast. The information saved in the MIB can be requested with the simple network management protocol (SNMP).

Based on information provided by DATACOM Buchverlag GmbH[4].

B. SNMP

The Simple Network Management Protocol (SNMP) allows a central network management of different network components. It allows to change parameters on a device in the network but also supports the monitoring of events or requesting certain information about a device. Similar to LLDP it comes along with an agent on the device as well and offers the information in the form of MIB.

Based on information provided by DATACOM Buchverlag GmbH[5].

C. Python

Python is a very universally applicable language and has a strong emphasis on the correct indentation of code instead of relying on brackets. It was first introduced in 1989 and is to this day still one of the most popular programming languages out there[8]. Python is quick to pick up and is perfect for prototyping applications and has a variety of libraries that support one's development flow.

D. pySNMP

pySNMP[1] is an SNMP library for python and comes along with support for v1/v2c/v3 of SNMP. It allows communicating to the SNMP agent of remote devices and extracting MIB information but can also modify SNMP data. The default implementation is built on synchronous communication but support is given for asynchronous communication libraries in case this way is preferred.

E. NetworkX

NetworkX[2] is a graph library for python and is often used in the context of networks as well. It is a very scalable framework and allows the analysis of social networks and other networks for specific metrics. The API offers simple management and creation of graphs and comes with inbuilt support for JSON output to easily use it for a web application.

F. JavaScript

JavaScript is a scripting language from 1995 and nowadays the most popular language for web development[6]. The so-called ECMAScript is the standardization for JavaScript and is still heavily developed and currently in the 10th edition. It was initially created to make websites more dynamic by allowing them to load content after the HTML and CSS were loaded or to manipulate the HTML. Nowadays JavaScript is heavily used for client and server programming as it creates a nice synergy to stay within one programming language.

G. Node.js

Node.js[3] is an event-driven and asynchronous JavaScript runtime, which is based on the v8 Engine. The v8 engine was developed by Google and supports basically most platforms natively. It is an implementation of JavaScript and uses just-in-time compilation to increase performance during runtime. As the JavaScript is executed in its own thread it would block I/O operations, but libuv was developed especially for Node.js to allow non-blocking I/O operations. It works by outsourcing the task to the kernel which then opens up another thread to execute the task. As soon as the task is done a callback is executed that tells the Node.js thread that the operation is done.

This mechanism is controlled bt the event loop which takes care of non-blocking I/O operations. It is divided into 6 phases that execute in row timer, I/O operations, and javascript callbacks.

H. NPM

The Node Package Manager[4] (NPM) allows installing modules that can either be written in JavaScript or C++ which allows using native functions and protocols. NPM plays an important part in the world of Node.js as most projects are heavily based on the usage of other modules.

[1] https://github.com/etingof/pysnmp
[2] http://networkx.github.io
[3] https://nodejs.org/
[4] https://www.npmjs.com/

I. Express.js

One of the most popular packages in the world of Node.js is Express.js[5]. It is an HTTP server framework and is directly based on the native HTTP module of Node.js which makes it quite performant.

Express.js is a supercharged framework and comes with a lot of inbuilt functions. One of them is that it automatically supports different types of templating engines like pug, ejs or handlebars. On top of that, extra modules exist for Express.js like support for cookies or compression which have to be installed through NPM again.

The most useful use case for Express.js is to use it as a backend server to provide some sort of application programming interface (API) to the outside world. These are provided as so-called routes in Express.js.

J. REST

REST stands for REpresentational State Transfer and describes how systems have to communicate with each other, in this case, it is a specification for an API. The information was partly derived from personal experience and a very good dissertation about architectural styles[2].

REST is based on six architectural constraints which are as follows:

1) **Client-Server-Modell** - it requires a client-server modell where the user interface is abstracted from the actual data. This shall allow the application to scale easily.
2) **Statelessness** - client and server have to be stateless, this means that the client has to send all required information with each request. This brings scalability, visibility, and reliability along as each server is capable of serving the request.
3) **Caching** - to increase network efficiency client and server are able to save previously sent messages. Server-side caching is not that typical in production environments but client-side caching is very useful for offline applications.
4) **Unified Interface** - the service provides a general and unified API that is encapsulated from the actual implementation and is not adjusted to any special requirements of an application.
5) **Layered System** - when talking to a REST API it might not always be clear whether it is already the actual endpoint or whether there are multiple layers in between. This is very typical for REST and is perfect for microservice architectures.
6) **Code-On-Demand** - this is optional and not met that often in the real world. Essentially instead of returning a resource, it returns executable code that the client can then execute.

A REST service usually supports the following methods GET, POST, PUT/PATCH and DELETE.

- **GET** - requests a resource from the server.
- **POST** - sends data to the server.
- **PUT/PATCH** - changes existing files on the server.
- **DELETE** - delete an existing file on the server.

K. JSON

While XML is still a big thing nowadays for Java development it just couldn't really keep up with the requirements of web development. This is where the JavaScript Object Notation (JSON) plays a big role. JSON is a file format that is typically used for transferring information from frontend to backend and vice versa. The big advantage of JSON are that it is natively supported by every modern browser and that in case your backend and frontend are both written in JavaScript/Node.js there is no need to transform data back and forth to fit required standards and therefore allow quick development of applications. All basic data types are supported. JSON is based on a key-value store and is a perfect fit for NoSQL databases[1].

L. PM2

When reaching the point where you want to deploy your application or in our case let it run automatically on the tiger box on startup you will most likely not get around PM2[6], an advanced, production process manager for Node.js. Other options would be docker together with kubernetes as orchestration, but to keep it simple and actually executable on a Raspberry pi we preferred the process manager. PM2 allows defining a list of applications that are supposed to be started on startup. Other mentionable options of PM2 are auto clustering and watch and reload, meaning you could even use PM2 in a development environment to replace a tool like nodemon that just watches and reloads the application on file change.

M. Bootstrap

Bootstrap[7] is a frontend framework initially developed by Twitter and nowadays a big opensource project on GitHub. It is probably by far the most popular out there and got widely adapted to Vue.js, Angular and React.js to provide ready to use components in other frontend frameworks. Bootstrap is a CSS frontend framework in particular and provides CSS classes for your applications.

N. D3

D3[8], which stands for Data-Drive Documents, is often called the godfather of all graphic libraries. It allows creating 2D and 3D graphics based on provided data and provides them as Scalable Vector Graphics (SVG). Similarities to jQuery are given as the manipulation of the Document Object Model (DOM) is very similar, but D3 relies on CSS selectors instead of class or id selectors. Data is often provided in the form of JSON or a CSV and therefore fits perfectly together with Node.js as backend as no data conversion is required. Nowadays it is often used as an underlying framework to create a simpler abstraction of the rather complex API.

[5]https://expressjs.com/

[6]http://pm2.keymetrics.io/

[7]https://getbootstrap.com/

[8]https://d3js.org/

O. jQuery

jQuery[9] is a JavaScript library that enables you to manipulate and read the DOM. As previously mentioned it shows similarities to D3 when it comes to the syntax in regards to the manipulation but it makes use of class or id selectors instead of CSS selectors. With jQuery, you can manipulate every bit and piece of the HTML but also add new elements or remove elements. While a lot of people want jQuery to be gone for good as frontend Frameworks like React.js or Angular are on the rise it is still used by 72% of the 10 million most popular websites[7]. One of the most popular features is that it comes along with Ajax and allows a simple communication to an API or request extra data to dynamically change the DOM and increase the user experience.

P. Ajax

Ajax is the short form for asynchronous JavaScript and XML. It enables you to create web applications with client-server communication to exchange data or retrieve data in the background without blocking the UI usage for the user and dynamically changing up the application with jQuery without the need to refresh the page. While in the beginning it was restricted to XML it is now most commonly used together with JSON.

Q. Raspbian

Raspbian is officially supported and maintained by the Raspberry Pi Foundation but is developed independently as well and comes along with some standard software like python and java for quick development purposes[3]. Raspbian is based on the Debian distribution and comes in two flavors, one being a desktop version and the other being a pure server version.

III. Getting Started Guide

In the following we present getting started guides which will let you set up a testing environment using Raspberry Pis 2 or 3 and also use the topology detection application we provide with this work. These will allow you to use a Raspberry Pi as a switch and run the topology detection server on it, or run our application server on your computer.

A. Setting up Raspberry Pi as a Switch

In order to use the Raspberry Pi 2 or 3 as a switch you need to fulfill several requirements:

- A Raspberry Pi version 2 or 3
- Appropriate microSD card for the Raspberry Pi
- A microSD card adapter
- Image of Raspbian Stretch Lite version from 2019.04.09[10]

An example of a setup that can be achieved when all the prerequisites are fulfilled is presented in Figure 1. The first step is to prepare the microSD by flashing it with Raspbian:

[9]https://jquery.com/

[10]available here: http://downloads.raspberrypi.org/raspbian_lite/images/raspbian_lite-2019-04-09/2019-04-08-raspbian-stretch-lite.zip

Figure 1. Photo of an exemplary switched network setup using Raspberry Pi 3 single board computers and multiple network adapters

1) Unpack the Raspbian image you downloaded
2) Flash the SD card with the Respbian Image. For MacOS you can use the following instructions in a terminal window of your choice:

```
$ diskutil list
$ diskutil unmountDisk /dev/<identifier of
    your microSD card>
$ sudo dd if=/<path to folder with
    raspbian image>/2019-04-08-raspbian-
    stretch-lite.img of=/dev/<identifier
    of your microSD card> bs=2m
```

3) Add an empty file to the boot partition on the SD card. The file should be called: ***ssh***.This will enable ssh connections to the Raspberry Pi.
4) Eject the card from your computer and insert the card into the Raspberry Pi

Next, in order to enable the Raspberry Pi as a switch you need to reconfigure it. The following steps will guide you on how to to it:

1) Share internet from your device with the Raspberry Pi
2) Connect to your Raspberry Pi. On your machine in a terminal of your choice use the following command:

```
$ ssh pi@raspberrypi.local
```

   The default password is ***raspberry***
3) When connected run the following command to refresh repository information:

```
$ sudo apt-get update
```

4) Then, in order to allow for bridging configuration, bridge-utils need to be installed. This can be done with the following command:

```
$ sudo apt-get install bridge-utils
```

5) With bridge-utils installed, the networking interfaces need to be configured so that the Raspberry Pi can function as a managed switch. To achieve that you must edit the ***interfaces*** file. Utilize the following command:

```
$ sudo nano /etc/network/interfaces
```

Replace the contents of this file with:

```
# interfaces(5) file used by ifup(8) and
    ifdown(8)

# Please note that this file is written to
     be used with dhcpcd
# For static IP, consult /etc/dhcpcd.conf
    and 'man dhcpcd.conf'

# Include files from /etc/network/
    interfaces.d:
source-directory /etc/network/interfaces.d

auto br0
iface br0 inet static
    address 192.168.2.42
    broadcast 192.168.2.255
    netmask 255.255.255.0
    gateway 192.168.2.1
    dns-nameservers 8.8.8.8
    bridge_ports eth0 eth1 eth2 eth3 eth4
    bridge_stp on
    bridge_waitport 0
    bridge_fd 0
```

Note: You can choose your own IP addresses conforming to all of the addressing conventions.

6) Install LLDPD and SNMPD using the following command:

```
$ sudo nano /etc/network/interfaces
```

7) To enable SNMP access from the outside you need to edit the ***snmpd.conf*** configuration file. You can do so using the following command:

```
$ sudo nano /etc/snmp/snmpd.conf
```

You need to change the following in this file:

- Comment all agent addresses out and instead insert:

```
agentAddress  udp:161
```

- Comment out:

```
rocommunity public default -V
    systemonly
```

Instead insert:

```
rocommunity public
```

- Check if AgentX support is enabled, i.e. if there is a non-commented line with:

```
master agentx
```

8) Edit the ***lldpd*** configuration file:

```
$ sudo nano /etc/default/lldpd
```

Change the contents of this configuration file to the following:

```
# Uncomment to start SNMP subagent and
    enable CDP, SONMP and EDP protocol
DAEMON_ARGS="-x -c"
```

9) Reboot the Raspberry Pi to make sure the changes take effect:

```
$ sudo reboot
```

The Raspberry Pi will be available for you to access using ssh under the IP you specified. The username is ***pi*** and the password is ***raspberry***.

B. Setting up Raspberry Pi as a TIGERBox

Figure 2. Photo of the TIGERBox created during the project

The Raspberry Pi can also be used as a TIGERBox (see Figure 2) - an integrated solution for network topology detection. To use it in this way, first you need to fulfill several requirements:

- Successful completion of the "Setting up Raspberry Pi as a Switch" guide (see III-A)
- Internet access on the Raspberry Pi you want to use as TIGERBox

Next, in order to configure the Raspberry Pi as a TIGERBox you need to complete the following steps while being connected using ssh with the Raspberry Pi:

1) Update the repositories:

```
$ sudo apt-get update
```

2) Install python3:

```
$ sudo apt-get install python3
```

3) Install Git:

```
$ sudo apt-get install git
```

4) Install pip3:

```
$ sudo apt-get install python3-pip
```

5) Fetch pySNMP Python library:

```
$ pip3 install --trusted-host pypi.org --
    trusted-host files.pythonhosted.org
    pysnmp
```

6) Fetch Python NetworkX library:

```
$ pip3 install --trusted-host pypi.org --
    trusted-host files.pythonhosted.org
    networkx
```

7) Install nodejs and pm2:

```
$ curl -sL https://deb.nodesource.com/
    setup_12.x | sudo -E bash -
$ sudo apt-get install -y nodejs
$ sudo npm i -g pm2
```

8) Clone the git repository with the topology detection application:

```
$ git clone --single-branch --branch
    student https://gitlab.fokus.
    fraunhofer.de/openiotfog/topology-
    detection.git
```

9) Install all necessary node packages:

```
$ cd topology-detection/app
$ npm install
```

10) Set up the application and make it start automatically on system startup:

```
$ TIGER_BOX=true BEST_IP=192.168.2.42 pm2
    start ~/topology-detection/app/app.js
$ pm2 startup
!-- Follow the instruction that appears on
     the screen --!
$ pm2 save
(optional) $ pm2 restart app
```

Upon successful completion of the above steps, the application can be accessed at http://<IP you chose for TIGER-Box>:3000/.

C. Running topology detection natively on your computer

In order to run the topology detection application natively on your computer you first need to consider a few requirements:

- Python3 with pip3 installed
- NodeJS

Next, the application can be prepared and turned on in just a few steps. Commands should be executed using a terminal of your choice.

1) Fetch pySNMP Python library:

```
$ pip3 install --trusted-host pypi.org --
    trusted-host files.pythonhosted.org
    pysnmp
```

2) Fetch Python NetworkX library:

```
$ pip3 install --trusted-host pypi.org --
    trusted-host files.pythonhosted.org
    networkx
```

3) Clone the git repository with the topology detection application:

```
$ git clone --single-branch --branch
    student https://gitlab.fokus.
    fraunhofer.de/openiotfog/topology-
    detection.git
```

4) Install all necessary node packages:

```
$ cd topology-detection/app
$ npm install
```

5) There are two ways you can start the application. Firstly, you can make it run on the real network and discover the real network topology:

```
$ TIGER_BOX=true BEST_IP=<IP of a node
    that supports both SNMP and LLDP in
    the network you are trying to discover
    > node app.js
```

Or you can run it with a mock network topology locally (recommended for local development):

```
$ node app.js
```

Upon successful completion of the above steps, the application can be accessed at http://localhost:3000/

IV. Architecture

The architecture of this solution is manyfold, with components on the server- and client-side. This chapter will firstly give an introduction into the usage and architecture of the frontend, which is easily accessible with a web browser. Afterwards this chapter will provide an in-depth look at the backend architecture with an additional deep dive into the Python script responsible for the network topology detection.

A. Frontend

To provide a complete insight into the concept of the frontend of this solution, following will firstly be an introduction into the general usage paradigm with appropriate screenshots and secondly a look at the architecture of the frontend, namely its components, their respective responsibilities and their interworkings.

1) Usage: Before going into explanations regarding the inner workings, the components and their collaboration of the frontend, it is advantageous to briefly describe the intended use of the web client, i.e. its features.

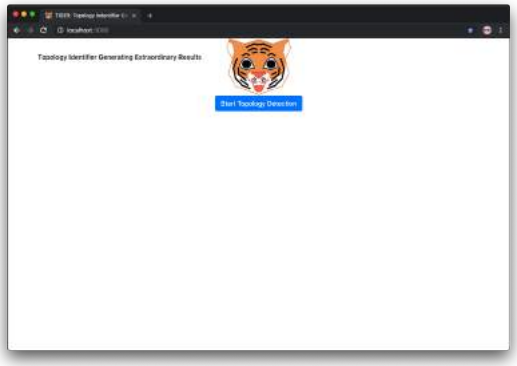

Figure 3. Screenshot 1: Start Page

When launching the web client, this is the start page you are being taken to. In case you have run the topology detection before and not either reboot the server or reset the web client, you will get taken straight to the topology (please see Figure 5), which the server conveniently stored for you with any personal configurations you might have perfomed.

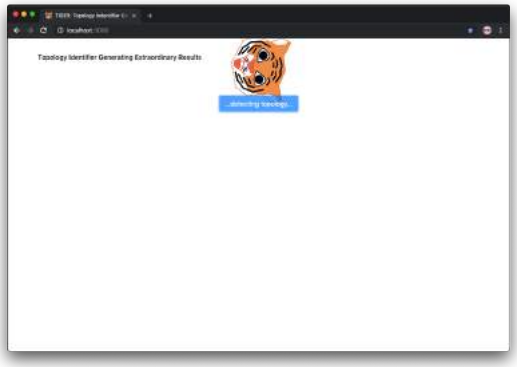

Figure 4. Screenshot 2: Waiting for Topology Detection

During the time of network traversal, which might vary according to the network size, you can watch the TIGER logo spin.

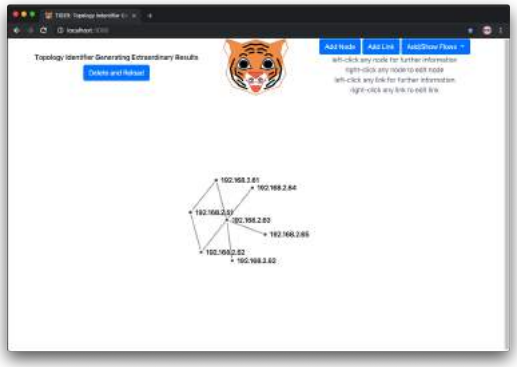

Figure 5. Screenshot 3: Topology

From the start page via the wait page the successful fetching of the network graph JSON lands you on the main page and interface of the web application. As mentioned below Figure 3, this is where you would be taken directly by accessing the server if there is a graph stored on the server-side.

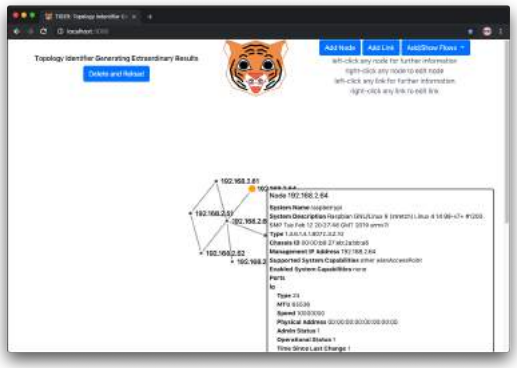

Figure 6. Screenshot 4: Tooltip with Node Information

Above you can observe on the one hand nodes being highlighted on *mouseover* and on the other hand the tooltip containing all available data on the node, which is available through a *leftclick* on any node.

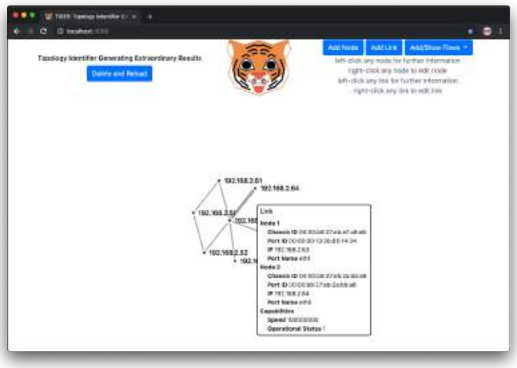

Figure 7. Screenshot 5: Tooltip with Link Information

Similarly to nodes, links are also being highlighted on *mouseover* and provide a similar information tooltip on a *leftclick*.

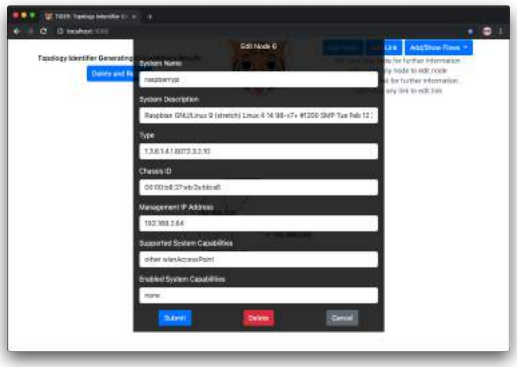

Figure 8. Screenshot 6: Overlay for Adding/Editing Nodes

A *rightclick* on any node opens the overlay for editing the respective information of that node. The field containing information on the **Management IP Address** is what is used by D3 to distinguish nodes in the graph.
The overlay that is accessible via the button **Add Node** in the upper right-hand corner is the very same as for the editing of nodes.
Any information that is either input or edited in the given forms at the discretion of the user is not checked by the front- or backend for its validity and always stored for the user in firstly the web client and, after submission, on the server-side.

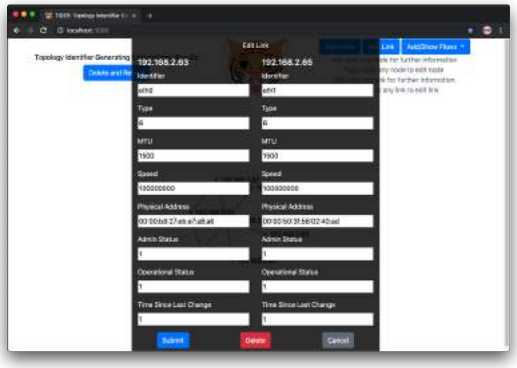

Figure 9. Screenshot 7: Overlay for Adding/Editing Link

The overlay for editing links is similarly available by a *rightclick* onto links.
The overlay that is accessible via the button **Add Link** in the upper right-hand corner is the very same as for the editing of links.
The aforementioned considerations for the content of the data fields applies in the same way.

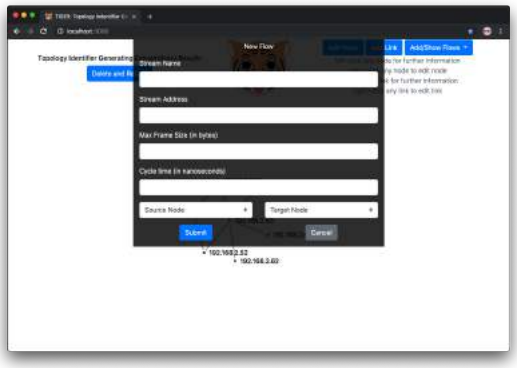

Figure 10. Screenshot 8: Overlay for Adding/Editing Flows

Via the dropdown menu under **Add/Show Flows** one can add information of traffic streams with the same coherent overlay.
The aforementioned considerations for the content of the data fields applies in the same way.

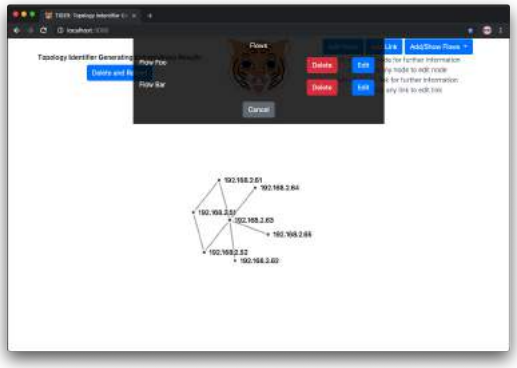

Figure 11. Screenshot 9: Overlay for Showing List of Flows

Also accessible via the dropdown menu under **Add/Show Flows** is an overlay with a list of created streams.

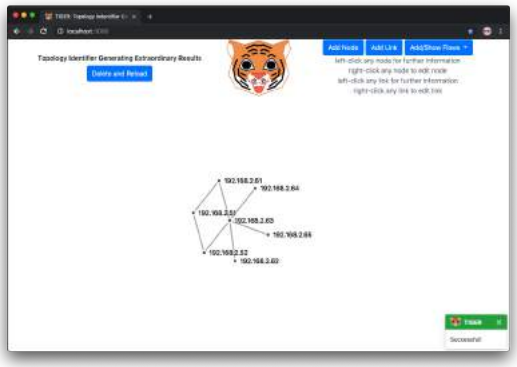

Figure 12. Screenshot 10: Toast with Success/Error Message

The toast message with a little TIGER logo in the lower right-hand corner, displaying either the successful or failed transfer of information on nodes, links or flows to the backend adds the final touch to the user interface.

2) Architecture: The following is a look at the inner workings and the components and their collaboration of the frontend.

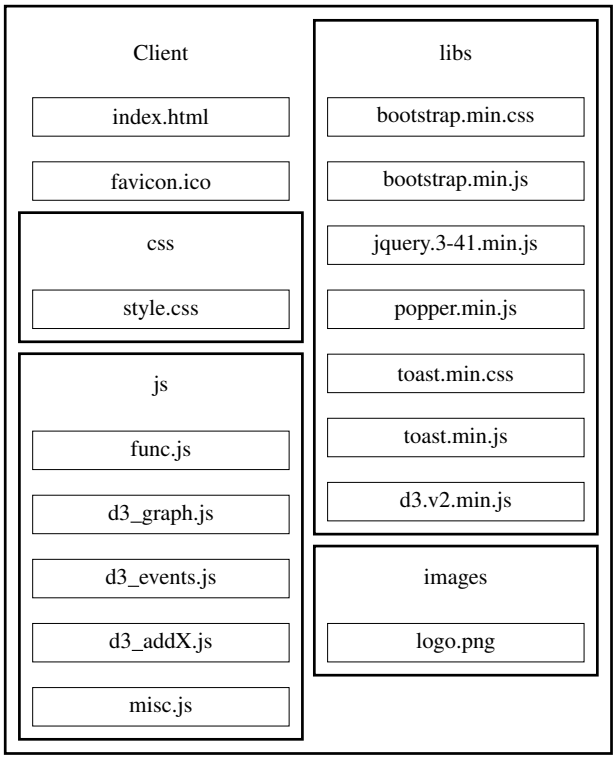

Figure 13. File Structure for Client

What one can immediately see from Figure 13 and the number of HTML files, is that the frontend was conceived and realized as a single-page web application. Also apparent from the listed library files is the fact, that we did not use a framework to accomplish this. The reason for this being, that with the main content of our page being the D3 graph, we did not see an advantage in proceeding on the goal of a single-page web app with a framework, but rather a disadvantage in the potential additional workload of harmonizing D3 with any given framework. Therefore all dynamically available content of the web application surrounding the graph was either modified using visibility tags or dynamically added to the HTML of the web page from JavaScript.

The libraries, besides D3 for all graph-related tasks, were Bootstrap (please see Chapter II-M.) for the style of the web application, which depends on jQuery (please see Chapter II-O.) and PopperJS, and ToastJS for notifications (please see Figure 12). The tooltips observable in Figures 6 and 7 and all overlays of Figures 8-11 were all manually designed and provided with `style.css`.

With the JavaScript files `func.js` is the first one being utilized and the one kicking off the topology detection on the server-side and afterwards receiving and parsing the graph JSON. From there `d3_graph.js` takes over in drawing the graph with D3 and giving its nodes and links attributes, that can later be utilized for displaying additional information or dynamically highlighting specific parts of the graph. `d3_events.js` is the component handling all the events for *mouseover*, *leftclick* and *rightclick* on predefined sections of the graph. Beyond that it dynamically composes the HTML code for tooltips and for overlays in the case of editing of nodes or links. Similarly `d3_addX.js` takes care of all overlays for the creation of nodes or links or the creation, editing or showing of flows, i.e. all functionality of the web app available through the buttons in the upper right-hand corner (please see Figure 5. No matter the originating JavaScript, all communication with the backend after the initial request for the graph is handled by `misc.js`. This is also the component responsible for briefing the user on success and error messages (please see Figure 12).

B. Backend - API

For the communication to the frontend, we built a REST API that is based on the previously mentioned technologies REST, Node.js, Express, and other not further mentioned sub-components. The advantage of using Node.js is that it allows us to stay within one language and avoiding any further data conversion.

The REST API uses a versioning scheme so that in case the project is being developed further in the future it can be distinguished between version one, two, and so on and the client related to version one won't break due to new features. Our REST routes are divided into the graph API and the flows API.

1) REST API example of the graph API: First of all, we keep the graph in memory to keep a low memory print for the underlying device, e.g. Raspberry Pi.

```
let savedGraph = {};
```

The GET or FIND request will return the JSON that we keep in memory.

```
router.get('/graph', (req, res) => {
    return res.send(savedGraph);
});
```

The POST route is a bit more complicated but is essentially the heart of our backend as it starts a python script that is further explained in the upcoming section [IV-C]. As we have previously heard in the technology chapter Node.js runs in its own thread and allows us natively to create sub-processes, so-called child processes, that can run any command on the underlying system and listen to standard output and error. We make use of it to create a child process that runs the python script [IV-C] and listen to the standard output which in our case is the complete JSON of the detected network topology. As the output is too big we have to buffer the data and wait for the child process to finish executing, afterward we can parse the JSON and send it to the frontend which then uses it further [IV-A].

The "__dirname" allows us to run the script platform-independent as it returns the current directory that from where it was called and as the file structure doesn't change within

our system we can easily determine where the python script is residing. Another important aspect is that we make use of environment variables that allow us to be a bit more flexible during startup and runtime of the application. For development purposes, we require no further environment variable to quickly start up the application and make use of a different python script that returns a mock JSON which is based on a previously detected network topology using our raspberry pis.

In case the environment variable "TIGER_BOX" is defined, any value will satisfy, the child process will execute the python script [IV-C] but also requires the "BEST_IP" environment variable to be defined to an IP address of the remote device that shall be used as an access point to the network.

As soon as a POST request is received the previously explained flow will be executed and the parsed JSON will be saved in memory and sent to the frontend as well.

```
router.post('/graph', (req, res) => {
    let scriptOutput = "";

    let pyScript = spawn('python3',
     ↪ [`${__dirname}/../../../python
     ↪ /readFile.py`]); // mock

    if (process.env.TIGER_BOX)
        pyScript = spawn('python3',
         ↪ [`${__dirname}/../../../python
         ↪ /getSnmpInfo.py`,
         ↪ process.env.BEST_IP]); //Run
         ↪ on tigerBox

    pyScript.stdout.on('data', data => {
        data=data.toString();
        scriptOutput+=data;
    });

    pyScript.stderr.on('data', data => {
        res.status(500).send(data);
    });

    pyScript.on('close', (code) => {
        scriptOutput =
         ↪ JSON.parse(scriptOutput);

        savedGraph = scriptOutput;
        res.send(scriptOutput);
    });
});
```

The PUT or PATCH route is rather kept simple and just replaces the locally kept JSON as we rely on the frontend to do the proper work.

```
router.put('/graph', (req,res) => {
    savedGraph = req.body;

    res.send(savedGraph);
});
```

The DELETE route sets the in-memory kept variable to an empty JSON and allows the frontend to redetect the network topology.

```
router.delete('/graph', (req, res) => {
    savedGraph = {};

    return res.send('Deleted resource!');
});
```

The flows API is kept super simple compared to the graph API and follows a similar scheme but uses the POST like the PATCH and just updates the JSON containing all the flows.

2) Overview: In the technology part, we mentioned that Express is already a supercharged HTTP framework for Node.js but in some cases, some extra packages are needed to complement it. We use the "body-parser" package to be able to parse incoming requests like POST/PATCH that actually contain data to immediately be interpreted as JSON. On top of that, we use Express as well to deliver our frontend so it contains actually the backend routes but also provides the files for the frontend by defining a static directory.

To give a quick overview of what exactly is offered we have environment variables.

- **PORT** allows to switch the port that the application listens on. Default is 3000.
- **TIGER_BOX** switches to the getSnmpInfo python script. Default is undefined.
- **BEST_IP** determines which device to use for the detection and is needed in case TIGER_BOX is defined. Default is undefined.

Our REST API routes are as follows:

- **GET** /v1/graph - returns graph
- **POST** /v1/graph - starts topology detection, body empty
- **PUT** /v1/graph - body contains graph JSON
- **DELETE** /v1/graph - deletes graph
- **GET** /v1/flows - returns flows
- **POST** /v1/flows - body contains flows JSON and replaces in-memory
- **PUT** /v1/flows - body contains flows JSON and replaces in-memory
- **DELETE** /v1/flows - deletes flows

C. Backend - Python Script

In order to discover the network topology, we decided to utilize a script written in Python that queries all discoverable network nodes for any LLDP information they possess. It concatenates the collected information about the links and nodes and creates a NetworkX graph of the network. Finally, it exports the graph as a JSON file for later use at the web server.

To achieve the desired functionality of the network topology detection script, we require a few libraries, most notably the pySNMP library. It is a full Python implementation of the SNMP protocol featuring SNMP Agent, Master and Proxy roles. In our case we are using the Master capability of the engine to send snmpget messages to the network nodes we want to ask for the LLDP information they posess.

To construct the output graph, we are using the NetworkX and JSON libraries. NetworkX is a package that enables us to efficiently create and manipulate network and graph structures. JSON package serves the purpose of exporting the acquired information in the proper JSON format.

The network topology is discovered in a recursive manner. As shown in Figure 14, when the script is called by the JavaScript backend function, it firstly checks whether the IP that specifies the first node to query is valid. Then, it initiates the ”Run detection” algorithm which takes the IP address received by the script and passes it to the ”Ask IP” function. It prepares space for soon-to-be-obtained information and passes the IP further to run the ”Run SNMP on IP” algorithm. This function sends SNMP requests to the specified IP. The queried information consists of the following OIDs (Object Identifiers):

- **LLDP-MIB** (*1.0.8802.1.1.2.1.4*) - SNMP Link Layer Discovery Protocol Management Information Base
- **IF-MIB** (*1.3.6.1.2.1.2.2*) - SNMP Interface Management Information Base
- **MIB-System** (*1.3.6.1.2.1.1*) - System Management Information Base

When a response is received the LLDP, interfaces and system information from queried node are extracted. The following parameters are saved:

- System name of the remote and queried node
- System description of the remote and queried node
- Chassis ID of remote and queried node
- Management IP address of remote and queried node
- Supported system capabilities of remote node
- Enabled system capabilities of remote node
- Type of queried node
- Interface information of queried node consisting of: Identifier, Type, MTU, Link Speed, Physical Address, Admin Status, Operational Status and Time Since Last Change

Furthermore, the nodes, newly discovered using information from LLDP-MIB, are added to results graph. If the queried node is present in the graph already, its information is updated. Should this device be absent from the graph, it is added to it using available partial information. The ”Run SNMP on IP” function also processes and prepares information about the queried node’s interfaces for later use.

This information is then passed back to the ”Ask IP” function which memorizes the IP that has already been queried, appends newly discovered IPs to unasked IPs list (if these are not already in asked IPs list) and collects newly discovered partial link information of the just queried node. The next step is where recursion happens. If there are still unasked IPs left, repeat the SNMP query process, otherwise pass the collected information to the main ”Run Detection” function.

In the next step the partial link information collected using obtained LLDP information is processed. During this process full links with all of the available information are constructed. Those are then converted into a JSON-like format and appended to the NetworkX graph. Next, the information on node’s interfaces is added to the nodes already present in the graph.

Finally, the graph is serialized, exported into JSON and printed to the console output. This digest is then captured by JavaScript backend that initially called the topology detection Python script.

V. Development Process

As one can infer from the previous chapters, the steps of development for this project presented as highly iterative. This became apparent to us very early on in the process, which is why we proceeded in a meticulously planned out fashion of sequential steps to achieve the goal of this project. Internally we thought of the process as the building of a garden. In the following chapter we will go into the steps we identified as our main milestones:

1) Building a Network
2) Detecting the Topology
3) Packaging the Topology
4) Displaying the Topology
5) Editing the Topology

After these steps we will have gone from just ploughing beds to planting flowers and building a gazebo in our *beautiful and harmonious garden*.

A. Building a Network

Figure 15. Our Garden: Step 1

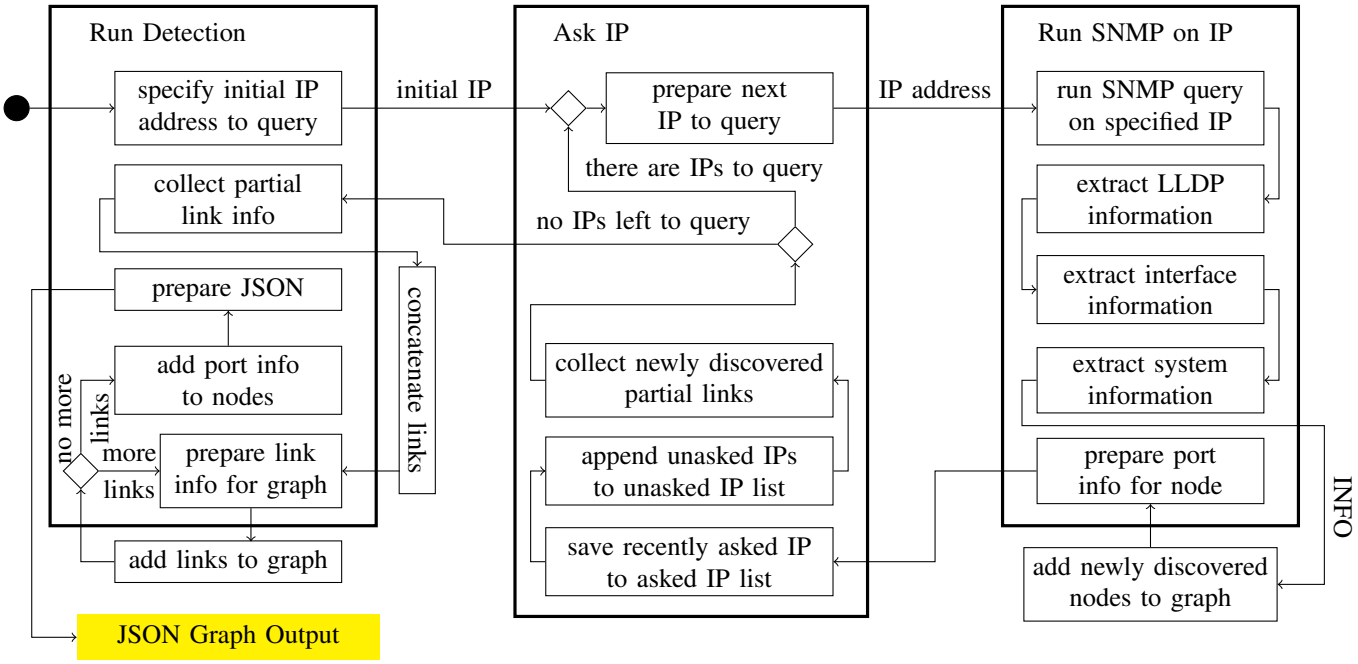

Figure 14. Functionality Diagram of the Python Network Topology Detection Script

This first of five steps was most definitely the most time-consuming one and presented the most frustrating dead ends. In eventually two sub-steps the goal here was to simulate, emulate or basically in any way build a network of participants that exchange the messages we would later in step two utilize in order to detect the topology of said network. Namely the participating nodes of the network would have to be able to act as multi-bridge switches, exchange LLDP messages (Link Layer Discovery Protocol), store the information of such received messages in the MIB (Management Information Base) of SNMP (Simple Network Management Protocol) and have the aforementioned information queryable with SNMP messages onto the appropriate OID (Object Identifier) of LLDP.

Our first attempt to achieve this extensive goal was to do it in software and either emulate or simulate the network.

Figure 16. Software for Emulation/Simulation of Network

We firstly tried to emulate the network using Mininet. We have managed to set up switches to which some hosts were connected. We were also able to turn LLDP deamons on all of the devices on and we could detect these messages being exchanged between all of the nodes. The problem arose when we tried to query the LLDP information using SNMP. Due to Mininet having a single namespace for all of the switches, we couldn't distinguish between those using SNMP. Also, the way LLDP and SNMP is implemented within Mininet, it is not possible to query the LLDP information using SNMP, as this information is not placed in the MIB of the nodes.

Next, we turned our look to the possibility of using a simulation environment to build our test network. Initially we tried using NS3/GNS3 to simulate the network, we were however unsuccessful in doing so due to lacking support of LLDP cooperation with SNMP in software that was available to us. This prompted a look at Omnet++, with which we were also not successful with due to lack of support for both SNMP and LLDP.

We were also prompted not to use simulation, as it is not running in real time. This means, every application we would be trying to run using a simulated network would have to be either specially developed to run within the simulation or we would need to construct a special connection point to the network from the outside. Such connection point would require considerations as to how to handle real-time and non-real-time traffic together which is not in the scope of this work.

Our second attempt to build the network then considered an approach through hardware. For this purpose we were equipped with multiple SOCs (System on a Chip), namely Raspberry Pi Model B (Version 2, 3, and 3+).

With previously positive experience with the OS, we firstly naively installed Ubuntu 18.04 for RasPi on the microcomputers. We were confident from online tutorials on version 16.04 that we could configure the OS so that the Pis act as switches. Sadly Canonical had apparently changed the responsible internal services to configure the device as a switch, i.e. multi-bridge, going from 16.04 to 18.04 and we were therefore only able to bridge two Ethernet ports and could

never reach a further attached node with a simple `ping`.
We secondly tried to use the OS OpenWRT and were able to achieve significant results with it. The configuration of the Pis as switches was a breeze and we could beautifully observe LLDP messages being passed around neighbours. Sadly the `lldpd` package used in OpenWRT does not natively support SNMP. Therefore we could not query the MIB with SNMP with the appropriate OID for LLDP, since there was no LLDP object stored in the MIB.
In an intermediate attempt to simply get the functionality regarding LLDP and SNMP to work in any kind of fashion, we set up two Xubuntu 19.04 virtual machines on our computers with VirtualBox. This would later turn out to be the stepping-stone to a working solution, since we were here finally able to query LLDP information with SNMP. We afterwards took what we learned here as a guide to go back to the microcomputers with.

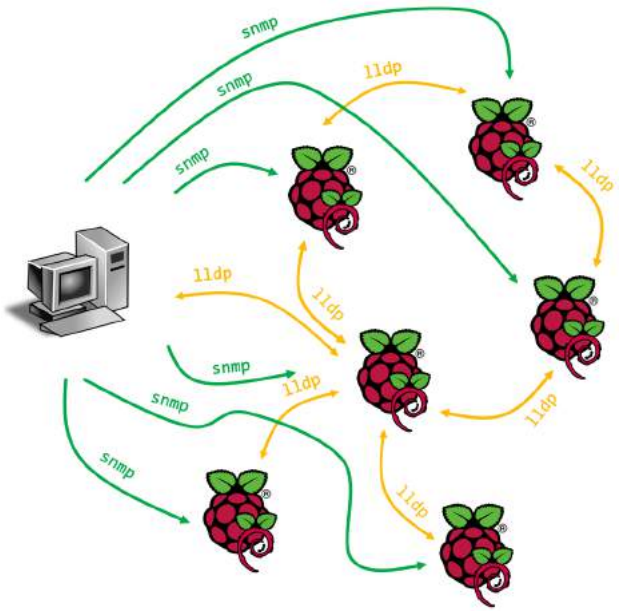

Figure 17. Raspberry Pis running Raspian w/ LLDP and SNMP

The final solution including all envisioned functionality was then the Raspberry Pis running Raspian Lite 2019-04-08, which were configured as switches using the package `bridge-utils` and LLDP and SNMP being available with the packages `snmpd` and `lldpd`.

B. Detecting the Topology

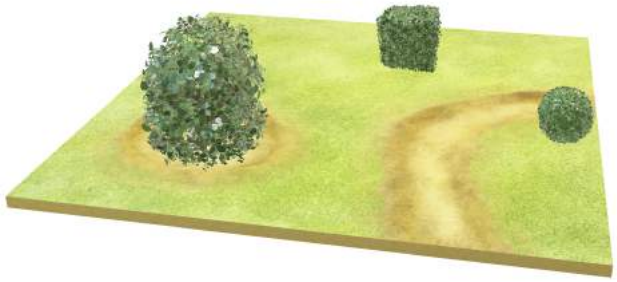

Figure 18. Our Garden: Step 2

With the excitement of having a working network - after the majority of the time allotted for the project having gone into getting it to work - we were quick to get from manually retrieving the LLDP information from the MIB with SNMP to having a Python script do the work for us. The choice of tools (please see Chapter III-C.) was clear given the prerequisite to the project of writing a Python script as the tool to detect the topology.
With the algorithm for the topology detection (please see Figure 14) and its inherent properties of recursion in discovering nodes and two-sidedness in discovering links, we were happy to see that there is a complete and full retrieval of all available LLDP information from nodes, when there is a network of at least two connected nodes. There can however naturally be no bilateral retrieval of information regarding links, if there is no link, i.e. only one node. Beyond this property there can also be no complete information retrieval, if the nodes in a network do not fulfill the required prerequisites of running the appropriate agents for LLDP and SNMP.
Given however a network of properly configured nodes we are happy to mark this step of the development process as complete. Such a network will be discovered as it is.

C. Packaging the Topology

Figure 19. Our Garden: Step 3

During the seemingly eternal work on step 1 ”Building a Network”, we at one point skipped ahead to this step 4. Without having any actual network set up to be detected we

did some brainstorming to consider a packaging structure for the topology. A mock network graph, drawn up with pencil and paper, and our knowledge of retrievable LLDP information served as a baseline for these considerations. Since we were aiming to display and edit the topology information in a web browser later on, the choice to use JSON as the file format was clear. The choice for the Python library NetworkX was also clear, because of the requirement to use Python for the topology detection.
What we also set up in advance during this side venture was a NodeJS Express server and with it HTTP request routes for a client to fetch the then mock JSON of a network graph.
Beyond the much needed motivation, these considerations helped immensely speed up the development process after that time that we got the network to work. Then we knew exactly what we required the topology detection script to output and already had the mechanisms set up to deliver the data to a web client.

D. Displaying the Topology

Figure 20. Our Garden: Step 4

As part of the excursion into the topology packaging, we also ventured into the exploration of possibilities to display graphs in a web browser. It only took little research to become convinced of the choice for D3 as the JavaScript frontend library for displaying data in 2D graphs of any kind. Since we had invested much time into considerations of the data structure of the graph topology at the time of only having a mock graph, the aforementioned side venture yielded basically an already final version of what the graph representation in the frontend would ultimately become.

E. Editing the Topology

Figure 21. Our Garden: Step 5

While our web client started out with a simple graph representation and only a functional design for the user interface, we were very keen to invest great creativity early on. This pertained especially to the user interface framing the graph and lead us to step by step include elegant and beautiful functional items from the Bootstrap library.
Indeed the polishing of the user interface was the final sprint of our project.

VI. Outlook

During this project we have successfully constructed a network of managed switches using several Raspberry Pi single board computers. Utilizing a testing environment based on those devices, we have developed and tested an application for network topology detection. Our solution implements a robust back-end that detects devices and links in a network of switches and end-hosts. The collected information is saved using a flexible JSON data structure that can easily be extended in the future. The application also presents obtained network information in an easy to use visualization plane.
During the development process we have also, in preparation, implemented multiple additions that allow for changes to the saved data structure containing the network topology. These consist of the ability to edit the data structure directly from the front-end. Addition, deletion and edition of nodes, links and information within them is possible. We have also provided tools for future integration of network flow management into the application.

A. Future Work

As this work only provides an easily extensible base for detection and control of networking infrastructure, there are multiple ways in which it should be extended in the future. In the following we list some of the most prominent ways in which our application could be improved:

- Currently only the saved data structure can be changed and edited. In the next step the ability to directly configure interfaces of network nodes using SNMP, NETCONF, or other protocols could be implemented

- Enable the application to influence flows in the network - create or edit paths the packets can take and apply them on the network level
- Visualize flows in the graphical network representation
- Introduce controlling mechanisms similar to OpenFlow/SDN Controller using SNMP or NETCONF. There are obvious limitations to such an approach, exemplary the need for information polling from the network nodes what introduces overhead. Those constraints would naturally have to be investigated.

B. Summary

All in all, we have developed an extensible network topology detection application using multiple modern technologies. We believe that it provides new opportunities for the TSN community and hope it will serve as a perfect base for other interesting and inspiring projects.

The code can be found in a GitLab repository that has been provided to us by Fraunhofer FOKUS: https://gitlab.fokus.fraunhofer.de/openiotfog/topology-detection/tree/student

REFERENCES

[1] Tim Bray. *The JavaScript Object Notation (JSON) Data Interchange Format*. RFC 8259. Dec. 2017.

[2] Roy Thomas Fielding. "Architectural Styles and the Design of Network-based Software Architectures". AAI9980887. PhD thesis. 2000.

[3] Raspberry Pi Foundation. *Raspbian*. 2019. URL: https://www.raspberrypi.org/downloads/raspbian/ (visited on 07/23/2019).

[4] DATACOM Buchverlag GmbH. *LLDP (link layer discovery protocol)*. 2013. URL: https://www.itwissen.info/LLDP-link-layer-discovery-protocol-LLDP-Protokoll.html (visited on 07/23/2019).

[5] DATACOM Buchverlag GmbH. *SNMP (simple network management protocol)*. 2012. URL: https://www.itwissen.info/SNMP-simple-network-management-protocol-SNMP-Protokoll.html (visited on 07/23/2019).

[6] Ivan Pan. *JavaScript*. 2019. URL: https://developer.mozilla.org/de/docs/Web/JavaScript (visited on 07/23/2019).

[7] W3Techs. *Usage of JavaScript libraries for websites*. 2019. URL: https://w3techs.com/technologies/overview/javascript_library/all (visited on 07/23/2019).

[8] John Wolfe. *A Brief History of Python*. 2018. URL: https://medium.com/@johnwolfe820/a-brief-history-of-python-ca2fa1f2e99e (visited on 07/23/2019).