UNIVERSITÄT LEIPZIG

Wireless Mesh as easy deployable and scaleable Internet backbone

Bachelorarbeit

Paul Spooren

Abschlussarbeit zur Erlangung des akademischen Grades
Bachelor of Science (B.Sc.)
Fakultät für Mathematik und Informatik
Institut für Informatik

09.08.2018
Betreuer: Dr. Michael Martin

Abstract

Within the last years the importance of internet driven applications continuously raised and so the need of its distribution. The question of distribution is broadly solved for static setups, however lacks an answer for quickly deployed installations setup by non-professionals which would allow Internet coverage of events for instance music festivals or winter markets but also in case of an emergency where other infrastructure crashed.

The main research question of this project is to find a way which allows easy creation of wireless access points (APs) with minimal manual configuration. It solves the challenge of establishing a backbone network which connects all devices offering an AP so that traffic is routed to the Internet.

A feasible approach to easily setup a backbone network is to use a mesh topology, where nodes are connected to their surrounding neighbors (other nodes). This topology can be seen as an alternative to the classical star topology where one device connects to multiple others. Changes within the network, like extending the covered area with more APs, are automatically detected by an application running on every node. The network becomes easily redundant once nodes are connected to multiple neighbors. In case of device failures alternative routes are calculated and used without manual intervention.

The final outcome is a firmware to automatically create APs and a backbone network based on wireless mesh. To make this practically usable tools are created to monitor and configure the network in an easy fashion. Additionally, there are hardware recommendations attached to the software stack providing a full setup within this work which is practically usable for the use cases mentioned previously.

Contents

1 Introduction **5**
1.1 Motivation . . . 5
1.2 Aim . . . 5
1.3 Illustrative use case . . . 6
1.4 Structure . . . 6
1.5 Related work . . . 6
1.5.1 Freifunk's Gluon . . . 7
1.5.2 The Quick Mesh Project (qMp) . . . 7
1.5.3 LibreMesh (LiMe) . . . 7

2 Basics concepts **8**
2.1 Physical connection between devices . . . 8
2.1.1 Wireless access points . . . 8
2.1.2 Wireless mesh . . . 8
2.2 Routing of network traffic . . . 9
2.2.1 Classical network topologies . . . 9
2.2.2 Dynamic routing protocol . . . 10
2.3 Operating systems for routers & network devices . . . 10

3 Project requirements **11**
3.1 Overall project requirements . . . 11
3.2 Requirements of the Mesh Firmware (N) . . . 11
3.3 Requirements of the management interface (I) . . . 11
3.4 State of current mesh distributions . . . 12
3.4.1 Gluon . . . 12
3.4.2 qMp . . . 12
3.4.3 LibreMesh . . . 13
3.5 Design decisions . . . 13
3.5.1 BMX7 as routing protocol . . . 14
3.5.2 Monitoring . . . 14
3.5.3 Configuration tool . . . 18
3.5.4 Initial firmware creation . . . 20
3.5.5 Initial configuration . . . 20

4 Implementation **21**
4.1 Administrative backend . . . 21
4.1.1 Web interface & authentication via UBUS . . . 21
4.2 Initial configuration retrieving . . . 28
4.3 Usage of `bmx7-sms` plugin for configuration . . . 28
4.4 IP address collision avoidance . . . 29
4.5 Alerts in case of malfunction . . . 29

5 Collaboration **30**
5.1 LibreMesh extension and fixing . . . 30
5.2 Encrypted mesh networks . . . 30
5.3 Working at the UPC Barcelona . . . 30

6 Evaluation **31**
6.1 Monitoring evaluation . . . 31
6.1.1 Resource requirements for monitoring . . . 32
6.2 Management evaluation . . . 32
6.2.1 Initial node setup . . . 33
6.2.2 Testing the `cloud-sync` feature . . . 34
6.2.3 LibreMesh with encrypted mesh . . . 34

7 Conclusion **35**
7.1 Solved project requirements . . . 35
7.2 Created, extended and ported packages . . . 35
7.2.1 Package creation . . . 35
7.2.2 Package extension . . . 36
7.2.3 Package porting . . . 36
7.3 Projects drawbacks and limits . . . 36
7.3.1 Plain text configuration . . . 37
7.3.2 No roaming support between devices . . . 37
7.3.3 Complicated initial router flash . . . 37
7.3.4 Insecure connection to configuration interface . . . 38
7.3.5 BMX7 dependencies . . . 39
7.3.6 No IPv6 connectivity for clients . . . 39
7.4 Outlook . . . 39
7.4.1 Interface design . . . 39
7.4.2 Grafana to visualize metric graphs . . . 39
7.4.3 Test scalability of routing BMX7 . . . 40
7.4.4 Sensor network . . . 40
7.4.5 Encrypted transport layer . . . 40
7.4.6 Opportunistically encrypted open hotspots . . . 40
7.4.7 Modular configuration distribution . . . 41

8 Attachments **42**

Erklärung **47**

References **48**

1 Introduction

Due to the collaboration over the last two years with several Community Networks I learned about the advances of mesh networks in contrast to statically installed ones. During deployment of such networks I repeatedly realized the absence of a toolset to simplify the setup and keep track of the network health. In this work I implemented the missing tools and extended existing solutions to be easily usable in real environments.

Community networks introduce additional challenges which exceed the focus of this document. For that reason I decided to focus on use cases where all hardware is owned by a single instance, simplifying the setup and lowering security requirements. The tool created can nevertheless be used in community networks, although this may require some additional modifications to follow their decentralized requirements.

1.1 Motivation

Creating a backbone network is a non trivial task. It requires experts to plan the network topology ahead and define the way traffic network is routed within the network. Changes of the structure or optimization to prevent overload need manual intervention. Manual setups may also lead to misconfiguration and cause network downtimes. A more automated way could simplify the setup of backbone networks possibly with mesh networks.

While for many years WiFi mesh networks have only been used by network activists like Freifunk[1] or LibreMesh[2] the technology now has been adopted even by big companies like Google Inc.[3].

Having seen the mesh approach work in Community Networks[4] I decided to make it usable for other use cases as well, allowing individuals interested, which haven't touched this area yet, to deploy their own decentralized backbone network.

1.2 Aim

This work focuses on the simple initial setup, monitoring and configuration of a backbone network which distributes an Internet uplink, setup and managed by a central instance owning all nodes of the network.

This approach is not to be compared to tools and setups specialized on community networks as exceedingly discussed in other works [1, p. 4]. While parts of the implemented tools may be used in such a setup, the centralized management is focused.

The outputs of this project are the following three parts:

- Node firmware which automatically integrates into the mesh network.
- A monitoring tool to check the usage and health of the network.
- A configuration tool to set basic properties of the network.

[1] https://freifunk.net/
[2] https://libremesh.org/
[3] https://store.google.com/product/google_wifi_learn
[4] http://tomir.ac.upc.edu/qmpmon

Different from existing approaches for monitoring and managing wireless networks, no complex per device configuration is planned following the idea of zero configuration, keeping all devices as generic as possible. Neither is any longtime monitoring over several years implemented like for instance at Guifi.net[5]. This is due to this works focus on temporal installations, where no such long time periods occur.

1.3 Illustrative use case

To give a better idea of what the output of this work could be used for, below a short description of a possible use case.

A winter market takes place yearly within the center of a city. For this event small houses are built up and sell various goods. An electronic payment system requires an Internet connection to verify payments. As thousands of mobile devices with LTE arrive to the event, mobile telephony network may become overloaded and the payment system could break down, if it would rely on it. Instead, a number of WiFi access points could be created, where only authorized payment devices can connect to. The mesh approach explained before could be used in the following way, to allow the easy setup and spreading of APs. A small number of vendors receive a pre configured mesh node and connect them to a power plug. Active nodes connect with all other nodes in reach and offer an access point at the same time. A special gateway node announces its direct Internet connection to all active nodes, which then start using it.

If the winter market, due to unforeseen reasons, is shaped in a different way than initially planed, the nodes connect dynamically, providing the same functional backbone network. Additionally, if the winter market grows over the initially planed area, a node is added and the access point coverage increases. While the mesh approach could solve the growth and dynamic changes easily, a classical static network likely requires manual, configuration per devices.

1.4 Structure

This work is separated into sections starting from assessing requirements and ends with a summary of the final product. In the section *project requirements* 3 specific needs of the works scope are explained. It includes the sub-section *design decisions* 3.5 elaborating on used concepts and concrete realization - followed by section *implementation* 4 which covers technical details of used software and created code. The last section *conclusion* 7 gives a summary of all work done leading to fulfill the main objective.

Additionally provided are *attachments* 8 which discuss hardware and configurations used within this work.

1.5 Related work

Previous to this work established solutions in the area of mesh networks were reviewed to find a suitable starting point. The projects mentioned below are community driven with a focus on community networks (CNs) [2, p. 1]. While CNs are not focus of this work the software stack is open source and can therefore be modified to fit the projects requirements.

All below mentioned distribution or frameworks are based on OpenWrt and run on regular home routers.

[5] https://guifi.net/en/guifi/menu/stats/growthmap

1.5.1 Freifunk's Gluon

The Gluon software stack is broadly established in various states of Germany[6]. The software allows users to share their Internet connectivity in two ways. Primary it is possible to offer Internet connection (received by DSL or TV Cable) via an open AP to surrounding neighbors. The traffic is routed over a community-owned VPN to avoid possible lawsuits. Secondly the routers connect via wireless mesh with neighbored nodes and share their uplink if existent, also vice versa.

To create the mesh network both *802.11s* and *ad hoc* are supported, for routing the protocol B.A.T.M.A.N. Advanced[7] is used.

1.5.2 The Quick Mesh Project (qMp)

This distribution offers firmware images to setup a mesh network with minimal manual intervention. The firmware automatically sets up all requirements to be used in a mesh network and automatically connects to neighbors. If an Internet upstream is announced by the mesh routing protocol, it automatically offers the connection to connected clients. The project is actively used in the Barcelona city area[8] with an active user base.

The firmware relies on the mesh routing protocol BMX6[9] and creates the backbone mesh connection via *ad hoc* (in contrast to the newer *802.11s* modes).

1.5.3 LibreMesh (LiMe)

LibreMesh is more a community driven framework than a distribution. Due to its modularity it supports various use cases. It is possible to change activated routing protocols, preinstalled software and IP addresses of subnets used. A single configuration file[10] allows to control all WiFi, mesh and network related settings.

Additionally it offers precompiled firmware images which work without any further configuration. These images use a combination of *BMX6* and *B.A.T.M.A.N*.

The project is under active development and therefore offers a good starting point to fulfill the requirements stated in section 3.

[6] https://multi.meshviewer.org/
[7] https://www.open-mesh.org/projects/batman-adv/wiki/Doc-overview
[8] http://tomir.ac.upc.edu/qmpmon
[9] https://github.com/bmx-routing/bmx6
[10] https://github.com/libremesh/lime-packages/blob/develop/packages/lime-system/files/etc/config/lime-defaults

2 Basics concepts

This section aims to offer introducing knowledge of used technologies, covering basics of wireless mesh networks, routing and dynamic routing protocols and lastly the operating system for embedded devices OpenWrt.

2.1 Physical connection between devices

Using cable or fiber connection between the nodes is optimal in terms of speed but requires more work to setup. For the rapid deployment wireless connections are preferred. This approach is broadly tested in various community networks [2, p. 2].

2.1.1 Wireless access points

The most established setup for Wireless LAN or WiFi is an access point offering a network to which clients may connect. This approach is found in nearly every house with Internet access and many public places. This setup is also usable to provide uplink connectivity and is used by Wireless Internet Service Providers. A fail of the access point results in connection loss between all connected client nodes, even if they would be within line of sight.

2.1.2 Wireless mesh

The WiFi standard *802.11s* (*11s*) [3] offers the possibility that two or more devices connect without the limits of a *client* and *access point* setup. All devices are connected equally, allowing continuous connectivity even if single devices fail, as illustrated below. The *11s* WiFi mesh creates the foundation of the backbone network discussed in this project.

Important to mention, the *11s* standard also introduces a routing protocol which is deactivated in this project in favor of the routing protocol *BMX7*, offering required features.

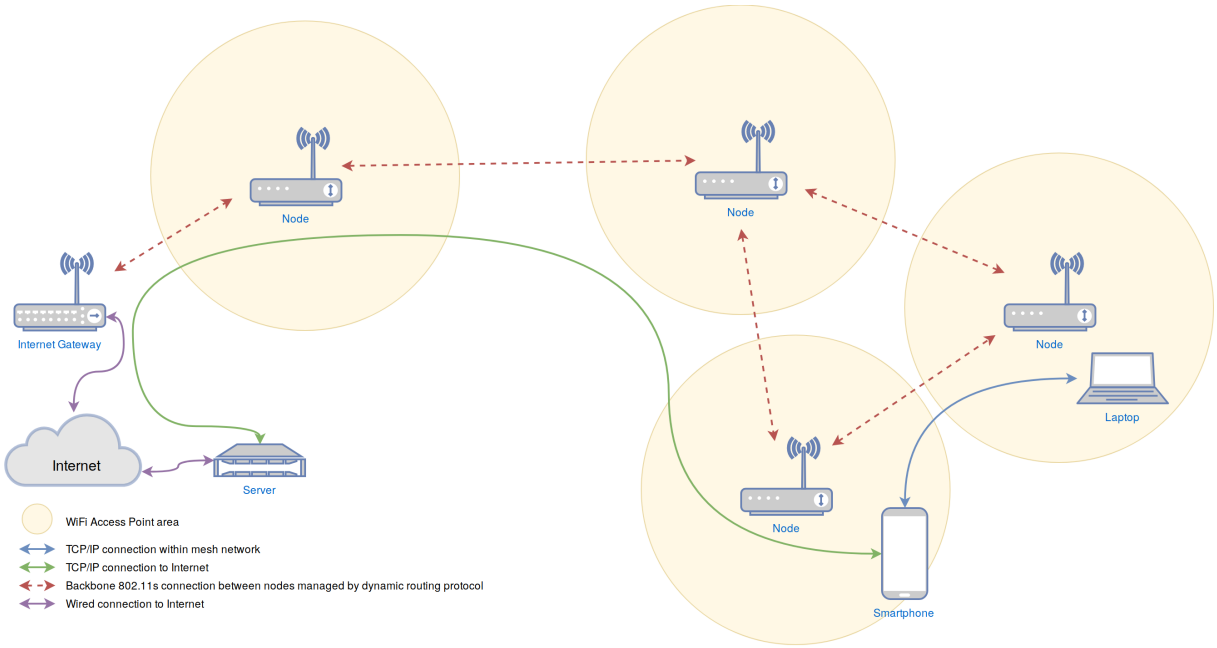


Figure 1: Wireless mesh network - Possible setup of a mesh network which offers inter mesh connections to clients and via a gateway Internet access.

Figure 1 shows an illustrative mesh network combining wireless mesh connections with *11s* and APs. The red arrows stand for the backbone which is handled by the dynamic routing protocol and transparent for connected clients. Clients may connect via IP addresses to other clients within the mesh or use a gateway node which offers uplink to the Internet.

2.2 Routing of network traffic

A telecommunication network means the connection from any number of devices in a way that they can communicate with each other. Nodes within the network require routing information to forward or deliver packages to its receiver.

The following information is vastly simplified and has an exceeding explanation in related works [4, p. 5].

2.2.1 Classical network topologies

A network topology describes the connection design between nodes of a network. Figure 2 pictures both star and tree topology.

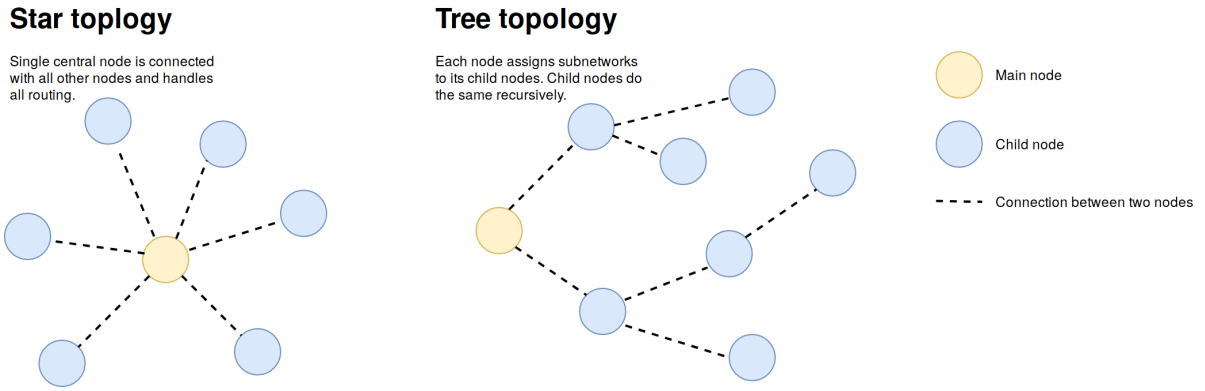


Figure 2: Star and tree topologies

This approach has the disadvantage that a node might move from one subnet to another. Furthermore chaining of nodes is difficult as it requires previous planning of subnetwork sizes.

2.2.2 Dynamic routing protocol

The statically routed network described above can be set once and then work without any additional daemons running on nodes of the network. To allow dynamic changes of the network, like adding a new device, an active service has to run on devices and react on changes. Dynamic routing protocols are active components which automatically adapt the routing so that devices stay reachable. The protocols use metrics to determine the link quality and choose the fastest link to connect between devices.

This project takes advantage of the actively developed and constantly improved mesh routing protocol *BMX7* [5]. While there exist multiple other dynamic and on mesh specialized routing protocols *BMX7* is chosen for its integrated features covered in section 3.5.

2.3 Operating systems for routers & network devices

Many home routers are able to run or are actually delivered with a very light Linux operation system called OpenWrt[11]. It uses a modified Linux kernel and has various optimizations to be usable on devices which often have as little as 8MB of storage and 32MB of memory.

OpenWrt is comparable to other Linux distributions like Debian, but focused on network related devices and embedded devices in general.

By default it can work as a wireless AP just usable as home routers do. However it is possible to install mesh related packages via a package manager and configure them depending on the required setup.

[11] https://openwrt.org/

3 Project requirements

The following requirements and decisions regarding design are separated in node and management specific lists. All items are based on my previous experiences with the deployment and maintenance of mesh networks and advises from existing Communities managing their own network.

Additional are aspects specially for this use case that cover the security of the network which is normally not implemented in Community networks as they use unencrypted connections between devices.

3.1 Overall project requirements

Meta requirements that lead general design decisions 3.5.

- M1 **Reuse existing solutions:** Before implementing new software it should be tried to adapt to or extend existing solutions.
- M2 **Modularity:** While focused on the described use case, created software should be modular enough to be used in other scenarios as well.

3.2 Requirements of the Mesh Firmware (N)

Covering the requirements of the software stack running on nodes.

- N1 **Zero-Conf Setup:** The initial setup of routers must work without per-device specific configuration, making it easy to rapidly deploy new devices.
- N2 **Encryption:** All wireless communication must be encrypted. Only devices authenticated with a password can enter the network.
- N3 **Frequency separation:** APs must run on a different frequency than the backbone mesh network to avoid connected clients to affect the backbone connection quality.
- N4 **Automatic device discovery:** New setup devices must automatically connect to the network and participate in the mesh network.

3.3 Requirements of the management interface (I)

Covering the requirements of the monitoring and management interface for administrative users.

Beneath the previously mentioned community collaboration the monitoring interface is inspired by preceding work in this area [6] [7] to build on previously acquired knowledge of demands in mesh network monitoring.

- I1 **Limited access:** Only authorized users can see monitoring data or change the network configuration.
- I2 **Simple setup:** No complex dependencies and no complex manual setup must be required to install the management interface, allowing non technical users to setup their own network.

- I3 **Network graph:** A graph showing all mesh-nodes of the network and the connection quality between them. This visualizes the network and helps to spot bottlenecks and malfunctioning devices.
- I4 **Automatic monitoring:** New mesh-nodes must be automatically added to monitoring and management interface, allowing configuration and status checks.
- I5 **Alerts:** In case of device failures or overloads the network administrators must automatically receive a warning.
- I6 **Network configuration:** The interface must allow the configuration of basic parameters of the network, like Access Point names (SSID) and passwords.

3.4 State of current mesh distributions

In this subsection some key limitations of the mesh distributions above described are mentioned and thereby the choice for *LibreMesh* will be explained.

3.4.1 Gluon

- **req. I2** Gluon offers a very sophisticated monitoring system[12] which exceeds the scope of this work. However it worked as a good inspirational source for the herewith created interface.
- **req. N1** Nodes are individually configured and use manually selected IP addresses[13].
- While it is not a primary goal of this work to be highly scaleable, it is reasonable to offer greater numbers of access points to cover large areas. However using *B.A.T.M.A.N.* as mesh routing protocols comes with limitations in terms of scalability and thereby requires additional configuration around this issue[14].

3.4.2 qMp

- **req. N2** The final firmware is required to offer encrypted wireless connection between nodes and clients but also between nodes. This isn't possible for *ad hoc* mode and only available via *802.11s*. In recent development versions of *qMp* support for *802.11s* was added[15] for testing but long after the decision to use *LibreMesh* as a base.
- **req. I3** Work was done for a monitoring system but never became fully integrated in the *qMp* [6]. The scope of the monitoring system was also different as it was implemented in a decentralized way.

[12] https://develop.meshviewer.org/
[13] https://config.berlin.freifunk.net/wizard/routers
[14] https://trier.freifunk.net/2017/08/28/die-hoods-kommen/
[15] dev.qmp.cat/projects/qmp/repository/revisions/003700f24b3859836644425de4f30c773bf6acca

3.4.3 LibreMesh

- *LibreMesh* did not offer any monitoring possibilities apart from a very rudimentary network graph offered by the `luci-app-bmx6`[16] package.
- **req. I6** With a package called `first-boot-wizard`[17] it is possible to distribute the initial configuration file called `lime-defaults` to other, unconfigured nodes within read. While this approach works for an initial setup it does not allow continuous changes nor distribution of settings outside the `lime-defaults` file, like SSH keys.

Table 1: Mesh distribution features: *[Y]* exists, *[N]* missing, *[U]* unknown, *[E]* extended within this work, *[I]* implemented within this work.

req. ID	Feature / Requirement	Gluon	qMp	LibreMesh
	Node			
N1	Zero-Conf Setup	N	Y	E
N2	Encryption	N	N	E
N3	Frequency separation	N	Y	I
N4	Automatic device discovery	N	Y	E
	Management interface			
I1	Limited Access	N	N	I
I2	Simple setup	N	N	I
I3	Network graph	Y	N	I
I4	Automatic monitoring	N	N	I
I5	Alerts	U	N	I
I6	Network configuration	N	N	I

3.5 Design decisions

This sections cover the output of this work and explains how the previous stated project requirements were fulfilled. The content is focused on fundamentals and basic concepts and less technical than the following section covering the concrete implementation.

[16] https://github.com/openwrt-routing/packages/tree/master/luci-app-bmx6
[17] https://github.com/libremesh/FirstBootWizard

3.5.1 BMX7 as routing protocol

While there exists various mesh routing protocols like Batman Advanced[18], OLSR [8] and Babel [9], three main reasons led this project to use *BMX7*.

- It is predecessor *BMX6* is the main routing protocol of qMp [4, p. 23] and LibreMesh both were used as a main source of inspiration.
- *BMX6* was evaluated to likely show better scaling and overall performance then the competitors Babel and OLSR [10, pp. 53–54].
- It offers various plugins avoiding to re-implement required features[19].

3.5.2 Monitoring

To maintain a stable mesh network it is crucial to know key live information of the devices. This can be done by a monitoring system obtaining data from nodes and presenting them in an easily readable fashion.

Required data

It has to be distinguished between *device* and *network structure*. The combination of both enables network administrators to quickly check the health of the network and find bottlenecks.

Monitoring of nodes

The collected metrics must allow the administrator to easily recognize device malfunction and give an overview of the running network. The *per device* monitoring therefore contains the following data:

- CPU load
- Free memory
- Traffic of AP and mesh
- Number of connected clients
- Firmware version
- Device hardware

Network structure and quality graph

Next to individual device data it is useful to track the connectivity between nodes allowing an optimization, following the idea of *If you can't measure it, you can't improve it*. It must be possible to check the health of links and the estimated maximal achievable bandwidth between devices, all in an easily readable graph.

Required information about the structure are therefore:

- List of nodes
- Connection between nodes (links)
- Estimated bandwidth of link
- Used interface for connection (Cable or WiFi)

[18] https://www.open-mesh.org/projects/batman-adv/wiki/Doc-overview
[19] https://github.com/bmx-routing/bmx7#bmx7-plugins

Monitoring backend

A monitoring backend must be capable of efficiently storing metrics and offer a convenient API to extract the data. Newly added nodes must appear automatically in the monitoring system without any manual intervention. Also metrics are *pulled* from the nodes by the monitoring instance rather than *pushed* from the node to the backend. The *pull* approach offers various advantages for this setup:

- Easily setup multiple monitoring instances for redundancy.
- Migration to a new monitoring machine does not require any node configuration.
- Even with a higher number of nodes, the monitoring backend won't be overloaded by to many incoming metrics pushed by nodes. The backend decides on its own when to collect metrics from nodes.
- Let the backend decide which metrics are of interest instead of receiving all possible metrics available[20].

This project will not cover a comparison between monitoring engines so an established and easy to setup solution is chosen: Prometheus[21]. It offers the required backend to scrape nodes, store the data and offer them via an API[22] or various web interfaces like Grafana[23].

Monitoring web interface

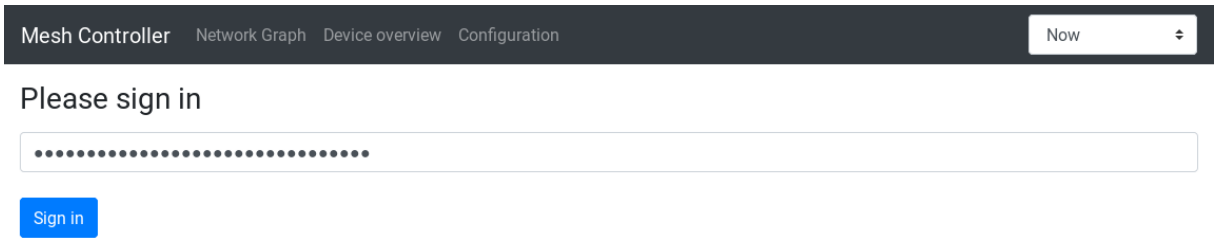


Figure 3: meshrc Login page. All interaction, receiving monitoring data or changing settings, requires authentication which is handled by UBUS.

The web interface requires an authentication to only allow administrators to view stats and change the configuration. Instead of implementing an individual authentication it is taking advantage of OpenWrt micro Bus system called *ubus*[24] which manages user login. The password entered in fig. 3 is send to *ubus* via the *uhttpd-mod-ubus*[25] package which allows Bus communication over HTTP(S).

[20] https://github.com/prometheus/node_exporter/#filtering-enabled-collectors
[21] https://prometheus.io
[22] https://prometheus.io/docs/prometheus/latest/querying/api/
[23] https://prometheus.io/docs/visualization/grafana/
[24] https://openwrt.org/docs/techref/ubus
[25] https://openwrt.org/packages/pkgdata/uhttpd-mod-ubus

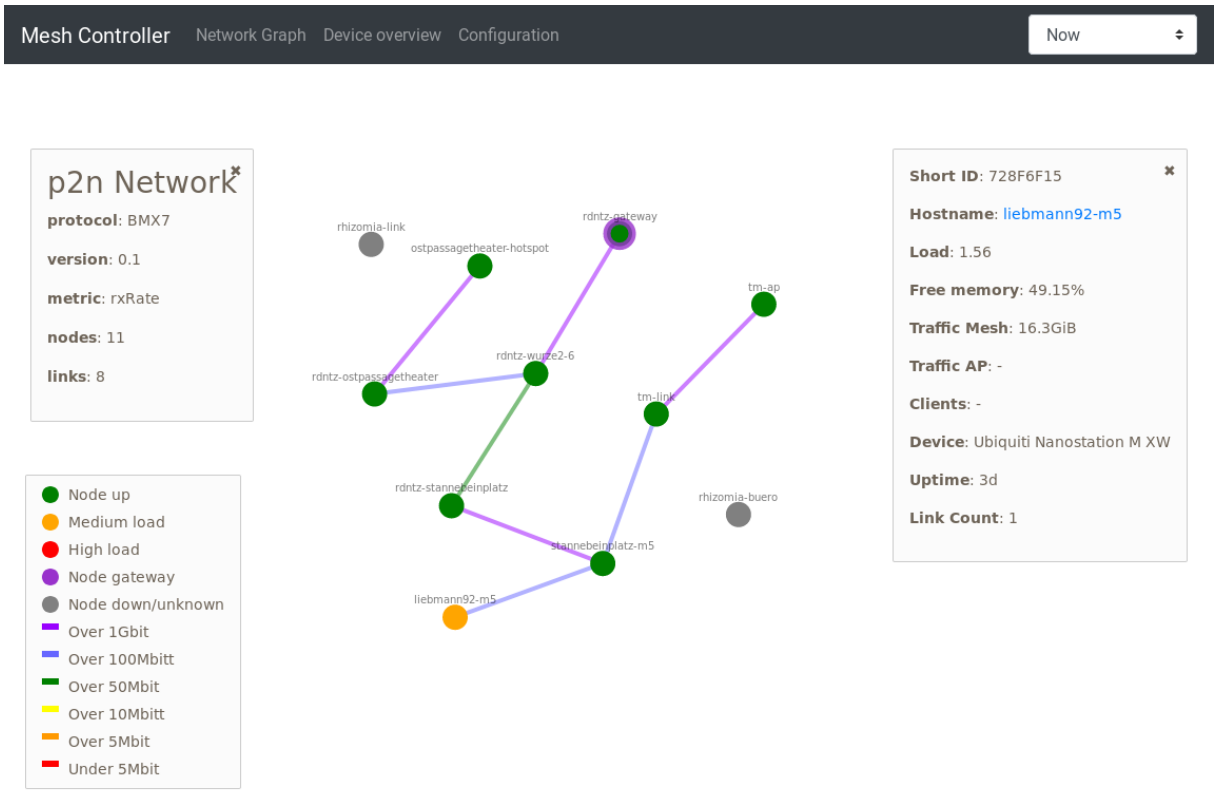


Figure 4: meshrc Graph now - Shows the current state of the network rendered based on the JSON output of *p2n*.

Figure 4 shows a test-bed where a total number of 11 nodes are deployed. All nodes are running the *prometheus-node-exporter-lua* script with the mentioned *BMX7* plugin to show their active links. Most connections work fine and are estimated with more than 50Mbit/s bandwidth. Connections with higher throughput are colored in purple and are likely by cable.

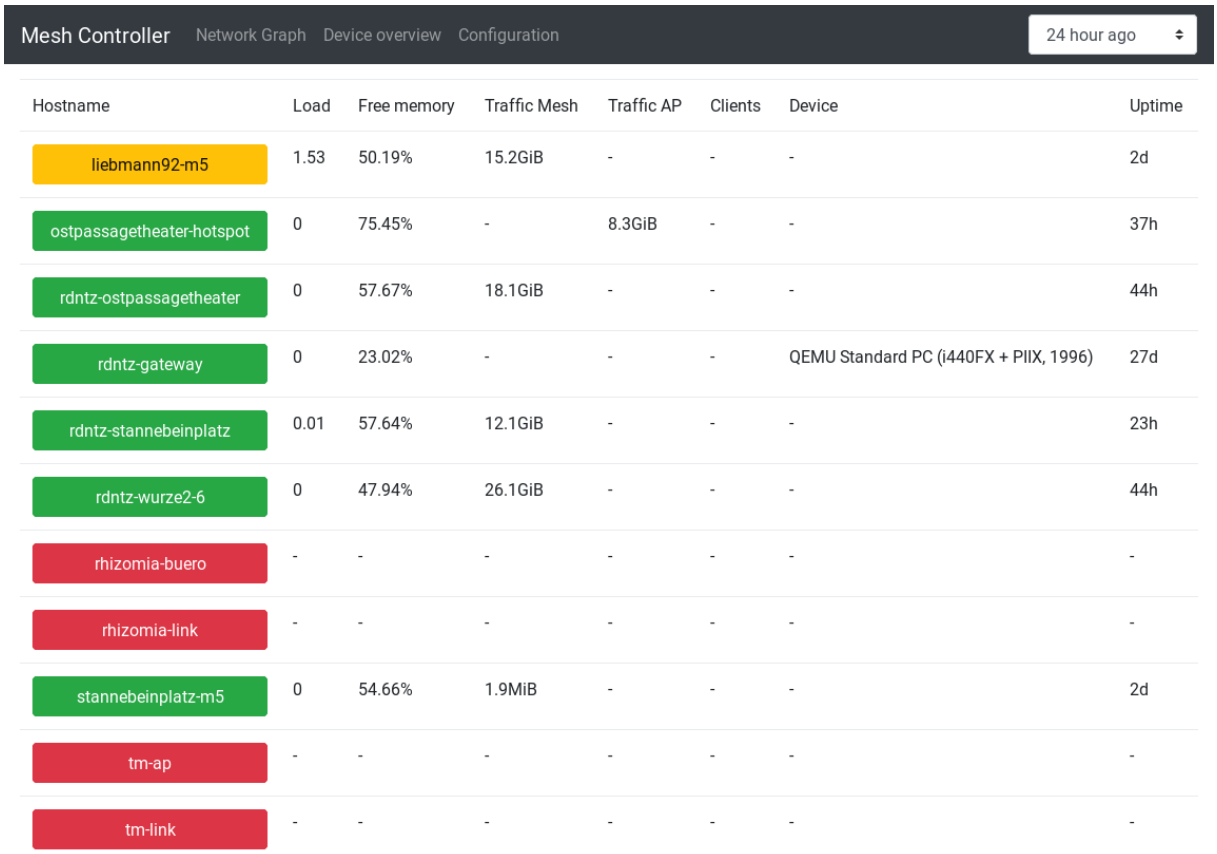
Additionally, the node *liebmann92-m5* has currently high load and is shown in orange. This likely happens due to the use of weak hardware, not fulfilling the minimal requirements described below, a malfunctioning application or script causes the high load. The viewer can easily spot where further manual inspection is required. Both nodes *rhizomia-buero* and *rhizomia-link* are currently down and no data can be obtained. Nodes that provide uplink connectivity (gateways) are marked with a purple border around the dot, helping the viewer to understand the network flow.

Mesh Controller Network Graph Device overview Configuration Now

Hostname	Load	Free memory	Traffic Mesh	Traffic AP	Clients	Device	Uptime
liebmann92-m5	1.56	48.96%	16.3GiB	-	-	Ubiquiti Nanostation M XW	3d
ostpassagetheater-hotspot	0	73.88%	-	9.1GiB	-	TP-Link TL-WDR3600 v1	2d
rdntz-ostpassagetheater	0	53.88%	19.9GiB	-	-	Ubiquiti Loco M XW	2d
rdntz-gateway	0	21.31%	-	-	-	QEMU Standard PC (i440FX + PIIX, 1996)	28d
rdntz-stannebeinplatz	0	54.54%	22.6GiB	-	-	Ubiquiti Loco M XW	47h
rdntz-wurze2-6	0	47.81%	37.5GiB	-	-	Ubiquiti Loco M XW	2d
rhizomia-buero	-	-	-	-	-	-	-
rhizomia-link	-	-	-	-	-	-	-
stannebeinplatz-m5	0	52.14%	226.7MiB	-	-	Ubiquiti Nanostation M XW	3d
tm-ap	0.08	87.32%	99.6MiB	7.5GiB	-	ZBT-WG3526 (16M)	3d
tm-link	0	17.15%	15.9GiB	-	-	Ubiquiti NanoStation M5	3d

Figure 5: meshrc Overview now - Rendered table based on the JSON output of *p2n*.

Figure 5 shows a table with additional information about all devices. All data monitored can be added to this table. The exact metrics of interest should be evaluated in further practical tests and exceeds the scope of this work.



Hostname	Load	Free memory	Traffic Mesh	Traffic AP	Clients	Device	Uptime
liebmann92-m5	1.53	50.19%	15.2GiB	-	-	-	2d
ostpassagetheater-hotspot	0	75.45%	-	8.3GiB	-	-	37h
rdntz-ostpassagetheater	0	57.67%	18.1GiB	-	-	-	44h
rdntz-gateway	0	23.02%	-	-	-	QEMU Standard PC (i440FX + PIIX, 1996)	27d
rdntz-stannebeinplatz	0.01	57.64%	12.1GiB	-	-	-	23h
rdntz-wurze2-6	0	47.94%	26.1GiB	-	-	-	44h
rhizomia-buero	-	-	-	-	-	-	-
rhizomia-link	-	-	-	-	-	-	-
stannebeinplatz-m5	0	54.66%	1.9MiB	-	-	-	2d
tm-ap	-	-	-	-	-	-	-
tm-link	-	-	-	-	-	-	-

Figure 6: meshrc Overview 24 hours ago - The NetJson was generated with the relative timestamp 24h to show a previous state of the network.

Finally the top right corner of fig. 6 is a drop-down menu allowing to see previous states of the network, various values between 2 minutes and 24 hours. In this test-bed some nodes were down 24 hours ago.

3.5.3 Configuration tool

The mesh configuration is limited to a very basic set of options to allow only surfacing settings to be changed, protecting non technical users from breaking the network. As mentioned in the project requirements above, the network must be self configuring as much as possible.

User interface for configuration

Figure 7 shows that the basic configuration interface only allows to change the *mesh password* which is used for the mesh backbone connection and the *access point password* which is used for clients. Distribution of initial configuration can be disabled via the top left checkbox and additional SSH keys for remote access can be set.

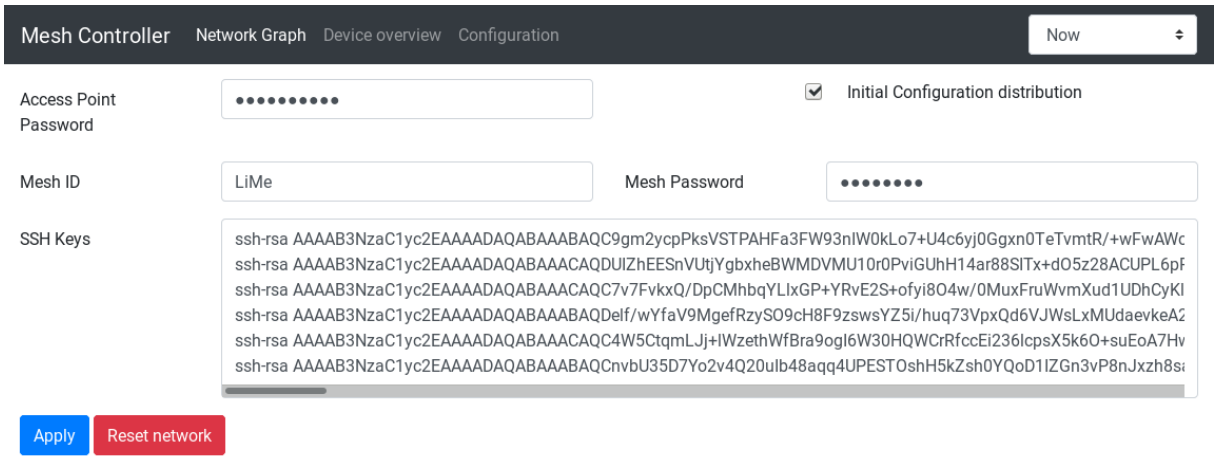


Figure 7: meshrc Configuration - Basic configuration is possible via the configuration interface.

Also, the red *reset* button will flush all configuration on all nodes. This is useful to bring all nodes back in a mode where they try to fetch a configuration via a *link local* connection [11, p. 11]. Further details about this are described below in the section *meshrc-initial* 4.2. Additionally, as shown in fig. 8 it is possible to set individual settings per node, regarding the nodes *host-name* and it is *access point name*.

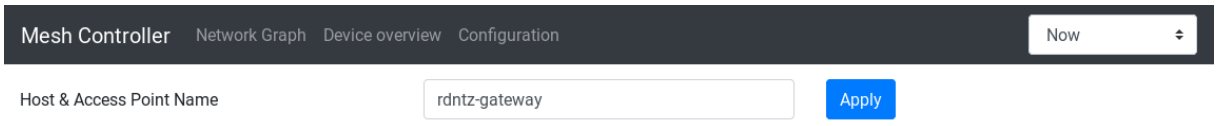


Figure 8: meshrc Node configuration - Per node configuration with limited functionality to prevent user from breaking connectivity.

Distribution of new configurations

Instead of implementing a new tool to distribute data over a mesh network, requiring also a secure way for authentication, I chose to use the existing solution called *bmx7-sms*[26]. It solves the two requirements needed:

- Distribute commands or configuration files to all nodes of the mesh network.
- Add authentication layer to allow nodes verifying the received command or configuration is legitimate.

All nodes of a *BMX7* based mesh network have a unique identification[27] which is based on a public key pair created on first boot. All descriptions[28] are signed by the nodes and so are the messages distributed by the *bmx7-sms* plugin [12].

[26] https://github.com/bmx-routing/bmx7#sms-plugin
[27] https://github.com/bmx-routing/bmx7#global-id
[28] https://github.com/bmx-routing/bmx7#descriptions

Badly signed messages are not accepted by nodes and thereby not distributed nor processed. Received messages are stored in a file system containing the sending nodes ID. Thereby it is easy to check legitimate commands by controlling if the name of the received file starts with the ID of a trusted node.

An additional point of consideration is the question on how to deal with connection breaking changes. Meaning, if for instance the *mesh password* is changed, nodes further distanced from the initial sending nodes can no longer communicate and thereby not receive the new password. To address this problem I implemented a function to wait until the cloud is successfully synced.

3.5.4 Initial firmware creation

Nearly all router models require individual firmware images due to different specifics like used CPU architecture and available hardware components[29]. Instead of creating preconfigured images for a specific setup, there is an *initial firmware* which has all required packages preinstalled and a generic configuration (8.2) with disabled wireless radios for security reasons.

Obtaining the firmware can be achieved by using an online tool like Chef[30] or setting up a local OpenWrt build environment[31].

The section below covers the topic how a specific configuration is applied.

3.5.5 Initial configuration

Changing settings in the web interface results in the creation or update of an archive containing all configuration files. OpenWrt already uses this approach to backup and restore configured routers[32]. The very same approach is performed for initial node configuration in the following way.

A daemon running on the *initial firmware* checks if the node is already configured. If not, it tries to download the node configuration archive from all devices connected via link local. The idea behind that is, after a flash with the plain *initial firmware* image, the device just needs to be connected via cable to the gateway node and will automatically configure itself.

This allows to use a firmware image only containing a generic `lime-defaults` configuration file 8.2. A reset of the router will remove all configurations and restart the daemon looking for an applicable configuration.

[29] https://openwrt.org/toh/start
[30] https://chef.libremesh.org
[31] https://openwrt.org/docs/guide-developer/build-system/install-buildsystem
[32] https://github.com/openwrt/luci/blob/master/modules/luci-mod-admin-mini/luasrc/controller/mini/system.lua#L18

4 Implementation

This section covers technical details on how the previously stated requirements are implemented. As stated in req. M1 as overall paradigm it was tried to avoid re-implementing existing software and rather combine present tools to fulfill the task in a simple way.

All source code is published on GitHub.com in the `meshrc` repository[33] and in the official OpenWrt upstream packages.[34]

Figure 9 gives an overview of all implemented, ported and extended tools and their position within the mesh network.

Figure 9: Tool stack - Tools installed on the management and gateway node or on a generic node.

4.1 Administrative backend

To allow portability on different operating systems and easy extension without the need to be extremely performant[35], the web interface and it is components are written in Python3, Shell, and JavaScript.

4.1.1 Web interface & authentication via UBUS

The web interface was initially planned to use the Python Web Framework Flask[36] but was quickly replaced for pure JavaScript and the use of OpenWrts ubus[37] to minimize library dependencies and therefore storage footprint. Also the use of a full grown JavaScript framework to generate the web interface, apart from the graph creation, was decided against. The current implementation is done using only basic JavaScript function and contains in total less than 450 lines of code. To be usable on a mobile phones Twitters Bootstrap[38] is used for styling which also offers a responsive design for mobile administration.

All web interface functionality is guarded by *UBUS*, meaning only after a successful authentication it is possible to receive collected monitoring data or execute configuration commands. This follows *req. M1* as no additional

[33] https://github.com/aparcar/meshrc
[34] https://github.com/openwrt/packages/
[35] The code was design to never exceed $O(n)$, where n is the number of nodes. Time measurements were done with up to 100 virtual nodes resulting in response time of a few milliseconds.
[36] http://flask.pocoo.org/
[37] https://openwrt.org/docs/techref/ubus
[38] https://getbootstrap.com/

authentication stack is implemented. After a successful login *UBUS* returns a token which is attached to every request by the web interface. It automatically times out five minutes after the last usage.

Receiving JSON data via *UBUS* simplifies the implementation as the front-end part is entirely written in JavaScript without additional server side code. This brings the advantage that all rendering is performed on the viewing devices instead of the server, therefore it is also easier to scale if parts of the monitoring data should be accessible to clients without authentication.

Figure 10 illustrates the login sequence and communication between web interface and the backend which creates the JSON data. This and the following sequence diagram only handles the case that the provided password was correct. In case of bad input an error is returned by *UBUS*, brute force attacks are handled equally to UNIX logins, with delayed responses.

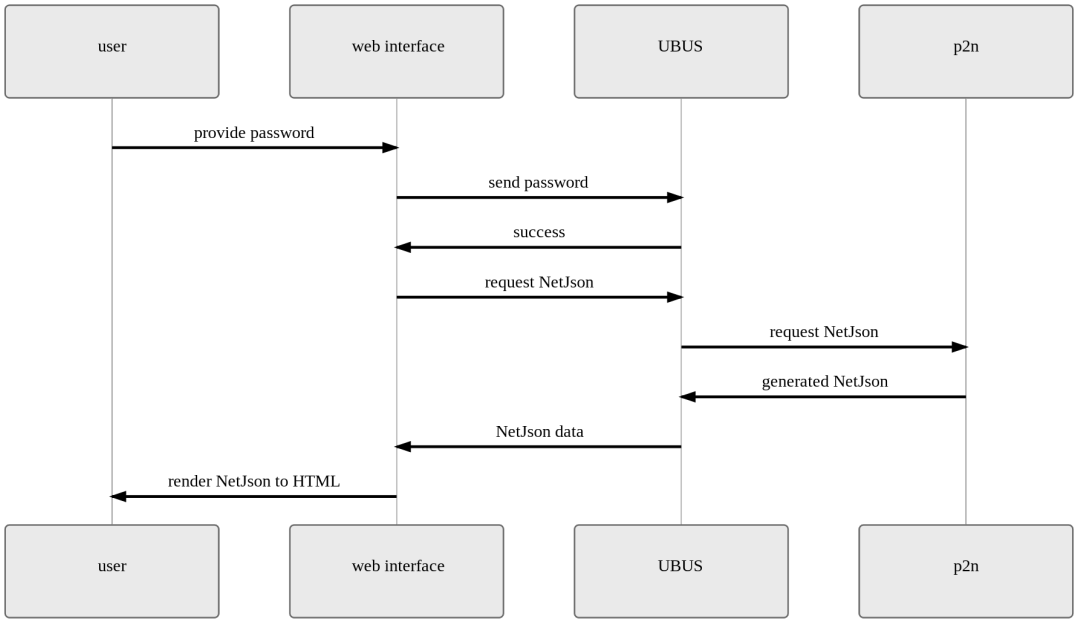


Figure 10: Login sequence - The user has to provide the *root* password authenticate in UBUS. After a successful authentication NetJson is requested from the UBUS daemon. UBUS runs the below discussed NetJson generator *p2n* and forwards it is output to the web interface. Via JavaScript the JSON data is rendered to HTML and presented to the user.

Just as for receiving monitoring data an authentication is required to change any configuration. Again *UBUS* is used, meaning a successful login allows seamlessly switching between checking monitoring data and applying configurations, as the *UBUS* token works for both functionalities. The use of *UBUS* for the configuration is shown in fig. 11.

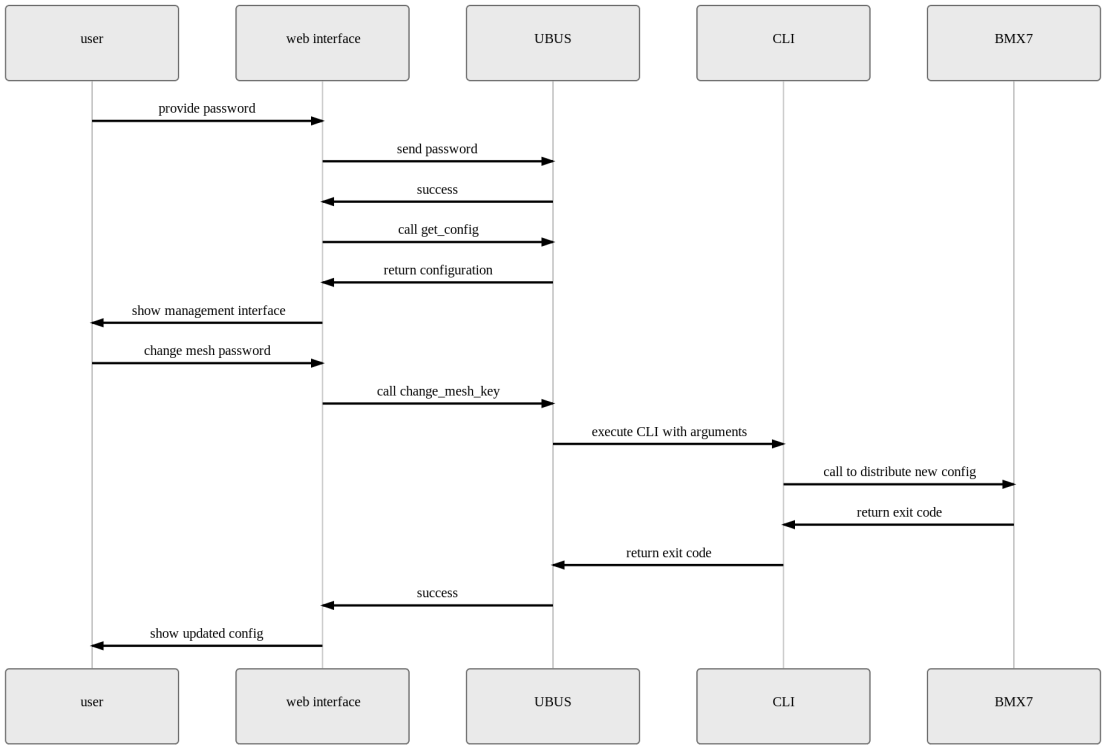


Figure 11: Configuration sequence - User has to provide a valid password to login. On success the configuration is loaded in the configuration form. To apply new settings changed values are submitted to the UBUS daemon which then executes the CLI. *BMX7* is requested by the CLI to distribute the new configuration files as described below.

Obtaining data from nodes

To fulfill *req I7* all nodes require a running service to parse local device metrics and offer them in the Prometheus format[39]. This can be achieved using an active service running on each nodes which parses device information from places like `/proc/` and expose them on a specified port via HTTP.

The tool `prometheus-node-exporter-lua` is a LUA re-implementation of the official Prometheus `node_exporter` [40] and offers such functionality compatible with Prometheus.

While the tool already works for device metrics, some key information are missing to make it usable for this work. Following *req. M1* the script was extended with additional export capabilities:

- `prometheus-node-exporter-lua-openwrt`[41]: Exports installed firmware version and hardware model to spot outdated devices within the monitor and have an overview of used hardware.
- `prometheus-node-exporter-lua-bmx7`[42]: While per device metrics are a standard requirement for monitoring, this work needed a way to monitor the network structure as well. The created *BMX7* plugin uses the

[39] https://prometheus.io/docs/instrumenting/exposition_formats/
[40] https://github.com/prometheus/node_exporter
[41] https://github.com/openwrt/packages/pull/6007
[42] https://github.com/openwrt/packages/pull/5578

routing protocols JSON output stored in `/var/run/bmx7/json/` to export connections and resource usage of the routing service. This allows to view network development over time, how link quality changes and CPU and memory increase during network growth.

- `prometheus-node-exporter-lua-textfile`[43]: Additionally a plugin was created to generically export metrics from bash scripts or static text files. This allows to monitor further metrics like node owner, sensor data or device locations.

All changes were accepted upstream and all plugins are independently available in the official package repository of OpenWrt, following *req. M1* and *req. M2*.

Radio frequency separation

To avoid user traffic interfering with the backbone connection those frequencies should be separated (*req. N3*) to 2.4GHz for access points and 5GHz for mesh. This feature was not available in the LibreMesh framework so an additional package was created called `lime-smart-wifi`[44].

It scans for all attached wireless radio transmitters and checks their capabilities (supported frequencies, transmission power and channel bandwidth). With a simple heuristic approach it assigns roles to radio devices:

- **single band:** AP and mesh share the same radio.
- **dual band:** AP runs on 2.4GHz and mesh on 5GHz.
- **tri band:** like dual band but additional AP on 5GHz with different channel.
- **quad band:** dual AP and mesh each on different frequency and channel.

The tool chooses channels with maximum distance for minimal interference.

During the creation of this project the *hostapd*[45] daemon did not support to run an encrypted mesh and an encrypted AP on the same radio. Therefore the package is also required to obey *req. N2*.

As the package has no dependencies apart from the LibreMesh framework it is usable independently of this works software stack (*req M2*).

Porting Prometheus to OpenWrt Buildroot

To allow an easy installation (*req. I2*) of Prometheus on OpenWrt an architecture specific package is required.

The OpenWrt operating system manages all packages in a public Git repository. It stores per package a Makefile containing all information required for creation. So called Buildbots automatically download the git repositories and containing Makefiles and compile the packages, resulting in the official package feeds[46]. It is important that code is cross compiled which means to be compiled on a different architecture than eventually run. This is necessary as most routing devices use other architectures than x86[47].

OpenWrt Buildroot can compile various programming languages to run on targeted devices. The support for Go[48] was added in May 2018[49] and so allows porting Prometheus to OpenWrt.

[43] https://github.com/openwrt/packages/pull/5894
[44] https://github.com/libremesh/lime-packages/pull/321
[45] https://w1.fi/
[46] https://downloads.openwrt.org/snapshots/packages/
[47] https://openwrt.org/toh/start
[48] https://golang.org/
[49] https://github.com/openwrt/packages/pull/5780

Therefore the Prometheus package was ported within this project[50]. It required to understand the newly introduced features of OpenWrt's Buildroot to be compatible with the Buildbots (*req. 1*).

Network visualization with NetJson

Looking for tools to visualize network graphs, especially in the field of mesh networks, the only viable tool was NetjsonGraph.js.[51] The graph framework uses the data format NetJson[52] as input data and shows an interactive graph with additional metrics, zooming and dragging capabilities.

NetJson is a format designed for network statistics. It also allows the handling of connections between devices[53] which are then visualized by NetjsonGraph.js.

As written earlier within the design decisions 3.5, Prometheus is used for device monitoring. While it offers an API to read stored metrics, the file format is insufficiently to directly use it as input for a bidirectional graph.

For this work the tool Prometheus2NetJson (*p2n*)[54] was created. It outputs valid NetJson containing all nodes with collected metrics as well as all links and their quality. On execution *p2n* connects to a configured Prometheus instance and performs various API requests to collect all desired data. The received metrics are then merged into a single NetJson file. Next, the output is used by NetjsonGraph.js[55] and the overview table.

To show the link quality between devices the NetjsonGraph.js framework was extended to colorize the edge connection nodes depending on estimated throughput[56].

Additionally *p2n* can take a time input as relative time delta, specific time in the Unix time stamp format or following RFC3339 [13]. This enables to create NetJson of any time in the past, allowing to visualize development of the network.

For the overview page the generated NetJson is interpreted and visualized in a table. The visualization of the graph and table is done in both cases via JavaScript relieving the monitoring node from render the actual output. This implementation allows a modular use of the interface even for other NetJson files not produced by *p2n*.

Automatic monitoring of new nodes

Prometheus supports various ways for service discovery[57] meaning to automatically add monitored targets after creation. Until now there was no solution for automatically monitoring (*req. I4*) of all devices of a mesh network based on the routing protocol *BMX7*. The package `prometheus-bmx7-targets`[58] was created to automatically add *BMX7* nodes to a file readable by the Prometheus instance[59]. A minimal configuration for Prometheus is attached below in 8.3.

[50] https://github.com/openwrt/packages/pull/5903
[51] https://github.com/netjson/netjsongraph.js
[52] https://netjson.org
[53] http://netjson.org/#third/1
[54] https://github.com/aparcar/meshrc/blob/master/packages/meshrc/files/web/usr/bin/p2n
[55] https://github.com/aparcar/meshrc/blob/master/meshrc/templates/graph.html#L102
[56] https://github.com/aparcar/meshrc/blob/master/packages/meshrc/files/web/www/meshrc/static/css/netjsongraph-theme.css#L196
[57] https://prometheus.io/docs/prometheus/latest/configuration/configuration/
[58] https://github.com/aparcar/meshrc/tree/master/packages/prometheus-bmx7-targets
[59] https://github.com/aparcar/meshrc/tree/master/packages/prometheus-bmx7-targets

To work the `bmx7-json` plugin[60] is enabled which creates JSON files containing status information of the running *BMX7* daemon. The created tool monitors changes via `inotifywait`[61]. Concretely it scans the folder `/var/run/bmx7/json/originators/` where status information of all *BMX7* nodes of the network are stored. The wait command blocks until a file (node) is added or removed. Once unblocked, the output file containing Prometheus targets is updated and `inotifywait` blocks again until another change of connected nodes happens[62]. Prometheus detects changes to the target file automatically and starts scraping new entries.

Synchronize cloud via `cloud sync`

When changing connection disruptive commands like change of the mesh ID or key, it is important to happen on all nodes at the same time. If a single node misses the new configuration it is lost with outdated configurations, no longer being able to connect.

The current implementation distributes a *SMS* of the form `<received_command>-ack` to all other nodes, stating its acknowledgement of the command. On receiving an `ack` from another node, the node compares the list of known nodes with the list of received acknowledgments. Once both lists are equal the node executes the received command. This process is illustrated in fig. 12.

[60] https://github.com/bmx-routing/bmx7#json-plugin
[61] https://linux.die.net/man/1/inotifywait
[62] https://github.com/aparcar/meshrc/blob/master/packages/prometheus-bmx7-targets/files/usr/bin/prometheus-bmx7-targets

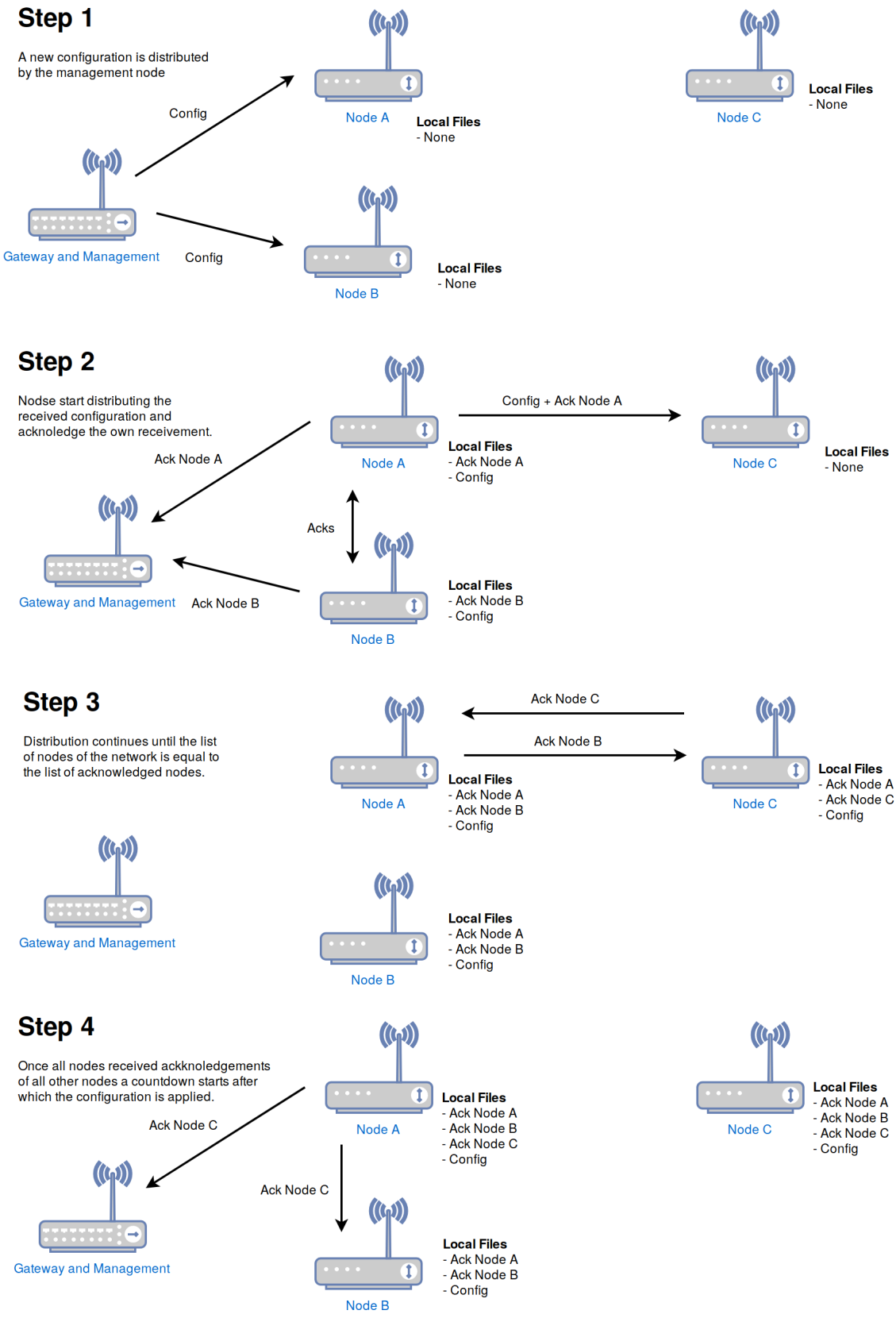


Figure 12: Cloud sync visualization

4.2 Initial configuration retrieving

To distribute the *initial configuration* the gateway node has to install *meshrc-web*. Changing settings in the web interface sends the configuration to connected nodes and also updates a locally stored archive containing these settings and offer it via a web server to directly connected devices.

Unconfigured nodes try to find all directly connected neighbors of the *LAN* bridge (`br-lan`). It is scanned via an IPv6 broadcast ping[63]. The *meshrc-initial* daemon tries to download a file called *config.tar.gz* from all found neighbors. In case the config archive was downloaded successfully, it is extracted to the devices file system root and a reboot performed after which all settings are activated.

As the `link-local` address [14, p. 11] is used only directly connected nodes can download the archive and no regular IP addresses are required. This implementation prevents external parties from accessing the archive containing passwords only making it accessible to users which have hardware access to the gateway node.

4.3 Usage of `bmx7-sms` plugin for configuration

All nodes setup with the *initial configuration* automatically trust the *BMX7* ID of the node they received the configuration from.

An ID could look like `F28A940935AA41D9019476FB397913CB45D798B593AD45544DBDC33A` which is the SHA224 hash of the nodes public key.

Files distributed via the `bmx7-sms` plugin are prefixed with this ID, so a file `mesh` becomes `F28A940935AA41D9019476FB397913CB45D798B593AD45544DBDC33A:mesh` and is stored on all other nodes in `/var/run/bmx7/sms/rcvdSms/`. If a node tries to fake the ID of another (maybe known to be trusted) node the spoof will fail due to the fact that the faking node is not in possession of the real node's private key, which is required to sign the distributed file. Badly signed files are deleted by surrounding nodes and not further distributed to the mesh.

It is trivial to scan the previously mentioned folder for files starting with one of the *trusted* IDs. In case of a match, the file name and content is examined and processed.

The following list shows all supported commands distributed as file. The file content is used as value for the command. It distinguishes between two filename prefixes for LibreMesh specific commands `lm-*` and raw commands `rc-*`. The former only change the framework file `/etc/config/lime` and applies the changes via `lime-config && lime-apply` while the latter run the command directly on the node.

Currently the following list of functions is implemented:

- `lm-hn_<short_id>`: Set Hostname of specific node with `<short_id>`.
- `lm-mk`: Set the mesh password (key) used by the backbone network.
- `lm-mi`: Set the mesh ID used by the backbone network.
- `lm-ak`: Set nodes access point password (key).
- `rc-rst`: Resets nodes to the state of the *initial firmware*.
- `rc-ssh`: Set the ssh keys allowed to login as root.

[63] `ping6 -I br-lan ff02::1`

For all of the above commands it is possible to append `_<short_id>` to the filename so it is only executed on the specific node. Notable that changing the Mesh ID or key of a single node would result in connection loss.

As of commit `9883383`[64] the `bmx7-sms` plugin only supports filenames up to a length of 16 characters. This can result in confusion if implementing own commands.

4.4 IP address collision avoidance

Following *req. N1* all nodes must automatically choose a IPv4 subnetwork from which they offers addresses to connecting clients. The IPv4 addresses must not overlap as the routing protocol wouldn't know which node handles the subnetwork. Per default[65] LibreMesh tries to avoid IP collision by picking pseudo random data, like the MD5 hash of the used AP name. While this works, the devices IP address changes every time the AP name changes. To avoid this, an additional parameter, which uses a MD5 hash of the primary MAC address, was implemented for this project. This results in an equally small chance of collision but keeps a static address for each device. The changes[66] were accepted upstream.

The attached `lime-defaults` configuration file 8.2 uses `10.%m1.%m2.1/24` which results in a collision probability of $n/(256^2)$ where n is the number of used nodes.

4.5 Alerts in case of malfunction

Adding alert support when using Prometheus is fairly trivial. The attached configuration file 8.4 sends an alert in the following three cases:

- `node_down`: A previously connected node is offline for more than 10 minutes.
  - Indicating device failure.
- `node_high_load`: The average load of the last 15 minutes is over 1.0.
  - Indicating a bottleneck or software malfunction.
- `link_bad_quality`: Estimated link quality between two nodes is under 10Mbit for 10 minutes.
  - Indicating an obstacle may started to block direct line of sight or antenna malfunction.

The alert system can use any metric collected by Prometheus and so can easily be extended to custom needs[67] in other use cases (*req. M2*).

[64] https://github.com/bmx-routing/bmx7/blob/9883383dc26df16da67b9ef7ba99efe62f79c4e7/lib/bmx7_sms/sms.h#L25
[65] https://github.com/libremesh/lime-packages/blob/develop/packages/lime-system/files/usr/lib/lua/lime/network.lua#L41
[66] https://github.com/libremesh/lime-packages/pull/349
[67] https://prometheus.io/docs/alerting/configuration/

5 Collaboration

This work exists due to the collaboration with multiple individuals and communities working on various aspects in the area of mesh technologies. Instead of reinventing existing solutions for the initially described project requirements, I decided to evaluate established approaches and extend their functionality if possible to fit the stated needs.

5.1 LibreMesh extension and fixing

As explained before in the section design decisions 3.5 I worked in various areas[68] of LibreMesh[69] and added multiple features or fixed existing bugs.

The most important bug fix regarding this project was the fixing of interface detection of *BMX7*[70]. To successfully estimate the possible bandwidth of a network interface *BMX7* has to know about the nature of the interface (cable or wireless). Before the fixing patch[71] all wireless interfaces would be interpreted as a cable and wrongly diagnosed with full Gigabit bandwidth making monitoring of link quality useless.

5.2 Encrypted mesh networks

Started with an issue on GitHub[72] regarding an encrypted mesh network the OpenWrt core developer Daniel Golle[73] and me started to investigate the problem together. With various tests and bug fixes in an especially for this issue setup test-bed, we eventually created a working version and closed the initial issue. This also showed the path for the backbone encryption of this project.

5.3 Working at the UPC Barcelona

During my exchange semester in Barcelona I could highly benefit from the collaboration with the network laboratory at the UPC[74]. Thanks to it is members I could setup a test-bed to evaluate initial ideas for the implementation. Also I could discuss there various features and drawbacks of LibreMesh and the Quick Mesh Project[75].

Members of the community project Guifi.net[76] offered insights in their mesh setups and I had the chance to discuss various requirements for a useful monitoring solution. Furthermore I got the chance to run the software created within this project on various networks to have a more complex test-bed.

[68] https://github.com/libremesh/lime-packages/commits?author=aparcar
[69] https://libremesh.org
[70] https://github.com/libremesh/lime-packages/issues/335
[71] https://github.com/bmx-routing/bmx7/pull/12
[72] https://github.com/libremesh/lime-packages/issues/208
[73] https://github.com/openwrt/openwrt/commits?author=dangowrt
[74] https://upc.edu
[75] https://qmp.cat
[76] https://guifi.net

6 Evaluation

After implementing an initial prototype of the software stack it was iteratively evaluated and checked for functionality. This section consists of two parts, the summary of the monitoring and the management part.

The former was tested within an actively used mesh network while the latter could only be tested in a test-bed with perfect conditions due to the lack of hardware and location changes.

6.1 Monitoring evaluation

In various parts of Leipzig exists mesh infrastructure handled by Freifunk[77] and a wireless internet service provider[78]. The monitoring system was installed on parts of this infrastructure as it is not putting the devices connectivity at risk. It merely requires adding the read only exporter per device. The underlying mesh routing protocol used in the network was already *BMX7*, so no additional configuration was required.

The figures 4, 5 and 6 are screenshots of the *meshrc* interface running in the previously described network. Collected data is consistent with manually gathered information, therefore the output can be considered valid.

As fig. 5 shows the network consists of various devices, partly used for the backbone connections (*Loco M XW* and *Nanostation M XW*), access points (*TL-WDR3600*) or both (*ZBT-WG3426*). Devices are connected as shown in fig. 4 where backbone distances vary between 100 and 2000 meters.

This shows a shortcoming of the current monitoring approach as the distances between devices are not visualized. The metric is not available and so could not be implemented. However a future version of the monitoring view could be extended to handle the metric if available or calculate distances based on locations provided by a static file containing location coordinates.

While setting up the monitoring system a high load on node *liebmann92-m5* was discovered and investigated. A bug of the LibreMesh firmware caused on that device the constant crash and reload of a service, resulting in high CPU usage. The malfunction later was fixed normalizing the load. A case like this document the need for an active monitoring system.

[77] https://leipzig.freifunk.net
[78] https://reudnetz.org

6.1.1 Resource requirements for monitoring

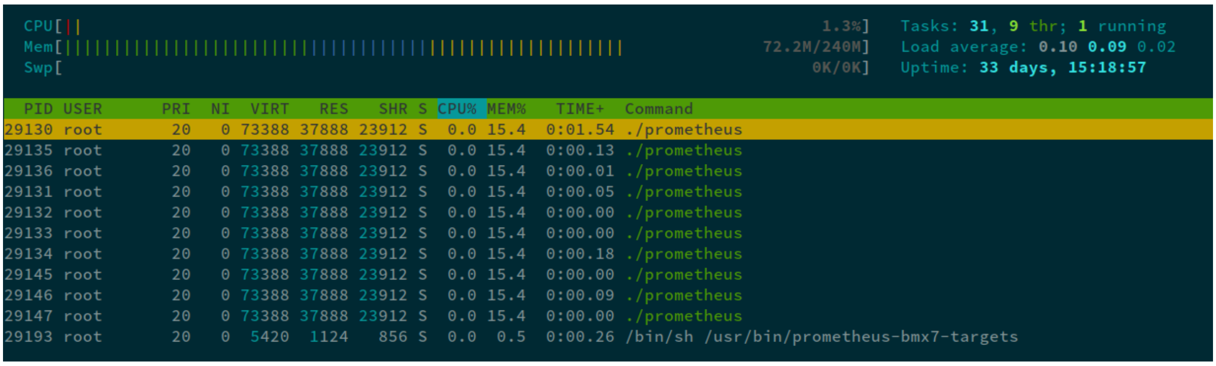


Figure 13: Unix tool `htop` shows memory and CPU usage of running Prometheus instance on monitoring node.

Figure 13 shows the low resource footprint of the running Prometheus instance. As shown, it uses less than 20% of the 240 MByte available memory, meaning under 50 MByte for the Prometheus process. This allows to use small embedded devices like the Raspberry Pi[79] to act as the monitoring node and does not require a real server setup.

This value was captured with only a single instance of the monitoring web interface opened and only 11 nodes participating in the network. Further research has to show the resource usage of the Prometheus process in a growing mesh network. Fortunately Prometheus offers to monitor its own resource usage as well, allowing to easily compare it to the number of monitored nodes.

Beneath the monitoring service the data is aggregated by the *p2n* script which runtime naturally increases with the number of nodes. It is designed to run in $O(n)$ and should therefore be scaleable with higher numbers of nodes.

A test with the test-bed described above however showed an unexpected long time for the NetJson creation.

```
root@rdntz-gateway:~# time p2n > /dev/null
real    0m 0.51s
user    0m 0.45s
sys     0m 0.02s
```

In the current implementation of *p2n* all metrics are serially retrieved which is performed via a `localhost` HTTP API request. Establishing the connection does significantly slow down the response, meaning more nodes would affect response time less than additional metrics. One could gather all metrics in parallel and use additional caching to stop unnecessary recreation of the NetJson output.

6.2 Management evaluation

Due to a location change and the fact that the mesh network in Leipzig was actively in use, it was not feasible to test and improve the management part there. For that reason a small test-bed was created allowing the

[79] https://www.raspberrypi.org/products/raspberry-pi-3-model-b-plus/

evaluation of the control instance. It consisted of the previously mentioned **Raspberry Pi**[80] (Pi) and three **Buffalo WSR-1166DHP** routers. The *Pi* was used to monitor and control the network while the routers formed a partly wired and partly wireless mesh network. Hardware choice was made based on availability in Tokyo, Japan, low price and compatibility with Prometheus, OpenWrt and the *802.11s* mesh mode.

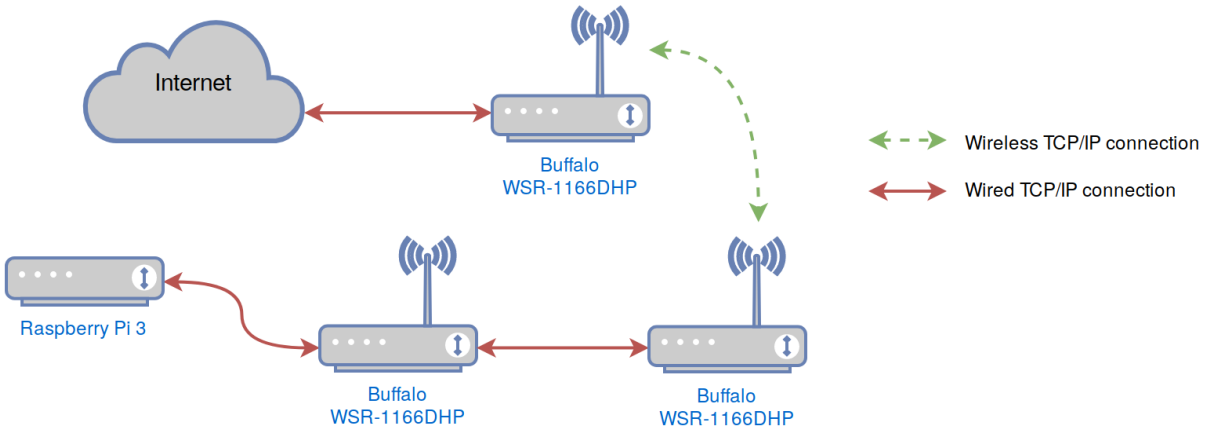


Figure 14: **Management test-bed** using 4 devices connected wired and wireless.

The test-bed was setup as illustrated in fig. 14, with distances of only a few meters. As the *Pi* features only a single LAN port it is not suitable as a gateway. As a replacement one of the nodes was used as gateway which is seamlessly possible due to LibreMesh's automatic configuration. Due to the combination of wireless and wired combination the distance between the management node *Pi* and the last *Buffalo* node connected to the Internet has as many hops (nodes in between) as possible, which gives a slightly more realistic test-bed.

6.2.1 Initial node setup

Firstly the initial configuration (*req. N1*) via the package `meshrc-initial` was tested. A generic image was created which would not setup any wireless connections, instead searching for a node connected via cable to retrieve an initial configuration. This step worked flawlessly as once a node was connected to the *Pi* it instantly downloaded the configuration, applied it and rebooted. The reboot let routers LED's flash which signals implying configuration retrieval was successful.

[80] https://www.raspberrypi.org/products/raspberry-pi-3-model-b-plus/

Listing 1: Example of initial config download.

```
daemon.info meshrc-initial: Try neighbor fe80::<mac1>:
daemon.err meshrc-initial: Downloading 'http://[fe80::<mac1>%br-lan]:8123/config.tar.gz'
daemon.err meshrc-initial: Connecting to fe80::<mac1>:8123
daemon.err meshrc-initial: Connection error: Connection failed
daemon.info meshrc-initial: Try neighbor fe80::<mac2>:
daemon.err meshrc-initial: Downloading 'http://[fe80::<mac>%br-lan]:8123/config.tar.gz'
daemon.err meshrc-initial: Connecting to fe80::<mac2>:8123
daemon.err meshrc-initial: Writing to 'config.tar.gz'
config.tar.gz        100% |*****************************|   670   0:00:00 ETA
daemon.err meshrc-initial: Download completed (670 bytes)
daemon.info meshrc-initial: successfully downloaded config
```

Below log paste shows how a node finds two neighbors (MAC addresses substituted) and tries to download the configuration. The second node is the *Pi*. All three devices are connected via cable.

Changing option in the management interface fig. 7 instantly updated the archive containing all settings and started also to distribute it over *BMX7*. This ensures that new initialized nodes as well as already setup nodes contain the same configurations.

6.2.2 Testing the `cloud-sync` feature

Due to the small test-bed and perfect conditions regarding connectivity only limited tests were possible. However, stopping a node manually from sending a command acknowledgment lets other nodes wait, which is the intended behavior. Once the *ack* is sent, other nodes apply the new settings.

The testing however brought up the difficulty of *split brain* prevention, a state where a distributed system is unintentionally clustered due to disparate states of configuration. In this case, the `cloud-sync` feature acknowledges the successfully received configuration, however the distribution of acknowledgments is no further verified. Meaning, a node could reboot before it distributed other nodes acknowledgments, resulting in other nodes to infinitely wait for missing *acks*. To work around this issue, the current implementation starts a timer once all acknowledgments are received, taking for granted that all nodes are reachable as they sent an *ack*. During the countdown ticking, *acks* are further distributed, assume all nodes received all *acks* within the countdown.

As *BMX7* rapidly forwards received configurations a short countdown of 5 seconds were chosen, however this should be further evaluated in a real test-bed with more devices.

6.2.3 LibreMesh with encrypted mesh

As LibreMesh is an established framework it mostly worked as excepted and the biggest extension of functionality in the package `lime-smart-wifi` was the possibility to encrypted mesh connection (*req. N2*). It turned out to perform very well with a stable connection and no immense performance drawback.

Only additional node is the initial connection time. As both nodes communicate encrypted they have to exchange a secret via a Diffie–Hellman key exchange, resulting in extra calculation before the connection to another node is established. Depending on the node's computation power, this can take up to a minute, while unencrypted mesh connections are nearly instantly ready.

7 Conclusion

All initially set requirements where fulfilled and so the work should be seen as successful! It is usable for the target group and flexible enough to extend it to additional needs. During the development and collaboration with community network, some already stated interest to use the software.

7.1 Solved project requirements

- req. N1 **Zero-Conf Setup:** Extended via the creation of *initial configurations* which are generically included in all created firmware images.[81]
- req. N2 **Encryption:** Within this project the SAE encryption of *802.11s* [15, pp. 17–19] connections were exceedingly tested in various test-bed setups and reached an usable state.
- req. N3 **Frequency separation:** The newly `lime-smart-wifi`[82] package separates the frequency used by the access points from the one used by the mesh backbone connection.
- req. N4 **Simple setup:** All created tools for the monitoring are installable via the `opkg`[83].
- req. I1 **Limited access:** Fulfilled by using OpenWrt authentication system via *ubus*.
- req. I2 **Simple setup:** Feasible installable via an *opkg* package.
- req. I3 **Network graph:** Implemented via the `meshrc-web` package and NetJsonGraph.js.
- req. I4 **Automatic monitoring:** Implemented via the `prometheus-bmx7-targets` package.
- req. I5 **Alerts:** Possible via the ported `prometheus-alertmanager` package. The alert manager requires individual configuration based on the desired notification channel[84].
- req. I6 **Network configuration:** Implemented via the `meshrc-cli` and `meshrc-web` package.

7.2 Created, extended and ported packages

Here is a list of software which was created, extended or ported by me. I'd like to emphasize the work on the extending and porting part, as it required to learn new programming languages and to understand building environments and processes.

7.2.1 Package creation

The web front-end is (apart from the NetjsonGraph.js library) a new created code. It was packed to be installable via the OpenWrt package manger called `opkg`.

- `meshrc-web`: Web interface giving an overview of the network and allowing the configuration of various parameters.

[81] https://github.com/libremesh/network-profiles/tree/master/meshrc/node
[82] https://github.com/libremesh/lime-packages/tree/develop/packages/lime-smart-wifi
[83] https://openwrt.org/docs/guide-user/additional-software/opkg
[84] https://prometheus.io/docs/alerting/configuration/#%3Cemail_config%3E

- `meshrc-cli`: Command line interface which is used by the web interface. Allowing more detailed configuration of nodes. Also used for the distribution of the *initial configurations*.
- `meshrc-client`: Daemon to be run on clients which checks for incoming commands and executes them.
- `meshrc-inititial`: Daemon to be run on clients which automatically download and setup an *initial configuration* which is supplied by the `meshrc-cli`.
- `prometheus-bmx7-targets`: Automatically adds active nodes of a *BMX7* mesh as targets to a running *Prometheus* instance.

7.2.2 Package extension

In the development branch of OpenWrt already existed a tool called `prometheus-node-exporter-lua` which was missing some features required for this project. The functionality was extended and the created code will be available in the next stable release of OpenWrt.

- `prometheus-node-exporter-lua-bmx7`: Exports *BMX7* related metrics like active links, CPU & memory usage of the `bmx7` daemon and loaded plugins.
- `prometheus-node-exporter-lua-textfile`: Exports content of existing text files already in a *Prometheus* compatible format. Uses a generic way to export output of shell scripts or static information like node locations coordinates.
- `prometheus-node-exporter-lua-openwrt`: Exports OpenWrt specific metrics like currently installed OS release and node model.

7.2.3 Package porting

As Prometheus is written in Go, it is possible to run it on various devices if the binary package was specially compiled for that architecture. Normally the architecture specific compilation is done by the OpenWrt Buildroot[85], but until May 2018 no Go code compiling was available[86].

After support became available I ported Prometheus and the alertmanger to OpenWrt, adapting to the new build methods[87]:

- `prometheus`
- `prometeus-alertmanager`

7.3 Projects drawbacks and limits

Even though I could implement previously stated requirements, during the process I recognized various flaws of the design decisions 3.5 which are shortly covered in the following sections.

[85] https://wiki.openwrt.org/doc/techref/buildroot
[86] https://github.com/openwrt/packages/pull/5780
[87] https://github.com/openwrt/packages/pull/5903

7.3.1 Plain text configuration

With the current configuration approach all commands, including passwords, are sent unencrypted to all nodes of the network. It is within this project's design as the mesh network requires an authentication to participate. However the current approach wouldn't be suitable for bigger network where multiple *mesh clouds* are connected, sharing an uplink connection, but are managed by different individuals or communities. Within the scope of this project this is not a problem due to mesh encryption.

7.3.2 No roaming support between devices

Moving as a client between access points results in a short loss of connection as the clients requests a new IP from the newly connected access point. While for web browsing this shouldn't be a problem, VoIP telecommunication might be disrupted.

The LibreMesh framework offers a roaming solution for that by using B.A.T.M.A.N Advanced[88] (batadv). While the use of *batadv* with *BMX7* is possible, it is currently insufficient tested. Further work may provide a working setup including *BMX7* and *batadv* for full roaming support.

Another approach could be the use of Mobile IPv6 [16], using the *gateway node* as *Home Agent*. However, this requires further research, implementation and by far exceeds the projects scope.

7.3.3 Complicated initial router flash

As this project does not use specialized hardware, the preinstalled software is not suitable to be used in a mesh setup. A custom firmware flash is required, which requires specific device knowledge and is usually not generically possible.

[88] https://www.open-mesh.org/projects/batman-adv/wiki/Doc-overview

Figure 15: Serial connection with bricked router.

Figure 15 shows a router which struck a power cut during firmware flashing. The router wasn't able to boot anymore and so was opened to connect a serial console to directly enter commands without the need of a working network connection.

A possible solution would be to focus device homogeneity in a mesh setup or use specialized hardware with the mesh framework LibreMesh preinstalled[89].

7.3.4 Insecure connection to configuration interface

While this project solved authenticated configuration of mesh nodes, the configuration between the administrators device and the gateway running the configuration web interface is unencrypted. With the current software a secure connection can still be used by establishing secure shell connection (*SSH*) with port forwarding to the gateway node.

[89] https://librerouter.org/

7.3.5 BMX7 dependencies

The entire project is build around the routing protocol *BMX7* which creates a strong dependence on exactly this software. As mentioned earlier there are various routing protocols, which may outperform the current choice. All created tools, however, depend on *BMX7* JSON output and its API.

Future versions could be designed in a more flexible way, just like the *LibreMesh* framework itself.

7.3.6 No IPv6 connectivity for clients

The current implementation does not offer the handling of IPv6 addresses for clients. While IPv6 should be granted in today's network, it is a long standing issue of the LibreMesh distribution[90]. Therefore it would exceed the scope of this work and will be tackled apart from this work.

7.4 Outlook

This project can be seen as an advanced proof of concept. It is functional and fulfills the initial's goals. However, various rough edges could be improved and are shortly explained in the following.

7.4.1 Interface design

Based on practical experiences the web interface could be extended to show a better collection of metrics, helping network administrators to maintain the network in an easier fashion. This could happen in cooperation with the authors of previous works [6] [7].

7.4.2 Grafana to visualize metric graphs

The current overview offers a specific time frame of metrics, but lacks an over time view. Grafans graphs could be embedded into the current interface, a quick illustration for a possible output in Figure 16.

[90] https://github.com/libremesh/lime-packages/pull/71

Figure 16: Graph created by Grafana shows gateway traffic over time.

7.4.3 Test scalability of routing BMX7

It is possible to see the CPU and memory usage of the running mesh routing protocol *BMX7* process. The resource usage of the routing daemon can be monitored and the limitation of scalability can be estimated. This is especially useful with historical information about the number of links and nodes.

7.4.4 Sensor network

Using the `prometheus-node-exporter-lua-textfile` daemon allows to monitor generically any floating point based metric and the delivery of labels containing text information. The project could be extended to be used in a sensor network tracking data like air quality or car traffic.

7.4.5 Encrypted transport layer

While the wireless connection between nodes is encrypted, the tunnels created by the routing protocol to connect nodes are unencrypted, allowing evil nodes to spy on forwarded traffic. This prevents the current network setup from being open, to allow others to participate in the mesh, while staying secure. A possible solution is the use of Wireguard [17] allowing encrypted peer to peer connections. The setup would define a trusted gateway, where all clients nodes establish a encrypted connection to. From this point on, no node can eavesdrop and so the network can be opened to non-self-controlled nodes.

7.4.6 Opportunistically encrypted open hotspots

Currently both the mesh connection as well as the AP is encrypted to provide secure data transportation for connected users. While the previously mentioned encrypted transport layer could open the former, the AP must remain encrypted to prevent attackers from monitoring users traffic. This approach requires authentication and thereby a password - disallowing open hotspots.

A possible solution to provide open hotspots and encrypted wireless connections is the newly introduced *Opportunistic Wireless Encryption* [18], which could solve this in the future.

7.4.7 Modular configuration distribution

To avoid the dependence on `bmx7-sms` the configuration distribution should be implemented in a more modular way, allowing the usage of other mesh routing protocols. Two different ways are possible:

- The tool *Alfred*[91] allows distribution of arbitral data within a mesh network without relying on a specific routing protocol.
- A central server (within the mesh network) which is periodically polled by nodes for new configurations.

Both approaches lack authentication of received configuration files. To counter this lack it is possible to use *usign*[92], which comes preinstalled on all OpenWrt nodes. The tool is used to verify packages downloaded by the package manager *opkg*, but may also check other signed files, like received configurations.

[91] https://www.open-mesh.org/projects/alfred/wiki
[92] https://git.openwrt.org/project/usign.git

8 Attachments

Attached is a list of advises and configuration which enable an interested user to setup a mesh network. Just as the design decision 3.5 all recommendations are based on evaluations with existing mesh communities.

8.1 Recommended hardware for mesh networks

In the following a short suggestion on what hardware can be used. As the hardware support of the *802.11s* standard broadly varies it is highly recommended to only use devices, concretely chip-sets and WiFi cards, which are known for good support.

This project does not cover deeper analysis of hardware support regarding mesh setups, however, during the software development some hardware worked better then other. For further tests and interested developers I'd like to point out the experiences I made.

8.1.1 Mesh nodes

Recommendations for nodes which are used providing access points and connections to the backbone network.

Dual-Band

As the mesh approach of this project mostly involve connections over WiFi, all devices should be equipped with potent wireless cards.

Simply said, the 2.4GHz frequencies pass better through obstacles while offering a smaller bandwidth to transport data. On the other hand 5GHz has a higher throughput with a shorter range[93].

In a practical setup most nodes would be positioned so that they have a direct line of sight. Clients likely move around and signal can be reduced by obstacles. For this reason the *backbone* network should run on 5GHz while the access points use 2.4GHz. As a result the used hardware should have at least two bands covering both frequencies.

Storage and memory

The used mesh software, discussed below, requires slightly more space than devices used as regular home routers due to the used routing protocol and it is dependencies of cryptographic libraries. Therefore additional storage and memory is recommended. The created firmware image has a size of about 6MB so devices with 8MB are perfectly suited.

Optional: outdoor proof

This requirement can be archived in various ways using waterproof boxes or even plastic bags. The waves of 2.4GHz frequencies are just the same as of microwave ovens used in kitchens and so easily pass plastic shells with no problematic reduction of signal.

[93] https://www.speedguide.net/faq/is-5ghz-wireless-better-than-24ghz-340

Various companies offer outdoor routers with a water and sunlight proof case.[94]

Optional: Power over Ethernet (PoE)

Most router devices use a power supply of 12 Volt and up about 1 Ampere. The power supply can be connected directly to the device or instead PoE is used to supply the needed current via a regular patch cable. The advantage of this is to connect devices to power and LAN with a single cable.

Special hardware

There are various specialized devices for mesh networks, from professional companies[95][96] to communities creating own hardware.[97] While companies and communities may use equal protocols like *802.11s* the software is vastly different and incompatible.

Most devices offer dual band or even triple band (2x5GHz) to offer access points in different frequencies.[98]

Home routers

Instead of using specialized hardware, which often comes to higher prices, it is also possible to use a wide range[99] of home router which fulfill the requirements above. As mentioned before the support for *802.11s* and compatibility for OpenWrt must be granted.

Suggested node hardware

The following home routers are found of ideal hardware in terms of support, fulfilling all requirements:

- Buffalo WSR-1166DHP[100] (AC)
- Xiaomi MiWifFi 3G[101] (AC)
- TP-LINK TL-WDR4300[102]

Home router flashing

As these devices are usually sold with branded and locked down software, devices must be flashed with the software mentioned below. This process varies between devices, vendor and CPU.[103][104][105].

[94] https://www.ubnt.com/products/#unifi
[95] https://www.openmesh.com/products/wifi
[96] https://unifi-mesh.ubnt.com/
[97] https://librerouter.org/
[98] https://librerouter.org/document/specifications-sheet-v6/
[99] https://openwrt.org/toh/start
[100] https://wikidevi.com/wiki/Buffalo_AirStation_WSR-1166DHP
[101] https://wikidevi.com/wiki/Xiaomi_MiWiFi_3G
[102] https://wikidevi.com/wiki/TP-LINK_TL-WDR4300
[103] https://wiki.openwrt.org/toh/tp-link/tl-wdr4300#initial_installation
[104] https://wiki.openwrt.org/toh/xiaomi/mini#quick_openwrt_installation
[105] https://patchwork.ozlabs.org/patch/704793/

8.1.2 Gateway

The gateway node can be a node just like all other node as well. The mesh setup enables any node to serve a connected uplink to all other nodes within the mesh. While this is possible and adds flexibility to the network setup it does make sense to use special devices which serve as a gateway.

Gateway requirements

These requirements are additional but optional to the requirements of a *mesh node*. Gateways may have no wireless radio installed at all, if connected to other *mesh nodes* via cable.

Network address translation (NAT)

Independent of the IP protocol used within the network a gateway has to perform network address translation for IPv4 connections. This requires additional CPU power which is rare on the used home routers mentioned above. The gateway has to have noticeably more processor power than nodes to avoid becoming a bottleneck of the network.

CPU architecture

While OpenWrt supports a wide variety of hardware architectures[106] the used monitoring software Prometheus is written in Go[107] and so only supports a fraction of OpenWrts architectures[108].

Suggested gateway hardware

The previously mentioned APU Board from PCEngines[109] are ideal for the projects setup. The used architecture is x86/64 and so supports Go. A miniPCI SSD can expand the internal storage for logging and a miniPCI wireless adapter can be used to connect to the rest of the mesh network[110] allowing full 802:11ac speed.

[106] https://downloads.lede-project.org/snapshots/targets/
[107] https://golang.org/
[108] https://github.com/golang/go/wiki/MinimumRequirements#architectures
[109] https://www.pcengines.ch/apu2.htm
[110] https://www.asiarf.com/1W-High-Power-MT7612E-802-11ac-n-WiFi-Mini-PCIe-Module-2-4GHz-MT7612-G01-product-view-406.html

8.2 /etc/config/lime-defaults

Listing 2: Minimal lime-defaults file to be usable by meshrc.

```
config lime system
    option hostname 'meshrc-%m1%m2%m3' # initial hostname is MAC based
    option domain 'lan'
    option keep_on_upgrade 'libremesh base-files-essential /etc/sysupgrade.conf'

config lime network
    option primary_interface eth0
    option main_ipv4_address '10.%m1.%m2.1/24' # IP is MAC based
    option main_ipv6_address 'fd43:1508:%m1%m2:%m300::/64'
    list protocols ieee80211s # enable mesh
    list protocols lan
    list protocols bmx7
    list resolvers '1.1.1.1'  # dns providers to use
    option bmx7_over_batman false
    option bmx7_pref_gw none
    option bmx7_mtu '1500'

config lime wifi
    option legacy_rates '0'   # enable wider spectrum
    option country 'DE'
    option distance '1000'
    option ap_ssid '%H'       # set AP name to hostname
    option ap_encryption 'psk2+aes'
    option ieee80211s_mesh_fwding '0'
    option ieee80211s_mesh_id 'mesh'
    option ieee80211s_encryption 'psk2+aes'

config smart_wifi 'smart_wifi'
    option mesh_2ghz '0'            # disable mesh on 2.4 ghz
    list channels_2ghz '13 9 5 1'   # channels to use if multiple radios exists
    list channels_5ghz '128 100'    # same for 5ghz frequencies
```

8.3 /etc/prometheus.yml

Listing 3: Example configuration for Prometheus to monitor devices of the mesh.

```
global:
  scrape_interval: 2m               # Scrape nodes every 2 minutes

rule_files:
  - /etc/rules.yml

scrape_configs:
  - job_name: "mesh"
    file_sd_configs:
    - files:
      - "/tmp/targets_bmx7.json"    # File created by prometheus-bmx7-targets
```

8.4 /etc/rules.yml

Listing 4: Example configuration file for alerts.

```
groups:
- name: rules.yml
  rules:
  - alert: node_down
    expr: up{job="mesh"} == 0
    for: 10m
    annotations:
      summary: node {{ $labels.hostname }} is down
  - alert: node_high_load
    expr: load_15{job="mesh"} > 1
    annotations:
      summary: node {{ $labels.hostname }} has load of over 2.0
  - alert: link_bad_quality
    expr: bmx7_link_rxRate{job="mesh"} < 1000*1000
    for: 10m
    annotations:
      summary: link between {{ $labels.source }} and {{ $labels.target }} slow
```

8.5 Document creation

This document is written in Markdown with additional segments LaTex. The Markdown input is converted to LaTex via pandoc[111] using the *Eisvogel* template[112]

All photos, screenshots and images are self created.

- All network graphs were created with draw.io[113].
- Sequence diagrams were created via the Mermaid Live Editor[114].

[111] http://pandoc.org/
[112] https://github.com/Wandmalfarbe/pandoc-latex-template/blob/master/eisvogel.tex
[113] https://draw.io
[114] https://mermaidjs.github.io/mermaid-live-editor/

Erklärung

Ich versichere hiermit, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die im Literaturverzeichnis angegebenen Quellen benutzt habe. Alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten oder noch nicht veröffentlichten Quellen entnommen sind, sind als solche kenntlich gemacht. Die Zeichnungen oder Abbildungen in dieser Arbeit sind von mir selbst erstellt worden. Diese Arbeit ist in gleicher oder ähnlicher Form noch bei keiner anderen Prüfungsbehörde eingereicht worden.

Paul Spooren

References

[1] J. Kos, M. Milutinović, and L. Čehovin, “Nodewatcher: A substrate for growing your own community network,” *Computer Networks*, vol. 93, pp. 279–296, 2015.

[2] D. Vega, L. Cerda-Alabern, L. Navarro, and R. Meseguer, “Topology patterns of a community network: Guifi. net,” in *Wireless and mobile computing, networking and communications (wimob), 2012 ieee 8th international conference on*, 2012, pp. 612–619.

[3] G. R. Hiertz *et al.*, “IEEE 802.11 s: The wlan mesh standard,” *IEEE Wireless Communications*, vol. 17, no. 1, 2010.

[4] P. Escrich Garcia, “Quick deployment network using manet,” 2012.

[5] A. Neumann, “Cooperation in open, decentralized, and heterogeneous computer networks,” 2017.

[6] R. P. Centelles, V. Oncins, and A. Neumann, “Enhancing reflection and self-determination in a real-life community mesh network,” *Computer Networks*, vol. 93, pp. 297–307, 2015.

[7] L. Cerdà-Alabern, A. Neumann, and P. Escrich, “Experimental evaluation of a wireless community mesh network,” in *Proceedings of the 16th acm international conference on modeling, analysis & simulation of wireless and mobile systems*, 2013, pp. 23–30.

[8] P. Jacquet, “Optimized link state routing protocol (olsr),” 2003.

[9] J. Chroboczek, “The babel routing protocol,” 2011.

[10] R. Baig, “Evaluation of dynamic routing protocols on realistic wireless topologies,” *Universitat Autònoma de Barcelona*, 2012.

[11] I. RFC4291, “Version 6 addressing architecture,” Standards Track, IETF Network Working Group, 2006.

[12] A. Neumann, E. López, L. Cerdà-Alabern, and L. Navarro, “Securely-entrusted multi-topology routing for community networks,” in *Wireless on-demand network systems and services (wons), 2016 12th annual conference on*, 2016, pp. 1–8.

[13] G. Klyne and C. Newman, “Rfc 3339: Date and time on the internet: Timestamps,” *The Internet Society, Request for Comments Jul*, 2002.

[14] R. M. Hinden and S. E. Deering, “IP version 6 addressing architecture,” 2006.

[15] W. K. Tan, S.-G. Lee, J. H. Lam, and S.-M. Yoo, “A security analysis of the 802.11 s wireless mesh network routing protocol and its secure routing protocols,” *Sensors*, vol. 13, no. 9, pp. 11553–11585, 2013.

[16] C. Perkins, D. Johnson, and J. Arkko, “Mobility support in ipv6,” 2011.

[17] J. A. Donenfeld, “Wireguard: Next generation kernel network tunnel,” in *Proceedings of the 2017 network and distributed system security symposium, ndss*, 2017, vol. 17.

[18] D. Harkins and W. Kumari, “RFC 8110-opportunistic wireless encryption,” 2017.

List of Figures

1 **Wireless mesh network** - Possible setup of a mesh network which offers inter mesh connections to clients and via a gateway Internet access. . . . 9
2 **Star and tree topologies** . . . 10
3 meshrc Login page. All interaction, receiving monitoring data or changing settings, requires authentication which is handled by UBUS. . . . 15
4 **meshrc Graph now** - Shows the current state of the network rendered based on the JSON output of *p2n*. . . . 16
5 **meshrc Overview now** - Rendered table based on the JSON output of *p2n*. . . . 17
6 **meshrc Overview 24 hours ago** - The NetJson was generated with the relative timestamp `24h` to show a previous state of the network. . . . 18
7 **meshrc Configuration** - Basic configuration is possible via the configuration interface. . . . 19
8 **meshrc Node configuration** - Per node configuration with limited functionality to prevent user from breaking connectivity. . . . 19
9 **Tool stack** - Tools installed on the management and gateway node or on a generic node. . . . 21
10 **Login sequence** - The user has to provide the *root* password authenticate in UBUS. After a successful authentication NetJson is requested from the UBUS daemon. UBUS runs the below discussed NetJson generator *p2n* and forwards it is output to the web interface. Via JavaScript the JSON data is rendered to HTML and presented to the user. . . . 22
11 **Configuration sequence** - User has to provide a valid password to login. On success the configuration is loaded in the configuration form. To apply new settings changed values are submitted to the UBUS daemon which then executes the CLI. *BMX7* is requested by the CLI to distribute the new configuration files as described below. . . . 23
12 **Cloud sync visualization** . . . 27
13 Unix tool `htop` shows memory and CPU usage of running Prometheus instance on monitoring node. 32
14 **Management test-bed** using 4 devices connected wired and wireless. . . . 33
15 Serial connection with bricked router. . . . 38
16 Graph created by Grafana shows gateway traffic over time. . . . 40

List of Tables

1 **Mesh distribution features:** *[Y]* exists, *[N]* missing, *[U]* unknown, *[E]* extended within this work, *[I]* implemented within this work. . . . 13

Listings

1 Example of initial config download. . . . 34
2 Minimal lime-defaults file to be usable by meshrc. . . . 45
3 Example configuration for Prometheus to monitor devices of the mesh. . . . 45
4 Example configuration file for alerts. . . . 46