A BATMAN Testbed with Assisted Deployment and Metric Measurement

Project Report

Group 20gr550

Aalborg University
Department of Electronic Systems
Fredrik Bajers Vej 7B
DK-9220 Aalborg Ø

Department of Electronic Systems
Fredrik Bajers Vej 7B
DK-9220 Aalborg Ø
https://es.aau.dk

AALBORG UNIVERSITY
STUDENT REPORT

Title:
A BATMAN Testbed with Assisted Deployment and Metric Measurement

Theme:
Complex Distributed Systems

Project Period:
Fall semester 2020

Project Group:
20gr550

Participants:
Frederik Rentzø Fagerlund
Julian Jørgensen Teule
Marius Frilund Hensel
Nicholas Bernth Strømgaard Hansen
Victor Büttner

Supervisors:
Sebastian Bro Damsgaard
Tatiana Kozlova Madsen

Copies:

Number of Pages: 90

Date of Completion:
December 18, 2020

Abstract:

Mesh routing allows for multi-hop ad-hoc networks avoiding reliance on existing infrastructure. However, mesh network routing is not easy. Multiple routing algorithms have been proposed and multiple implementations have been created. This report investigates the possibility of developing a testbed for the mesh network routing protocol BATMAN. The testbed was developed with the goal of assisted deployment of a BATMAN network and continuously collecting data. A deployment system was used for assisted deployments, and a tracker was developed which collects both local BATMAN state and headers from packets sent by each node. In order to verify the applicability of the system, multiple tests were conducted. Using the deployment system proved successful for getting a working testbed running with nodes using BATMAN, as well as the developed tracking software. Once the testbed was deployed, three of four selected scenarios could be recreated, data could be collected throughout, and metrics could be calculated from this data.

Nomenclature

Abbreviation	*Meaning*
ACID	Atomic, Durable, Isolated, Consistent
alfred	Almighty Lightweight Fact Remote Exchange Daemon
API	Application Programming Interface
AS	Autonomous System
BATMAN	Better Approach To Mobile Ad-hoc Networking
CIDR	Classless Inter-Domain Routing
CLI	Command Line Interface
DHCP	Dynamic Host Configuration Protocol
DNS	Domain Name System
HTTP	Hypertext Transfer Protocol
IEEE	Institute of Electrical and Electronics Engineers
IETF	Internet Engineering Task Force
IPv4	Internet Protocol version 4
IPv6	Internet Protocol version 6
ISP	Internet Service Provider
IX	Internet Exchange
JSON	JavaScript Object Notation
JSONL	JavaScript Object Notation Lines
LL	Link Local
LLC	Logical Link Control
MAC	Media Access Control
MTU	Maximum Transmission Unit
NTP	Network Time Protocol
OGM	Originator Message
OLSR	Optimized Link State Routing Protocol
OOP	Object-Oriented Programming
OS	Operating System
OSI	Open Systems Interconnection
PDF	Portable Document Format
PTP	Precision Time Protocol
RFC	Request for Comments
RREQ	Route Request Packets
RTT	Round-Trip Time
SHA	Secure Hash Algorithm
SQL	Structured Query Language
SSH	Secure Shell
TCP	Transmission Control Protocol
txpower	transmission power

TQ	Transmission Quality
TT	Translation Table
VPN	Virtual Private Network
YAML	YAML Ain't Markup Language

Contents

Preface

The "BATMAN" term is used to encompass both the BATMAN IV routing protocol and its leading implementation, batman-adv. Other versions of the BATMAN routing protocol such as BATMAN V will be referred to as such.

If something is left out in the middle of a snippet `[SNIP]` is used, for instance:

```
1 int get_the_thing() {
2 [SNIP]
3     return thething;
4 }
```


Frederik Fagerlund
<ffager18@student.aau.dk>

Julian Teule
<jteule18@student.aau.dk>

Marius Hensel
<mhense15@student.aau.dk>

Nicholas Hansen
<nbsh18@student.aau.dk>

Victor Büttner
<vbattn18@student.aau.dk>

Aalborg University, December 18, 2020

1 Introduction

The internet these days is indispensable for most people, which in turn means that large amounts of resources and infrastructure are used to ensure a fast and stable network connection for people and companies around the world. However in some situations the infrastructure of the internet is sometimes not present or desired. Examples of such situations are natural disasters where infrastructure has been destroyed, or military operations which might have demands for performance or privacy. A potential solution to this could be establishing a decentralized network, that could act as a temporary alternative to the internet. Such a network will be referred to as an *ad-hoc* network as it is not reliant on existing infrastructure.

An ad-hoc network could also be a substitute for the more centralised networks of today, removing the reliance on ISPs (Internet Service Provider). ISP controlled networks are often unwanted because of the censoring, control, and surveillance which can be placed on its users. One example of a working ad-hoc network is the non-commercial free network deployed by Freifunk. The Freifunk network is seemingly the largest instance of such a deployment with more than 34 000 active nodes split among 310 sub-communities as of 2016. It is an entirely volunteer-driven project that seeks to live up to the original design goals of the internet with a redundant, distributed, open, and neutral network. In addition, Freifunk is also free of charge (gratis) and mostly free as in freedom (libre). [1]

These decentralised networks mentioned above often utilize wireless technology to connect devices as it is seemingly simpler to setup and maintain than wired connections. In order to extend the often limited range of wireless technologies, devices must work together to route other devices' data. Network packets can then travel large distances being forwarded by multiple intermediate nodes which are also users of the network themselves. This will be referred to as *mesh* network.

Contrary to mesh networks, the internet utilizes dedicated infrastructure for routing user network packets. Oftentimes a user pays for the internet, which means that some reliance is placed on the infrastructure to have the data forwarded both to and from the device used. In most mesh networks the user must trust other devices to forward the data correctly, Through a mesh network routing algorithm could make incentives for forwarding other users' data.

Both mentioned networks use *multi-hop* to forward data to the desired destination. One challenge with mesh networks is the routing algorithm used to allow cooperation between nodes. As such, mesh networks and the routing required to make it work will be explored.

1.1 Mesh Networks

Ad-hoc describes independence from existing infrastructure in a network, and mesh describes a topology where data is forwarded by other participating nodes. This is in contrast to networks with a star topology where a node hierarchy exists, in the form of a central node controlling all communication. One of the main drawbacks and characteristics of a star topology is the reliance on a central node, as the network cannot exists without the central node being online.

On the contrary, mesh networks are typically designed to be able to survive any node being down, by dynamically changing network behavior on network changes, as nodes join or leave. This is achieved with routing protocols which determine how nodes exchange network information and how this information is used to route packages. Mesh routing protocols are measured according to their ability to adapt to nodes going offline, moving, or new nodes being added and their speed at doing so.

This adaptive routing is a problem as there is no central authority, so information needed for routing might not be available to all nodes. The simplest approach is for every node to rebroadcast every new message, meaning each message is flooded through the network. This method, which is called flooding, scales badly with larger networks as it utilizes network infrastructure badly.

An optimization would be to route packets along a calculated path, instead of distributing it to every node in the network. Determining this path often requires information about how nodes connect in the network, which must be distributed between nodes. This information exchange can be done on demand with a special *route request* packet which can utilize flooding to find a route to a target node. These routes can be cached for later usage, to minimize route requests and other *control traffic*. This routing approach is called *reactive* or *on-demand* routing as network information is found when needed.

Another approach is for nodes to constantly exchange network information about online nodes and where they are in the network. This has the advantage that packets can be sent immediately as each node always has the needed information available to it. However for quickly changing networks such as mobile networks information can quickly become outdated, which will cause packets to take inefficient or incorrect routes. This can be partially solved by sending network updates more quickly. This proactive exchange of network information gives this approach its name *table-driven* or *proactive* routing.

In terms of routing protocols for mesh networks multiple solutions exist using methods described earlier i.e. reactive and proactive, and sometimes a hybrid between the two. One of the more modern proactive routing protocols is BATMAN.

The previously mentioned Freifunk started to work on the BATMAN protocol in 2008 [2, 3] (or 2006 according to another source [4, 5]). In 2016 it was the protocol used by the majority of the Freifunk network [1]. With the intent to create a scalable protocol to route large interconnected mesh networks, BATMAN was created as a proactive routing protocol. It was meant to replace Freifunk's usage of the OLSR (Optimized Link State Routing) protocol.

As the Freifunk network grows with more devices, a higher demand is placed on the BATMAN protocol. It is therefore still under heavy development (as of 2020-11) for it to fulfill these demands. Research on BATMAN are therefore quickly outdated as it changed.

To assist research and development of BATMAN this project will focus on assessment of BATMAN.

1.2 Assessing BATMAN

Table 1.1 shows different methods to assess routing protocols. It is important to distinguish between the implementation and protocol, as some methods will only assess the protocol and not a specific implementation.

Method	Description	Considerations
Analytical	Mathematically describe scenarios and analytically draw conclusions.	Impractical with complex scenarios. Only as accurate as the mathematical model used. Unfit to test protocol implementation.
Simulation	Implement protocol and networking behavior in software and run scenarios against it.	Only as accurate as the implementation. Random behavior can be implemented deterministically which gives reproducible results if all inputs are known.
Testbed	Replicate scenarios in a real world environment.	Accurately captures wireless imperfections and other real world effects. Hard to reason about error sources, which hurts reproducibility. Impractical with many devices.

Table 1.1: Description and considerations of methods for assessing routing protocols.

The analytical method can work well if scenarios and protocol can easily be described mathematically. This method does not take a specific implementation into account, and implementation specific faults are therefore ignored.

Simulation implements all network and physical behavior in software, including the wanted scenarios. Link quality and other physical behavior can also be simulated deterministically, resulting in reproducible results. However the simulation platform may not support running a real implementation, and can therefore perform differently than the real implementation. Simulation also may not expose faults caused by hardware or other real world effects, such as radio interference.

A testbed solves these issues by running the implementation on real hardware, meaning it will be as close to a real world use case as possible. This can be impractical as it can require many devices and large amount of space for complex scenarios. Node movement may also be harder to emulate as devices must be moved around physically.

A testbed was deemed the best solution for evaluating the performance of BATMAN. Furthermore, the motivation of this project was initiated by a scientist at Aalborg

University, whom had requested a testbed that can evaluate the performance of BATMAN. The development of a testbed that can easily be deployed and maintained by third-party users is a non-trivial task. This project scope is therefore placed on development of a testbed targeted at BATMAN for Aalborg University.

1.3 Initial Problem Statement

This leads to the following initial problem statement:

"How can a testbed be made for BATMAN?"

Problem Analysis 2

Now that mesh networks and BATMAN has been introduced, the initial problem statement will be investigated and further analyzed to deduce a final problem statement.

2.1 Introduction to BATMAN

As mentioned in the introduction, BATMAN is a proactive routing protocol, meaning that route information is continuously kept up to date and used when a packet must be sent.

Instead of keeping entire routes in its table, each node, including the sender, only keeps the next neighbour to send to. To build these routes, BATMAN nodes need information about possible routing endpoints (also called originators), which is transferred using OGMs (Originator Messages).

Originally formulated in the 2008 IETF (Internet Engineering Task Force) draft *"Better Approach To Mobile Ad-hoc Networking (B.A.T.M.A.N.)"*, BATMAN version I through IV uses the OGMv1 scheme [2]. In OGMv1, each originator floods the network with OGM packets containing information about the originator and a sequence number used for routing. Other nodes will record the OGMs in a local *originator table*, including information about which sequence numbers have been received from which neighbour. Nodes can therefore find which neighbour has the best connection to an originator by statistical analysis of the originator table.

A sliding window with a size, proposed by the standard, of 128 OGMs is used to store which sequence numbers were received. OGMs outside this window are ignored. The originator table stores a sliding window for each neighbour, repeated for each originator.

The sliding window is a list of fixed length N, sliding forward with the current expected sequence number n. It therefore contains the status of the last N OGMs with sequence numbers $n-N+1$ to n. The status of OGMs in the sliding window is therefore a measure of link quality.

This protocol has several problems described below:

- *"Wireless [network] interfaces usually come with packet loss varying over time, therefore a higher protocol transmission rate is desirable to allow a fast reaction on flaky connections. Other interfaces of the same host might be connected to Ethernet LANs / VPNs which rarely exhibit packet loss or link state changes. Those would benefit from a lower protocol transmission rate to reduce overhead."* [6]

- *"It generally is more desirable to detect local link quality changes at a faster rate than propagating all these changes through the entire mesh (the far end of the mesh does not need to care about local link quality fluctuations)."* [6]

2.1.1 BATMAN V

BATMAN V tries to solve the issues stated above by splitting the responsibilities of the OGM packet into two; ELP (Echo Location Protocol) for neighbour discovery and OGMv2 for flooding network information. [7]

ELP packets are frequently broadcast from each node, and each receiver records this entry in the neighbour table. Additionally nodes frequently ping their neighbours for throughput and status measurement. [8]

With the ELP packages, each node has a throughput measure for each neighbour link, which is used as the main metric for routing. Each routing node updates forwarded OGM packets with the throughput of the receiving link, so that it always holds the throughput of the weakest link. [9]

When a node receives an OGM, it decides whether the sending neighbour should be selected as a router for the originator. The neighbour will be selected if the throughput is higher than that of the currently selected router, or if the sequence number is substantially higher than the last received OGM. The OGM is then forwarded only if the neighbour is the (now) selected router. [9]

With each hop a throughput penalty is added to the OGM packet, meaning shorter paths are preferred over long ones. [9]

2.1.2 BATMAN in OSI

The OSI (Open System Interconnection) model is a conceptual model for classifying network protocols. It defines 7 stacked layers, each only interfacing with its 2 immediate neighbour layers. The first 3 layers of the model are described below:

(L3) Network Routes packages to other networks enabling large networks.

(L2) Data-Link Manages how frames are sent over L1. IEEE (Institute of Electrical and Electronics Engineers) defines two sublayers LLC (Logical Link Control) and MAC (Media Access Control):

- LLC connects multiple L3 protocols to a single MAC [10].
- MAC handles framing, addressing, and error checking [10].

(L1) Physical Defines how unstructured bits are transmitted over a the physical medium such as a wire or wireless.

The BATMAN protocol has two different implementations; *batmand* and *batman-adv*. Since batmand is deprecated it is deemed out of scope. The newer implementation batman-adv runs at L2 of the OSI model [11].

Contrary to other L2 protocols batman-adv requires an existing L2 protocol such as ethernet or wifi. Batman-adv packets can therefore have two ethernet headers as can be seen in figure 2.1.

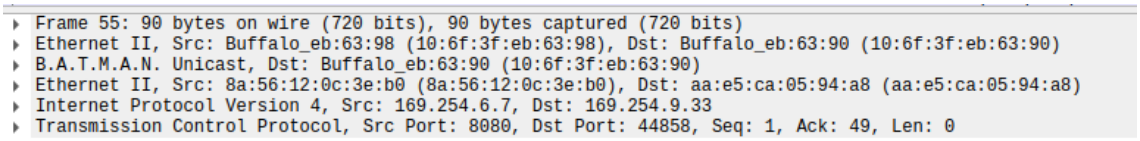


Figure 2.1: Headers added to a TCP packet sent through batman, captured through Wireshark.

2.1.3 Packet Size and Fragmentation

As stated in section 2.1.2 on the facing page, BATMAN adds multiple headers to each packet passing through the network, which is a problem if the underlying L2 protocol limits the frame sizes (MTU (Maximum Transmission Unit)).

If a higher MTU is not supported by e.g. drivers or hardware, BATMAN packets can be fragmented into smaller packets which fit within a limited MTU. However this comes at the cost of performance. [12]

2.2 Related Work

As stated in the introduction, in order to validate the potential of BATMAN, a testbed is created. Earlier projects have tried to validate the usefulness of BATMAN with testbeds. These testbeds often vary a lot in their setup, testing environment, and metrics measured, however examining them could still help narrowing the scope of which metrics that should be measured on this testbed.

An alternative to BATMAN is a routing protocol called OLSR (Optimized Link State Routing Protocol) and in [13, 14, 15, 16] comparisons of these two protocols have been made. In [15] it was found that OLSR performed better in their scenarios, which focused on the nodes being in movement, however it was also found that BATMAN showed better performance in terms of packet loss.

Packet loss

Packet loss is a focal point of ad-hoc networks due to the fact that re-transmission can be expensive when many hops are introduced. Whilst all 10 papers measured this metric, many of the results are not comparable due to the difference in the setups. The packet size is a factor in how much packet loss there is, in [17] it was found that in the 3rd hop an average of ~17.0% of the packets were lost with a packet size set to 73 byte, and ~71.2 % for the 4th hop. While using a packet size of 1500 byte it was found that packets were lost ~73.6 % and ~84.5 % of the time in the 3rd and 4th hop respectively. Both [18] and [14] managed to observe 0 % packet loss with 4 hops in a static environment. However, it is evident and intuitive that the packet loss percentage rises when movement is introduced and it was found that it impacts the system more negatively if the source node is moved rather than the destination node [13].

Generally the consensus seems to be that packet loss increases with the number of hops that is necessary to reach the destination node. Therefore, it might be of interest to investigate the possibility of decreasing the packet loss when multiple hops are used.

Throughput

This is a metric that looks at the rate of how fast messages are successfully delivered over a channel, which is often measured in bits per second. In [17] it was found that the packet size also greatly determines the throughput of BATMAN. With 73 byte packets it was managed to have a throughput of ~295 Kbit/s and with 1500 byte packets it was ~150 Kbit/s after 4 hops, while the throughput in the first hop was ~126 Mbit/s. According to [15] and [14] it was found that after the 3rd hop, throughput decreased more than 50%. Moreover, in [19] an experiment was conducted on a testbed which started with an average of 6.53 Mbit/s TCP throughput on the first hop, and when it got to the fourth hop the throughput dropped to 1.6 Mbit/s. According to [20] the throughput degradation of wireless ad-hoc networks follow the formula: $\Theta(W/\sqrt{n \cdot log(n)})$ for n randomly located nodes and W as the transmitting capability (bits/second). It could be interesting to see how close the throughput of a physical BATMAN testbed would follow the aforementioned throughput degradation model.

Delay

When data is transferred from one place to another it is bound to have a delay and oftentimes this delay increases when it is done wirelessly. [21] found that by placing 17 nodes on two different levels of a building, that the delay was around 10-30 ms. However, in the aforementioned paper it was not measured how well it handled the multi-hopping but instead just an average of all packets sent on the network. In [15] it was found that the average delay exceeded 2.5 seconds when data had to be moved over three or four hops. Furthermore, [17] found, as expected, that the packet size also had an impact on the delay i.e. a 1500 byte packet took more than double the time to transfer compared to a 73 byte packet.

Jitter

Jitter is the irregularities in periodicity in a periodic signal which is usually undesired in a system. A performance test between two different implementations of BATMAN, i.e. batman-adv and batmand, was conducted and jitter was found to be generally higher in the older implementation compared to batman-adv. In one of the scenarios all the nodes were static and the jitter was measured to ~60 ms for batman-adv and ~620 ms for batmand. [18]

2.2.1 Overview

In order to get an overview of how prominent the different metrics are in prior testbeds a chart was made which can be seen in 2.2. It should be noted that packet loss and packet delivery are counted as the same. The same should be noted for delay where latency and RTT (Round-Trip Time) are both counted as delay.

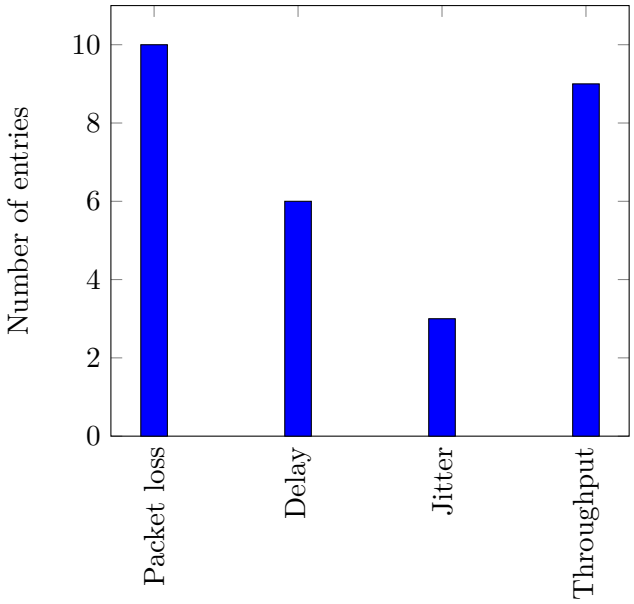


Figure 2.2: Distribution of metrics that were measured in 10 different BATMAN testbeds [16, 21, 13, 14, 15, 17, 18, 19, 22, 23].

In figure 2.2 it can be seen that packet loss was the most popular metric, followed by throughput which was measured in nine projects. Figure 2.2 could be used to decide which metrics that would be interesting to measure and which would not be. However, it is hard to say whether the most prominent metrics are the most important, easiest to measure, or nothing more than the most common.

2.2.2 Real Life Usage

BATMAN is in real use at Freifunk which is a volunteer organisation that operates free and open wireless community mesh networks.

Freifunk builds and distributes firmware based on the OpenWrt OS (Operating System), which is meant to be run on cheap home routers. Once connected to the mesh net, users can provide services (blogs, file sharing, news, etc. like any other online service) over IPv4 (Internet Protocol version 4) and/or IPv6 (Internet Protocol version 6). In the case of IPv4, only private RFC1918 (Request for Comments) addresses are available and assigned via central DHCP (Dynamic Host Configuration Protocol) servers. This means IPv4 services, by default, only are available to other Freifunk users and are not as decentralized as IPv6 services where public addresses are available and DCHP is not needed. [1]

There are Freifunk nodes operating in homes, rooftops, church towers, community centers, hackerspaces, cafes, venues, and even refugee camps so refugees can communicate with friends and family from their home country. Freifunk have also installed long haul radio links for redundancy and load distribution purposes. Freifunk has other meshnets (via VPNs), gateways to the internet and is present in the IX (Internet Exchange) in Hamburg and can be peered with through AS49009 (Autonomous System). [1]

Freifunk started in 2003 and some of the challenges they encounter are:

- **Frequent topology changes** mainly due to volunteers joining and leaving the network. [1]
- **Link quality changes** due to weather conditions, interference, and much of the network being run over Wi-Fi which is not nearly as reliable a wired connection. [1]

2.3 Test Scenarios

Open-mesh is the organization that hosts BATMAN infrastructure like code repositories, a wiki, and a website. They have defined test scenarios on their page that can showcase the routing of BATMAN under topology- and link quality changes in terms of the *Best Route*, *Convergence Speed*, and *Mobility* [24]. These scenarios could be interesting to carry out on a testbed to verify BATMAN behavior.

Best Route

The link quality between two nodes might not be the same for sending and receiving, as shown in figure 2.3. The protocol should detect the most optimal path through this network and sends data through that path.

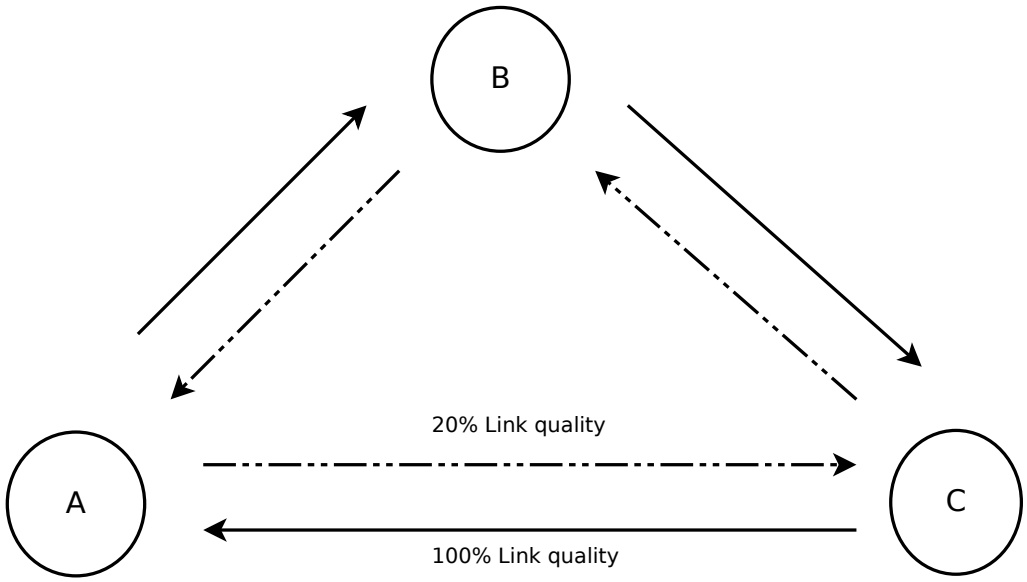


Figure 2.3: Simple link quality test configuration. [24]

In figure 2.3 it would not be the most optimal route for node A to send directly to node C, it is instead better to pass the packet through node B and then to node C, making it a multi-hop transmission. If the link quality between the two was 100% both ways, the path would simply be direct between them. The link quality in these figures is a representation of the best path compared with other less desirable paths. So the most optimal path is 100% and another path might be 20% as in the figure, meaning it is 20% as optimal as the best path, in terms of transfer speed etc.

Something that was not done in this example, that could be interesting, is specifying a route through the network separate to the optimal one, to compare the different paths'

travel time. This would, however, require more nodes than three, like the example in figure 2.3 on the preceding page.

To achieve this, a setup similar to the second test setup on open-mesh examining asymmetric paths could be used. This is shown in figure 2.4.

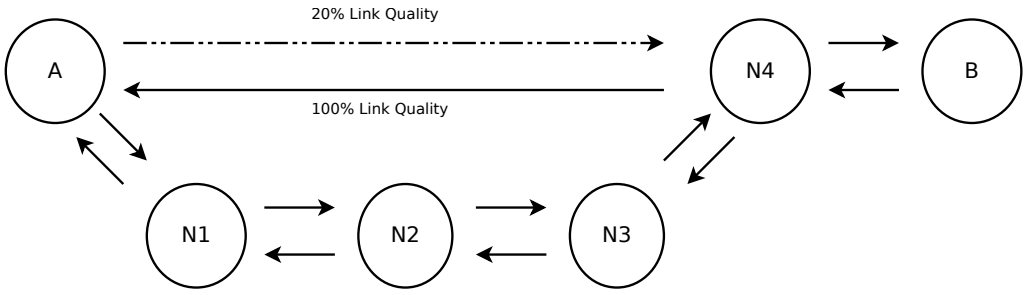


Figure 2.4: Expanded link quality test configuration. [24]

Here the optimal route for a packet from node A to node B is by going through the several 'N nodes'. Likewise for sending from B to A the path is still multi-hop with only one N node between B and A. But as previously mentioned, it might be interesting to compare the time the packet takes through undesirable routes with the most optimal one, for example forcing a packet to travel single-hop directly from A to B (if they can reach each other) and oppositely let a packet travel through the N nodes from B to A.

Having data on different paths for same size packets allows evidencing that the protocol is smart enough to travel through the most optimal paths.

Convergence Speed

Sometimes nodes disappear from the ad-hoc network, which might cause the rerouting to be revisited by the protocol for some nodes. It might also happen that the link between two nodes just disappears, and the protocol has to recalculate the best routing. Convergence speed tests show how fast the protocol is at altering the routing in these cases [24].

A simple way of doing this is shown in figure 2.5 on the next page. Here the solid line is the original routing, which is at some point broken caused by e.g. bad signal or the distance between the nodes are changed so that they are out of range of each other. The dotted line represents the newly calculated route for the packets.

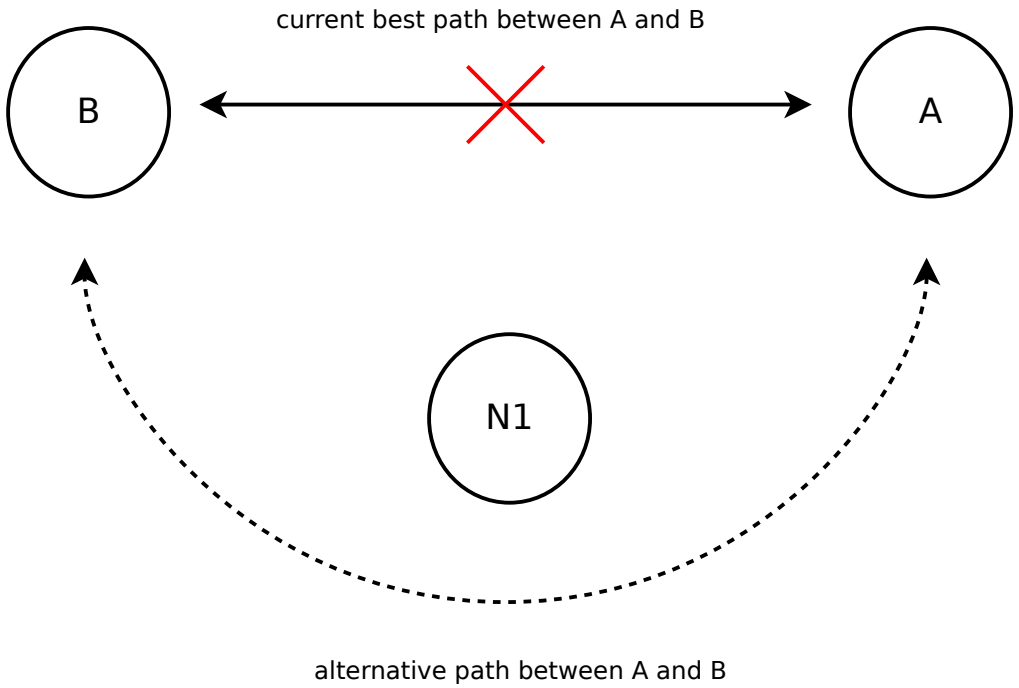


Figure 2.5: Simple broken link setup. [24]

However to test this in a more comprehensive manner, more nodes are necessary. Such a configuration is shown in figure 2.6.

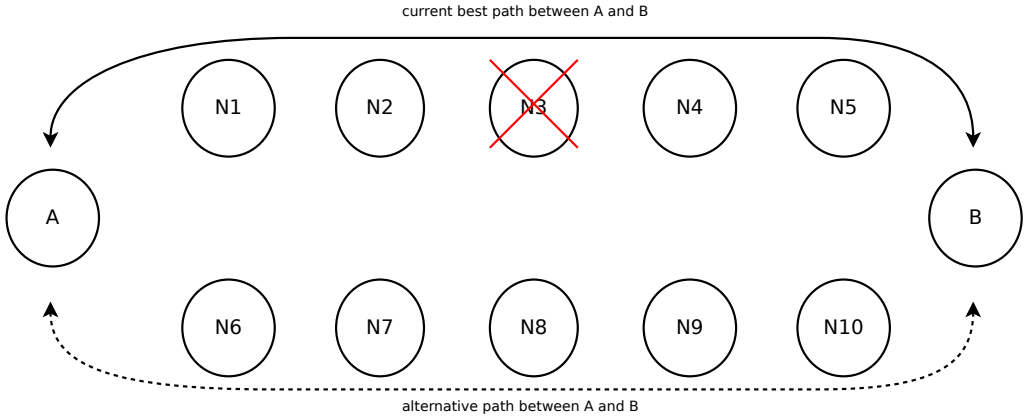


Figure 2.6: Expanded broken link setup. [24]

Figure 2.6 shows the routing from node A to node B which goes through some other nodes, shown as the solid line. Then one of the intermediate nodes is turned off and the route must be recalculated, represented by the dotted line. This is one of many ways of testing convergence speed, as there are many subsets of methods of testing the protocol in convergence speed. These all are similar and achieve different results, other example setups are shown in figure A.1 on page 77 and figure A.2 on page 77.

Mobility

The BATMAN protocol calculates the most optimal routes between nodes in frequent intervals. This is useful if nodes are moving, especially if some nodes come out of range

of each other and a new route is necessary for packets to reach their final destination. Open-mesh scenarios cover this, using figure 2.7 as a blueprint for the setup.

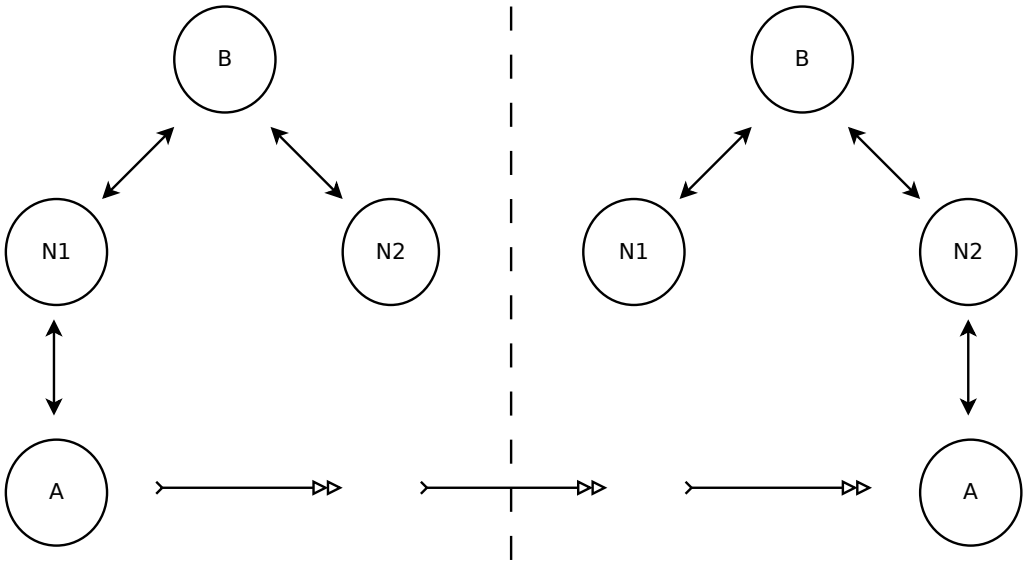


Figure 2.7: Moving node setup. [24]

Packets are moving from A to B by having intermediate nodes. Here the A node is moving and at some point A gets out of range of its neighbour node and in range of another, and a new route is calculated.

2.3.1 Adjusting Link Quality

Since link quality is important to BATMAN and these scenarios, it is necessary to simulate different link qualities, or nodes being out of range of each other. This can be difficult if space is limited since all nodes might be able to reach each other and this testbed is being made for a laboratory, so it is assumed that space will be limited. Therefore measures have to be taken to ensure that it is possible that link quality is adjustable and nodes being controllably in- and/or out of range of each other.

2.4 batadv-vis and alfred

batadv-vis is a program which can be used to export graphs of a BATMAN network in various formats. These graphs can give insight about a network that may be useful and as such *batadv-vis* is explored. An example of such graph is in figure 2.8 on the next page where the DOT export format, which is default, was used by piping it to the Graphviz [25] tool *fdp* producing a PDF (Portable Document Format) file: `# batadv-vis | fdp > fig.pdf`

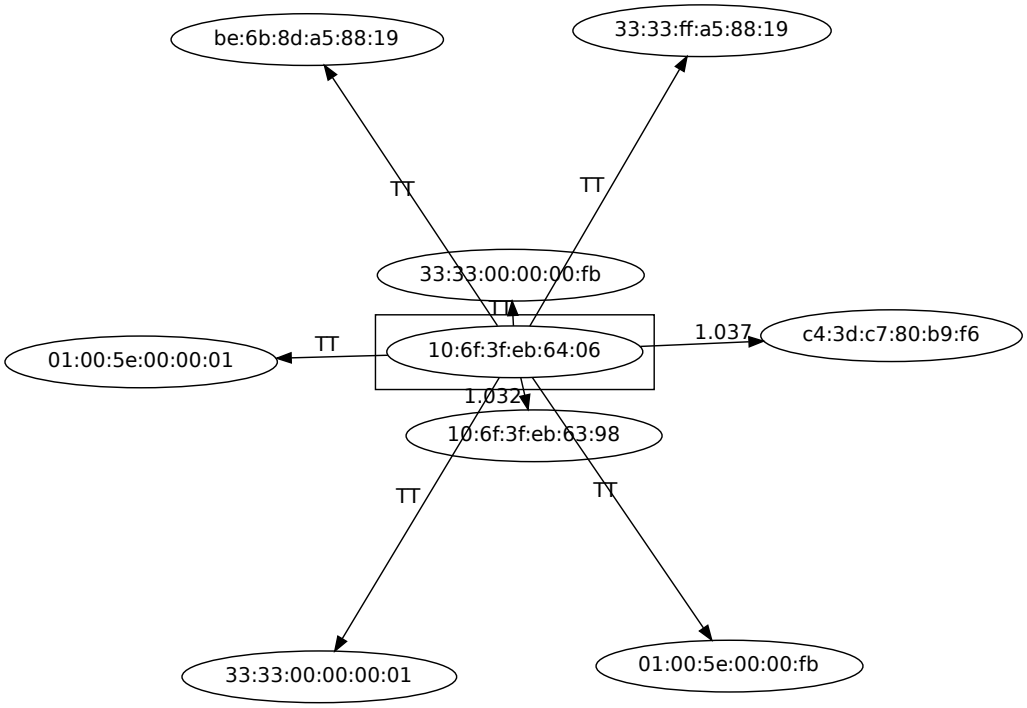


Figure 2.8: Example batadv-vis DOT output visualized with Graphviz while one node is running batadv-vis. DOT file source can be found in appendix B on page 79.

In figure 2.8, each ellipsis shaped node represents a network interface and edges represents links between two network interfaces. If the edge is labeled with a number where 1.0 is a perfect 100% TQ (Transmission Quality), 2.0 is 50%, 3.0 is $33.\bar{3}$% and so on. If the edge is labeled with "TT" (Translation Table), the *head node*[1] is a non-mesh client which can reach and be reached by the mesh network. Examples of such non-mesh clients are hosts being bridged into the mesh net and multicast MAC addresses (Media Access Control addresses) also show up as such. Rectangles group multiple BATMAN interfaces which belong to a single node. [26, 27, 28]

All nodes in figure 2.8 are described in table 2.1.

MAC Address	Description
`10:6f:3f:eb:64:06`	Network interface on the node which is running batadv-vis
`be:6b:8d:a5:88:19`	BATMAN interface on the node running batadv-vis
`c4:3d:c7:80:b9:f6`	Mesh net node, 1.037 TQ
`10:6f:3f:eb:63:98`	Mesh net node, 1.032 TQ
`01:00:5e:00:00:01`	IPv4 Multicast
`01:00:5e:00:00:fb`	IPv4 Multicast
`33:33:ff:97:7e:b8`	IPv6 Multicast
`33:33:00:00:00:fb`	IPv6 Multicast
`33:33:00:00:00:01`	IPv6 Multicast

Table 2.1: Description of nodes in figure 2.8.

[1] The node which is directed to.

Note that TQ is as seen from the node running batadv-vis. Additionally, of all non-mesh clients, only the non-mesh clients of the node which is running batadv-vis are visible. batadv-vis depends on another program, alfred (Almighty Lightweight Fact Remote Exchange Demo) which can make TQ and non-mesh nodes, as seen from other nodes, visible. alfred can distribute arbitrary information among other BATMAN nodes running alfred using multicast. It is a user space daemon that interfaces with batman-adv on the in-kernel netlink bus, which is available to user space through the netlink socket API (Application Programming Interface) [29]. batadv-vis is continuously run in server mode which will make alfred provide data from batman-adv in a unix-socket. While batadv-vis is run in server mode, another instance of batadv-vis can be run in client mode, which will print the graph. [28, 30]

The data flow (reading and writing) between sockets, buses, and programs is illustrated in figure 2.9.

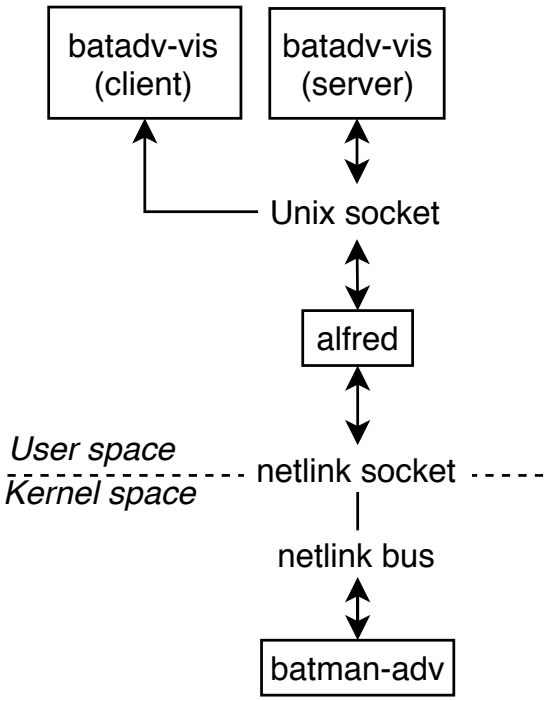


Figure 2.9: Data flow diagram of batman-adv, alfred, and batadv-vis.

alfred has two modes of operation, primary and secondary:

- Primary nodes can receive and send arbitrary information (from itself or secondary nodes to fellow primary nodes). [30]
- Secondary nodes can send arbitrary information to primary alfred nodes.

Running alfred and batadv-vis on two nodes in the same example setup used for figure 2.8 on the facing page and running batadv-vis (client) on a primary alfred node yields figure 2.10 on the next page.

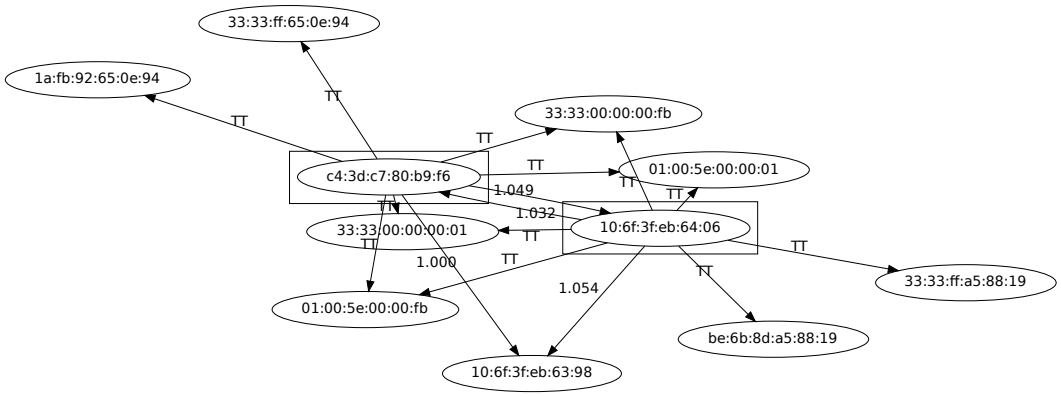


Figure 2.10: Example batadv-vis DOT output visualized with Graphviz while two nodes are running alfred. DOT file source can be found in appendix C on page 81.

In figure 2.10 it can be seen that the "TT" edges of the two nodes running alfred are now visible alongside TQ as seen by both nodes.

In conclusion, batadv-vis does not provide any of the metrics that related works have measured (see section 2.2 on page 7), but it could be interesting to include in the testbed. For instance, it could be used to verify that a test scenario (see section 2.3 on page 10) was performed properly (with the expected links, transmission quality, etc.).

2.5 Desired Capabilities

Some capabilities were discussed with Sebastian Bro Damsgaard, a research assistant at AAU that will use the testbed. The desired capabilities that were agreed upon are assisted deployment (including deployment of various applications, e.g. voice chat) and replicable results.

Having assisted deployment of the testbed allows streamlining of the setup of tests, making them more 'fool proof', and minimizing the work a scientist will have to do to set up the testbed. Examples of assisted deployments are for instance a way to reconfigure multiple nodes at once.

This assisted deployment system can also make a test configuration be reproducible or replicable. A distinction between reproducible and replicable is necessary, as the two have different implications. If results from a test are absolutely the same between two identical tests, then the testbed's results are reproducible. This would be ideal. However, since this is a real-world testbed the results can not be perfectly reproducible, but replicable. By replicable results it is meant that variability in the results of otherwise identical experiments should be minimized. By having replicable results, tests can be rerun (by oneself or by others) to further validate the results. Unfortunately there are *external factors* which practically cannot be controlled, such as interference (e.g. from nearby phones, laptops, access points, etc.) or hardware faults. On the contrary, *internal factors* within practical control of the testbed should be controlled.

Aside from assisted deployment and replicable results from tests, there should also be raw access to the data that leads to results. That is, regardless of what data is being gathered, *all* of the raw data from a test should be available to scientists using this testbed. Allowing for collection of local state like neighbor tables on each BATMAN node could also be useful for scientists using this testbed, for instance to determine the reason for unexpected behavoir. This can be done using existing tools like *batctl*.

Finally, movement of nodes during testing should also be allowed without the testbed breaking, but any assistance in moving the nodes is deemed out of scope for this project.

2.6 Final Problem Statement

In summary, BATMAN is a proactive routing protocol for ad-hoc meshnets that operates between layer 2 and 3 in the OSI model. Others have measured packet loss, delay, jitter, and throughput with various testbeds. Yet, there are many scenarios where BATMAN remains inexhaustively tested. It would be beneficial if a testbed could be constructed that allows for carrying out these scenarios in a replicable fashion while measuring useful metrics and integrating with existing BATMAN tools. As such, the following final problem statement is formulated.

"How can a testbed be made for measuring packet loss, delay, jitter, and throughput of the BATMAN under open-mesh's- and potentially additional routing scenarios at Aalborg University?"

Requirement Specification 3

Ad-hoc networking is an alternative to the traditional internet structure. An implementation of an internet protocol that allows this is batman-adv, which is an implementation of the BATMAN protocol. This is however a relatively new and unexplored technology, and therefore it is interesting to test the limits of this new internet structure, by creating a streamlined testbed. Reports have been created from tests of ad-hoc networks, using not only BATMAN but other protocols to route through ad-hoc networks. Out of the metrics these reports have measured it was found in the problem analysis that; packet loss, delay, jitter, and throughput were interesting metrics to measure in this testbed, as well as additional metrics from batctl, such as neighbors and originators.

To streamline the testing of BATMAN, a few test scenarios have been identified, that explores routing in terms of best route, convergence speed, and mobility.

From the problem analysis the following requirements are specified. The order in which they are presented have no meaning in regards to their ranking of importance, as they are all valued equally, for the testbed to function according to the problem analysis.

1) The testbed should measure packet loss, delay, jitter, and throughput.
   This makes it possible to analyse the protocol and implementation.

2) The testbed should store the measured metrics persistently.
   This makes it possible to analyze metrics after a test is done.

3) The testbed should have adjustable link quality between nodes.
   This makes it possible to deploy the testbed in environments with different sizes.

4) The testbed should have assisted deployment.
   This will make the setup easier.

5) All scenarios in section 2.3 on page 10 should be able to be carried out on the testbed.
   This ensures that the testbed can test BATMAN's ability to find the best route, convergence speed, and mobility

6) The results of carrying out scenarios on the testbed should be replicable.
   This allows verification of results.

7) The testbed should collect the local BATMAN state of nodes using existing tools.
   This gives an insight on how BATMAN takes routing decisions on each node.

Design 4

This chapter addresses thoughts on the infrastructure of deploying the system, how the software modules should interact with each other. Lastly, the chapter will describe other challenges that will need to be taken into consideration.

When developing a system that can conform with requirements such as the ones listed in chapter 3 on page 19, it was deemed necessary to sketch the system before the development began.

4.1 Network

For BATMAN to be tested, some nodes should be interconnected via BATMAN. As illustrated in figure 4.1, this network will be referred to as *batnet*.

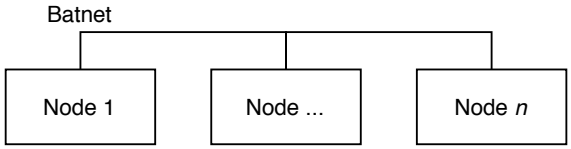


Figure 4.1: Nodes connected via BATMAN.

It is desired that the testbed should have assisted deployment which provisions the nodes with certain software versions. An instance of such software is a *Tracker* which is to be run on each node to collect BATMAN metrics. The communications channel to transfer metrics is named the secondary channel, even though it may be the batnet itself or a local cache on each node which is gathered after each experiment. The pros and cons of each considered secondary channel type for transferring metrics are shown in table 4.1.

Channel type	Pros	Cons
Cache on node	Minimal interference	Can not transfer metrics live
Wired	Can transfer metrics live	Requires a wire for each node
Wireless	Can transfer metrics live	Might interfere with batnet if they are on the same frequency
Batnet	Can transfer metrics live, Does not require additional set up	Interferes with batnet

Table 4.1: Pros and cons of secondary channel types.

It was decided that a wireless channel, which is not the batnet itself, should be prioritized as it is deemed to strike a balance between interference and practicality. Practicality in

the sense that running many wires or provisioning the nodes over batnet where reliability is unknown or setting up a third channel is impractical.

The use of a secondary channel for metric collection and deployment is illustrated in figure 4.2.

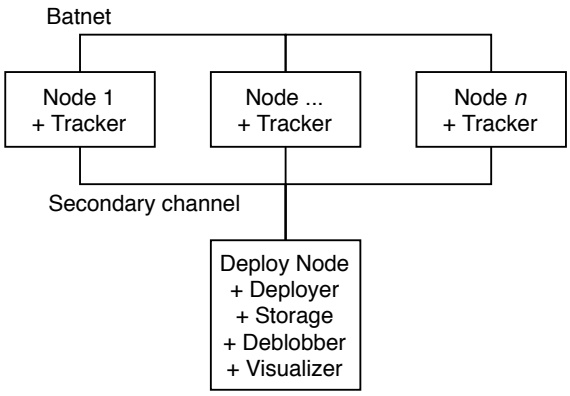


Figure 4.2: Batnet and secondary channel.

4.2 Deploy Node

Figure 4.2 also shows that metrics are gathered and stored centrally on the *Deploy Node*. This is done since there is no immediate need for distributed storage and storage being centralized makes ACID (Atomic, Durable, Isolated, Consistent) transactions easier to achieve. However, metrics are not necessarily sent in a format that can be stored directly, the data may be sent as a "blob" (a binary format) instead. Therefore a *deblobber* "deblobs" the data before sending it to storage. The deblobber can be run on each node or only the deploy node. This is left as an implementation detail though it is shown here as running on the deploy node. Once the data is stored, the visualizer program can illustrate the measured metrics.

In figure 4.3 on the next page it is shown that the deploy node also acts a gateway, providing an internet connection to the nodes. This is done to give the nodes access to online software repositories which may be used during deployment or debugging.

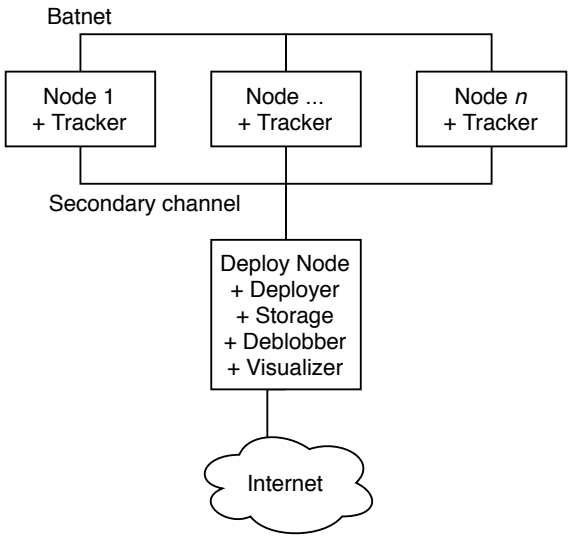


Figure 4.3: Batnet and secondary channel with internet acccess.

Having the deploy node configuring nodes over a connection does propose a problem though: How does the initial connection get configured? This is a 'chicken-and-egg' problem since the deploy node is supposed to roll out such configurations, but it requires a connection to do so.

Two solutions to this 'chicken-and-egg' problem were identified; manual configuration of the initial connection or modifying the disk image that nodes boot on to include configuration of the initial connection. The latter was chosen as disk images have to be distributed anyway. These modified disk images are copies of the same image, called a *golden image*.

4.3 Golden Image

Using a golden image is beneficial since it ensures that software versions, configuration files, etc. are identical across nodes.

Care should be taken not to let software versions, configurations files, etc. deviate from the golden image as that could harm replicability of results (e.g. by using a newer kernel). Letting nodes deviate also reduces the usefulness of the golden image as a test target of software that should be run on there, such as the tracker.

4.4 Tracker

Figure 4.4 on the following page shows the overall structure of the tracker software, and shows how different modules will be chained together. The chain will be event-based meaning each *Input Module* will push new metrics when they arrive. The tracker is event-based because packets arrive sporadically and they should have unique timestamps (as opposed to e.g. stamping multiple packets with the same timestamp).

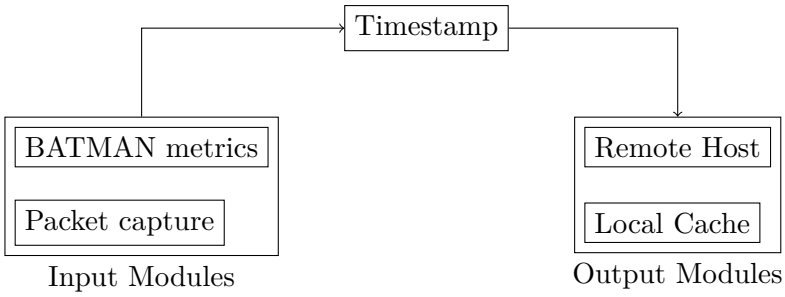


Figure 4.4: The structure of the tracker, with input- and output modules.

When a new metric has been emitted by the Input Module, it will travel along the chain until it reaches an Output Module. This chain structure was chosen to ease the process of adding Input Modules or extra links.

Before metrics reach the *Output Module* it passes through the *Timestamp Module* which stamps each metric with a timestamp and a node id. The timestamp must be precise and synchronized with other nodes to allow time comparisons between metrics from different nodes.

4.5 Clock Synchronization

Since a time stamper is needed, some sort of clock synchronization is necessary, so the nodes can agree on what time it is. However, it is not sufficient to just do this once, since the clock in a computer can drift. This drift can be caused by inaccuracies in the local clock source, and eventually not be running at the exact same time as a reference clock, so one will eventually be ahead or behind of the other. An example of this can be seen in figure 4.5, where three nodes have different perceived times.

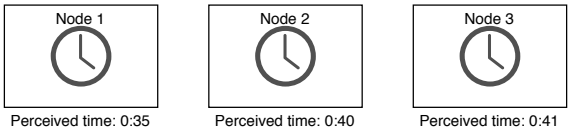
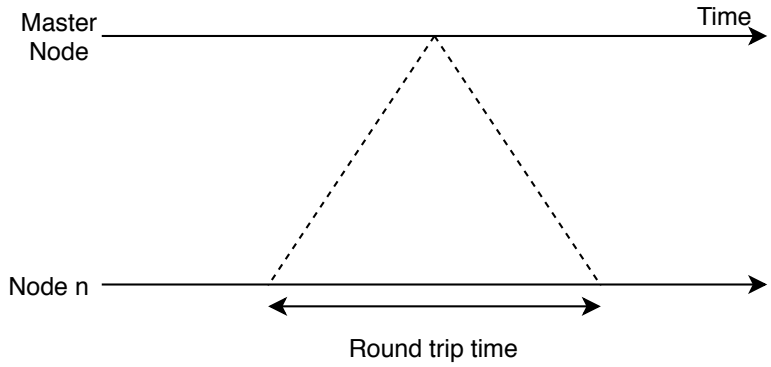


Figure 4.5: Desynchronised nodes having their own perception of time.

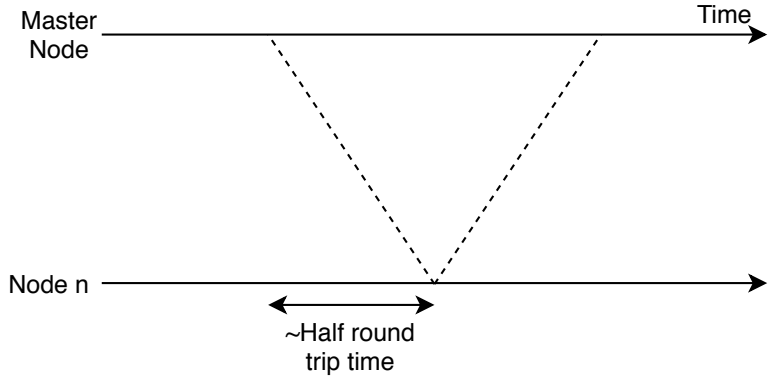
One potential solution to this problem is to have the nodes ask a NTP (Network Time Protocol) server, specifically designed for clock synchronisation. There are different implementations of this interaction; Windows Time, Chrony, SNTP, etc. To avoid having to synchronizing every node, a single node can work as a master node for clock synchronisation for all other nodes on the network. The master node can get its time either from an upstream provider or its own local clock. Benefits of using the master node's local clock are that it does not require an internet connection, but a drawback is that it can drift and make timestamps incomparable to outside devices like phones or wristwatch. The rest of the nodes synchronise their clock with that master node.

These clock synchronisation services work in two different ways. One is where the a node asks the server what the time is and gets a response and the other method, the server tells

the nodes what time it is without being asked. These interactions are shown in figure 4.6a and figure 4.6b



(a) Nodes ask the master node for clock synchronisation.



(b) Master node telling the nodes what time it is, and nodes replying with an acknowledgement.

Figure 4.6: Methods of achieving clock synchronisation.

Both of these methods have their own advantages and disadvantages. The schema on figure 4.6a has one major disadvantage. Namely, the time it takes for the full round trip, since a node has to ask the master node, and wait for a response. This can, if the connection is experiencing high variance in latency, known as jitter, cause the nodes on the network to still have different perceptions of what time it is. Contrarily this time difference would be smaller between nodes if the same situation arose for method two on figure 4.6b, where the master node just tells the nodes what time it is. This effectively means that the time it takes to synchronise clock on the network is half the round trip time of the first method. Here the master tells the time to the node, which that node responds to with an acknowledgement back to the master node. The method used by NTP is method one, as the computer knows best when it wants to have its clock synchronised, for example on boot up [31].

4.5.1 Conflicting Time stamps

Once the synchronisation of the clocks on the nodes has been completed, another problem can arise. Namely, what should happen if multiple measurements happen within a single clock tick. Table 4.2 on the following page shows such a situation.

Packet	Timestamp
A	20
B	21
C	21
D	22

Table 4.2: Example data with time stamps.

In this example the interval between clock ticks is slower than the rate at which packets are timestamped. This can make it hard to distinguish packets across nodes, and impossible to know with certainty which packet was stamped first from the timestamp alone. It also means certain metrics can not be calculated over sufficiently short time spans. For instance, bytes transferred from 20 to 20.8 is unknown since this example timestamps packets with an integer. It is assumed that clock ticks, in practice, are fast enough to allow for common metrics to be calculated and therefore it is not addressed. To address that packets can not be distinguished across nodes in case of conflicts, it was decided that the tracker should include a hash of packet contents which would make a sample timestamp conflict look as shown in table 4.3.

Packet	Timestamp	Hash
A	20	123423456
B	21	789678967
C	21	879045677
D	22	674567454

Table 4.3: Example data with time stamps and packet content hashes.

In table 4.3 packets can be distinguished across nodes even in they have conflicting timestamps due to the hash of packet contents.

4.6 Link Quality

As mentioned in section 2.3.1 on page 13 link quality and distance/space are important. Introducing varying distances between nodes in testing sessions might yield different results, which are also relevant when testing the limitations of any network. Using the same arrangement of nodes but with varying distances is one simple way of achieving this. The ways of adjusting link quality that are explored, are:

- Physical distance between nodes;
- software;
- shielding.

Physical Distance Between Nodes

The simplest way of testing the network in the scope of distance is by physically moving nodes away from each other. However, a large area of testing may be needed, which means

variance in the environment (e.g. different levels of interference) can be harder to account for. Consequently, high range nodes can make it harder to test BATMAN.

Software

Another method for adjusting the link quality between the nodes is with a software approach. This can be done by lowering the transmit power on the antennas, which would mean that the nodes would need to be closer in order to communicate, potentially allowing for a scaled down version of the testbed.

Alternatively, consequences of reduced link quality like increased packet loss and reduced throughput can be imitated in software, for instance via Linux Traffic Control [32].

Shielding

Similarly to reducing transmit power, change in link quality can be achieved in another way; by shielding the nodes. [33] achieved a simulated distance between nodes by inserting each of them into cardboard boxes wrapped in aluminum foil. This would allow nodes to communicate with each other in a semi-controlled manner, either by creating thicker layers of foil or moving boxes slightly further away from each other. However, it is difficult to translate the thickness of the shield to link quality reduction in the real world, and therefore should be used cautiously.

Regardless of how distance between nodes is implemented in the testbed, all of Openmesh's test setups are compatible with all of the previously mentioned solutions.

4.7 Summary

A deploy node is used to deploy a golden image of the testbed software across all nodes, which includes a working BATMAN setup, tracking software, as well as other implementation specific software.

The tracking software has input modules that are designed such that additional modules can be added to the testbed if needed. By default, the tracking software has input modules for packet capturing and batctl metrics. These metrics are time stamped at time of measurement and sent to an output module responsible for sending the data to the data store.

To ensure that timestamps are synchronised all nodes synchronize their clocks from the deploy node.

Lastly, three methods of reducing link quality have been identified. Of these, reduction of transmit power from software and physical distance will be used. Cardboard boxes are excluded since it is speculated that effects of aluminum foil are not identical to real world effects like distance. Linux Traffic Control is excluded since the configuration which best imitate the real world is unknown.

Implementation 5

The system was built to conform with the system design described in chapter 4 on page 21.

In order to make the system flexible for gathering various metrics, it was designed to be modular and thereby ease incorperation of additional metrics as seen in figure 4.4 on page 24.

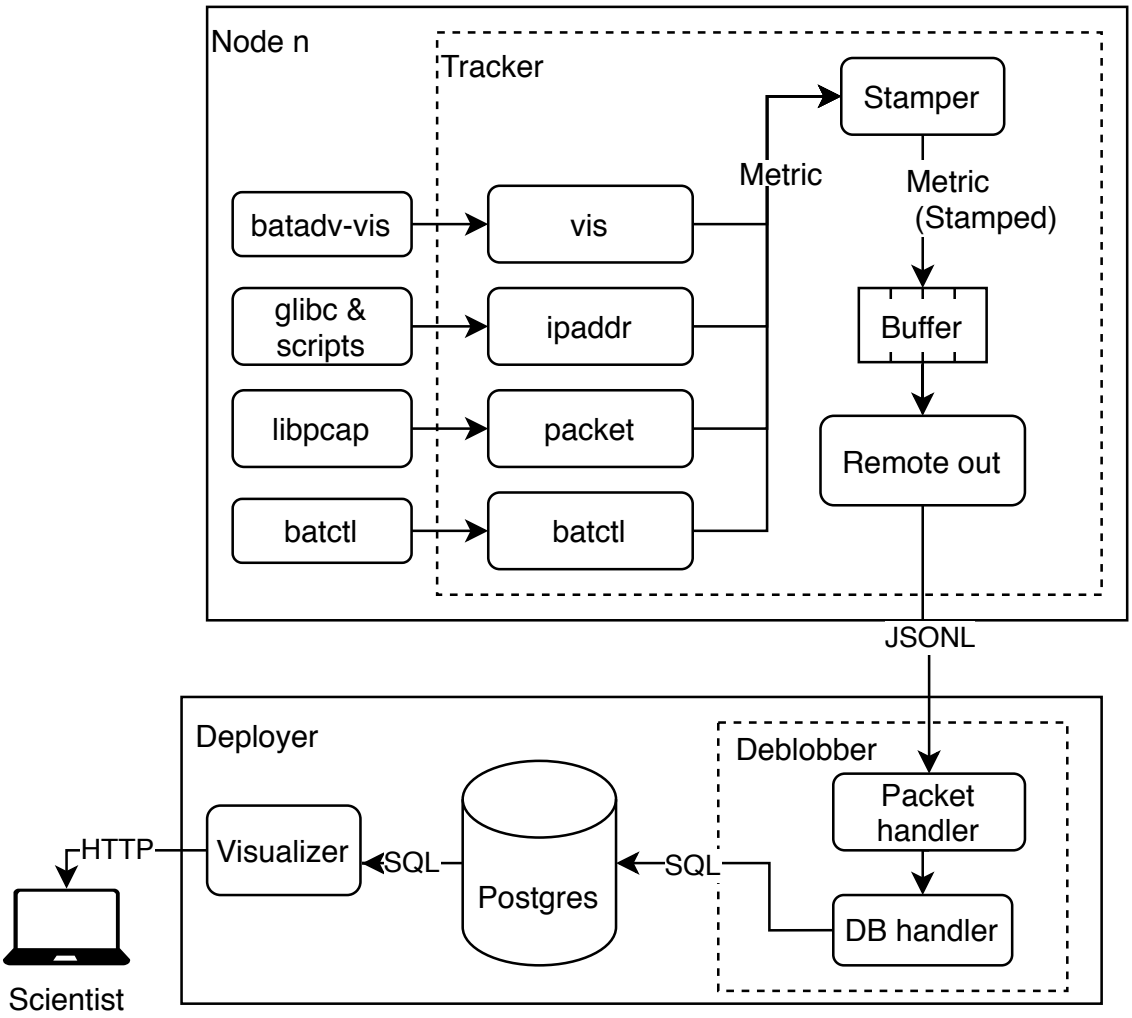


Figure 5.1: System overview

In figure 5.1 it can be seen that that the input modules i.e. `vis`, `ipaddr`, `packet`, `batctl` all send their data as a `Metric` object. In the `Stamper` module the Metric object gets assigned two new attributes: a timestamp and an node identifier. The stamped

Metric object then gets passed to a buffer module which is in place to secure that no other input module is blocked in case that the `Remote out` is unable to send. The Remote out module sends the data formatted as JSONL (JavaScript Object Notation Lines), to a remote host. The `Deblobber` module holds a socket open and continually listens to it. Afterwards, when data is received it is then processed and inserted into the database by the deblobber. Lastly, data can then be queried from the database and visualized with the `visualizer` tool.

As a final remark, it was chosen that the Deblobber should run on the deploy node. This allows for using a high-level language and dependencies without worrying much about computational resources or managing said dependencies across nodes.

5.1 Hardware

For this project several Raspberry Pi -2Bs, -3s, -3Bs, and -4s have been provided for developing and testing the testbed. The Raspberry Pi 2Bs use an ARMv7 architecture [34] while the -3s and 4s use ARMv8 [35] which is largely backwards compatible with ARMv7 [36].

It is not possible to exceed the 1500 MTU on the Raspberry Pis built-in Wi-Fi module, so a Wi-Fi dongle with a changeable MTU was decided to be the solution to this problem. The Wi-Fi dongles used for this are the Buffalo AirStation N150 Wireless USB Adapter (WLI-UC-GNM), referred to as Buffalo(s), which have a changeable MTU size.

The laptop used as the deploy node is an x86 Lenovo Thinkpad T520 borrowed for this project.

5.2 Deployer

The deploy node acts as a master node, which is responsible for pushing out golden images containing an OS (Operating System) with required configurations, along with a working BATMAN setup. Furthermore, the deployer also puts the tracker on the nodes. This deployment is achieved with Ansible which eases reconfigurating nodes and the tracker as well as the deployer itself. Using Ansible allows configuring of all nodes at the same time, so that they are all identical.

5.2.1 Golden Image

Since the golden image does not have to be modified often, it was decided to modify the prebuilt Raspberry Pi OS (based on Debian) [37] image. This is faster to learn than learning a more comprehensive build system. The Raspberry Pi OS image is `2020-08-20-raspios-buster-armhf-lite.zip` with a SHA256 (Secure Hash Algorithm) checksum of
`4522df4a29f9aac4b0166fbfee9f599dab55a997c855702bfe35329c13334668`.

In the Raspberry Pi OS image there are two partitions, one which is supposed to be mounted on `/` (named root partition) and another which is supposed to be mounted on

`/boot` (named boot partition). To modify the image, each partition is mounted by their offset which are found using *fdisk* as shown in snippet 1.

```
$ fdisk -l image.img
Disk image.img: 1.72 GiB, 1845493760 bytes, 3604480 sectors
Units: sectors of 1 * 512 = 512 bytes
Sector size (logical/physical): 512 bytes / 512 bytes
I/O size (minimum/optimal): 512 bytes / 512 bytes
Disklabel type: dos
Disk identifier: 0x907af7d0
Device     Boot  Start     End Sectors  Size Id Type
image.img1        8192  532479  524288  256M  c W95 FAT32 (LBA)
image.img2      532480 3604479 3072000  1.5G 83 Linux
```

Snippet 1: fdisk usage to view partition offset and sector size.

In snippet 1 it can be seen that the root partition (as identified by the Linux file system) starts at an offset of 532480 sectors. This means the offset in bytes is $532480 \cdot 512 = 272629760$ bytes (since the sector size is 512 bytes) which can be used by *mount*, for instance `mount -verbose -options offset=272629760 -types ext4 image.img /tmp/mnt`. Once mounted, the partition can be trivially modified as long as its size is not exceeded (there are ~275M available on root partition of the image that was used).

The modification performed to the root partition is a configuration of the initial connection to the deploy node as mentioned in chapter 4 on page 21, which is similar to the configuration explored in section 5.2.2 on the following page.

Two modifications are performed to the boot partition:

- A `/boot/ssh` file is created which turns on `sshd` on the nodes, which is needed for further configuration.
- a `/boot/wpa_supplicant.conf` file that sets the country code is created, which is needed for the radio to turn on.

Note that these files are "consumed" by default services running on Raspberry Pi OS, meaning they may not be present in `/boot` once the Raspberry Pi is booted on the image.

This modification process was semi automated by writing a few simple scripts. For instance, the creation of the `/boot/wpa_supplicant.conf` file is shown in snippet 2 on the next page (assumes the boot partition has been mounted in $MNT, which is done previously).

```
if [ "$TYPE" = "boot" ]; then
    # Create ssh file
    touch "$MNT/ssh"


    # Set wifi country code
    # A service on the rpi image will copy this and disable rfkill
    cat > "$MNT/wpa_supplicant.conf" << EOF
ctrl_interface=DIR=/var/run/wpa_supplicant GROUP=netdev
update_config=1
country=$COUNTRY_CODE
EOF
fi
```

Snippet 2: Creation of the `/boot/wpa_supplicant.conf` file in the boot partition.

To distribute the golden image, network booting was considered. However it is seemingly not more convenient than flashing memory cards as only wired network boot is supported by the Raspberry Pis [38].

5.2.2 Configuring Nodes

As mentioned in chapter 4 on page 21, an assisted deployment system should install needed software and keep versions consistent across nodes. Furthermore, it is also a requirement that this system can manage kernel modules and networking since batman-adv is a kernel module and needs according network configuration. Multiple solutions exists that can solve this problem, however Ansible was chosen, which is a tool that can help automating deployment of applications and provisioning of software versions. The reason for choosing Ansible over other similar tools is due to previous experience.

For instance, installing BATMAN and adding the batman-adv kernel module is done with the YAML (YAML Ain't Markup Language) in snippet 3.

```
- name: Install batctl and avahi
  apt:
    pkg:
      - batctl=2019.0-1
      - avahi-autoipd=0.7-4+b1
    state: present

- name: Add batman-adv kernel module
  modprobe:
    name: batman-adv
    state: present
```

Snippet 3: Installing BATMAN and adding the batman-adv kernel module.

It can be noted that *avahi-autoipd* is also installed. avahi-autoipd implements dynamic

configuration of IPv4LL (Link Local) addresses, as specified in RFC3927, which means nodes in the testbed automatically will configure an IPv4 address. As mentioned in section 2.2.2 on page 9, Freifunk use centralized DHCP servers and not IPv4LL. Presumably because IPv4LL addresses are limited to the network scope `169.254.0.0/16`, which contains $2^{16} = 65536$ addresses, meaning that address collisions would be a problem as the number of nodes in the mesh network increase. However, as there are not nearly that many nodes in this testbed, address collisions are not a problem. As seen in snippet 4 (of the network configuration on mesh nodes) avahi-autoipd is started every time the `bat0` interface is brought up.

```
1 iface bat0 inet6 auto
2         pre-up batctl-wrapper.sh
3 iface bat0 inet manual
4         up avahi-autoipd -D --force-bind bat0
5         down avahi-autoipd -k bat0
```

Snippet 4: Starting of avahi-autoipd and calling of batctl-wrapper.sh in the `/etc/network/interfaces`

With regards to IPv6, Rasberry Pi OS seemingly enables IPv6LL addresses when an interface is set to "auto". As such, each node has both an IPv4 and IPv6 address. In snippet 4 it can also be noted that before being brought up, batctl-wrapper.sh gets called. batctl-wrapper.sh is a script that adds a specified interface to BATMAN via `batctl if add`. The reason for this wrapper script is that as mentioned in chapter 4 on page 21, this testbed prioritizes a wireless channel which is not the batnet itself. In practice, this second channel is implemented using the Wi-Fi module that is built-in the Raspberry Pis.

Therefore there needs to be a way to distinguish between the built-in Wi-Fi module and the USB adapters. This could be done via interface names but Debian bug #101728 [39] means that the wlan0/wlan1 naming is not consistent across reboots. That is, wlan0 may correspond to the built-in Wi-Fi module before a reboot but correspond to a USB adapter after a reboot. A solution to this is using *systemd*'s predictable interface names. However, it seems systemd changes the naming scheme once in a while [40, 41] which would mean scripts around the naming scheme have to be modified whenever the scheme is changed. While the naming scheme would only change on major upgrades of Raspberry Pi OS since it is a breaking change, the scripts would seemingly have to be modified more than if the wlan0/wlan1 naming is kept.

Scripts are used since it was deemed beneficial to be able to distinguish between a built-in Wi-Fi module and a USB adapter by an arbitrary number of bytes of the MAC address, usually three to represent the manufacturer, which is not directly offered by systemd's naming scheme [41]. This is beneficial since the scientist does not need to check the MAC address of all wireless adapters, but only one for each manufacturer. As can be seen in snippet 5 on the following page, these MAC address prefixes are specified as a YAML list in Ansible variables.

```yaml
nodes:
  vars:
    batnet:
      macs:  # beware that networkmanager uses MAC addr randomization
        - 4c:e6:76  # buffalo a
        - 10:6f:3f  # buffalo b
        - c4:3d:c7  # netgear
```

Snippet 5: YAML list of MAC address prefixes as Ansible variables.

It seemed obvious to have the MAC addresses as Ansible variables since the file that contains these variables has to be edited anyway to specify nodes' IP addresses. As shown in snippet 7 and snippet 5, the MAC addresses get joined to a space separated string by Ansible templating. This is because the content of the double curly braces get evaluated by Ansible templating before placing the script on a node. Templating is executed using snippet 6 in an Ansible playbook. An Ansible playbook is essentially a collection of YAML files with tasks that the Ansible-playbook command line tool executes on specified IP addresses over SSH (Secure Shell).

```yaml
- name: Template and copy script templates to remote
  template:
    src: "{{ item }}"
    dest: "/usr/bin/{{ item | basename | regex_replace('\\.j2$', '') }}"
    mode: '755'
  with_fileglob:
    - "*.sh.j2"
  notify:
    - Restart networking
```

Snippet 6: Templating and placement of files with the .sh.j2 file extension (j2 is short for *jinja2*, the templating language used by Ansible).

In snippet 6 it can also be seen that there is a notify key with a "Restart networking" value. This means that if Ansible templates and places scripts, which it only does if the content of relevant files have changed, the "Restart networking" handler will be notified. A handler is like a task, except that it is only executed when notified and only executed once, even if it is notified multiple times.

```
[ -z "$BATNET_MACS" ] && BATNET_MACS="{{ batnet.macs | join(' ') }}"
[ -z "$IP_LINKS" ] && IP_LINKS=$(ip -brief link)


batnet_interface
```

Snippet 7: MAC address prefixes joined by templating into a script.

In snippet 7, note that the `batnet_interface` function is called. That function uses the

global variables `BATNET_MACS` and `IP_LINKS`, as shown in snippet 8.

```
batnet_interface() {
        local ii=1  # sed counts lines from 1
        local found=0
        for fullmac in $(echo "$IP_LINKS" | awk '{print $3}'); do
                for mac in $BATNET_MACS; do
                        # prefix match, allows for e.g. first 3 bytes of
                        ↪ MAC
                        if [[ "$fullmac" == $mac* ]]; then
                found=1
                # Will break out of two levels
                                break 2
                        fi
                done
                ii=$((ii + 1))
        done

        # if all lines were iterated, but no match, return 1
        [ "$found" -eq "0" ] && return 1

        # the interface from the line where mac address matched
        interface=$(echo "$IP_LINKS" | sed --quiet ${ii}p | awk
        ↪ '{print $1}')

        # only print if an interface was found
        [ -n "$interface" ] && echo "$interface" || return 1
}
```

Snippet 8: batnet_interface function.

The batnet_interface function loops over all lines in IP_LINKS and each line contains an interface name and a corresponding MAC address. When it finds the given MAC address prefix that matches a full MAC address it uses the `ii` (iterator) variable to determine which line of IP_LINKS the match was and prints the interface name of that line. The batnet_interface gets used by a mapping in `/etc/networking/interfaces` as shown in snippet 9.

```
mapping wlan*
        # Interface name is passed as argument, script prints "batnet" or
        ↪ "tracknet"
        script batnet-or-tracknet-from-if.sh
```

Snippet 9: Mapping of wlan interfaces to a configuration.

The mapping in snippet 9 matches all interfaces with the "wlan" prefix and passes each interface name as an argument to a script named `batnet-or-tracknet-from-if.sh`. This script uses the batnet_interface function in snippet 8 to determine if the argument is

supposed to be a batnet or tracknet, which is also called Secondary Channel in section 4.1 on page 21, and prints it. This print is used to determine whether a batnet or tracknet interface configuration should be applied which are shown in snippet 10 and snippet 11 respectively.

```
iface batnet inet manual
        mtu 1532
        wireless-channel 2
        wireless-essid BATCAVE
        wireless-mode ad-hoc
```

Snippet 10: Configuration of a batnet interface.

```
iface tracknet inet dhcp
        wpa-ssid Alfred PennyWorth
        wpa-psk azcomtek550phat$tacks
```

Snippet 11: Configuration of a tracknet interface.

In summary, the deploy node runs Ansible which places an `/etc/network/interfaces` file on each node. The interfaces file uses scripts, that have been templated and placed by Ansible, to apply a correct configuration to wlan interfaces decided by MAC address the prefix.

Tests

A unit test was written for the batnet_interface function, since it contains most of the logic for implementing addressing of nodes by MAC address prefix. Additionally, tests for the batnet_interface function is simpler to implement than e.g. testing of the `/etc/networking/interfaces` file (since that depends on a network, DHCP server, other BATMAN nodes, etc.). The batnet_interface function only depends on the two global variables: BATNET_MACS and IP_LINKS. Therefore a script was written, which can run multiple cases of the global variables that checks the output against an expected result. Such a test case is shown in snippet 12.

```
TEST_ID="dd"
export BATNET_MACS="00:28:f8:30"
export IP_LINKS="garbage"
OUT=$(../batnet-interface.sh.j2); RC=$?
EXPECTED_OUT=""; EXPECTED_RC=1
unittest
```

Snippet 12: Test case of batnet_interface function.

The unittest function tests the actual output and return code against the expected output and return code, which is shown in snippet 13 on the facing page.

```
unittest() {
  printf "\nRan test $TEST_ID... "
  if [ "$EXPECTED_OUT" != "$OUT" ]; then
    printf
    ↪  "Failed, mismatching output! Expected \`$EXPECTED_OUT\`, got \`$OUT\`\n"
    exit 1
  elif [ "$EXPECTED_RC" -ne "$RC" ]; then
    printf
    ↪  "Failed, mismatching return code! Expected \`$EXPECTED_RC\`, got \`$RC\`\n"
    exit 1
  fi
  printf "Success!\n"
}
```

Snippet 13: unittest function.

Compared to test frameworks, this unit test script may seem to lack features (e.g. test coverage calculation). However, as the only use for these tests is to ensure basic functionality and recreate bugs that are discovered, such that they never resurface, extra features are not needed and simplicity is prioritized.

5.2.3 Deploy Node

As mentioned in chapter 4 on page 21, the deploy node is responsible for giving internet access to nodes which will be done via:

- NAT (Network Address Translation);
- *dnsmasq* [42] as DHCP and DNS (Domain Name System) server;
- *hostapd* [43] to create a wireless access point.

It is also responsible for storage, visualization, and configuring nodes. Furthermore, as mentioned in section 4.5 on page 24, it should also act as a master clock synchronization node. It was decided to use NTP as the clock synchronization protocol due to its ubiquity and the Chrony implementation since [44] had good results with Chrony.

The deploy node happens to run NixOS due to prior experiences and the inner workings of NixOS are considered out of scope.

NAT

NAT is configured as shown in snippet 14 on the next page where `lanInterface` and `wanInterface` are strings of systemd's predictable interface names based on the physical location of the connector to the hardware, e.g. "wlp3s0".

```
nat = {
  enable = true;
  internalInterfaces = [
    lanInterface
  ];
  externalInterface = wanInterface;
};

interfaces.${lanInterface} = {
  ipv4.addresses = [
    { "address" = "192.168.1.1"; "prefixLength" = 24; }
  ];
};
```

Snippet 14: NAT configuration.

dnsmasq

DHCP and DNS are provided by dnsmasq, as mentioned earlier, and is configured as shown in snippet 15.

```
dnsmasq = {
  enable = true;
  extraConfig = ''
    interface=${lanInterface}
    domain-needed
    bogus-priv
    dhcp-range=192.168.1.32,192.168.1.63,infinite
    dhcp-option=option:router,192.168.1.1
    dhcp-authoritative
    cache-size=5000
  '';
};
```

Snippet 15: dnsmasq configuration.

Note, the nodes keep the first address they receive from DHCP because of the infinite lease time. This is practical since it means the nodes can be addressed by their IP address without worrying about them changing. Also notice that the DHCP range fits the CIDR (Classless Inter-Domain Routing) block `192.168.1.32/27` which allows up to $2^5 = 32$ addresses and thus makes using *nmap* on the range faster than a common `/24` block. With regards to DNS requests, dnsmasq forwards them to an upstream server if there is a cache miss, which would be a failed attempt to read that block of data.

hostapd

hostapd creates a Wi-Fi access point and it is configured as shown in snippet 16 on the next page.

```
hostapd = {
  # enable if interface name starts with "w" (is hopefully wireless)
  enable = if (builtins.substring 0 1 lanInterface) == "w" then true else
  ↪  false;
  ssid = "Alfred PennyWorth";
  wpaPassphrase = "azcomtek550phat$tacks";
  interface = lanInterface;
  countryCode = "DK";
};
```

Snippet 16: hostapd configuration.

hostapd is only started if the interface name starts with "w", meaning it is a wireless interface. This is done since the lanInterface in principle could be a wired connection, in which case it is not necessary to start hostapd and it would probably also fail to start.

Storage

PostgreSQL is used for storage and is set up as shown in snippet 17.

```
postgresql = {
  enable = true;
  enableTCPIP = true;
  authentication = lib.mkForce ''
    local all all trust
    host all all 0.0.0.0/0 trust
    host all all ::0/0 trust
  '';
  ensureDatabases = [ "batman_testbed_db" ];
  ensureUsers = [
    { name = "readonly"; ensurePermissions =
    ↪  {"DATABASE batman_testbed_db" = "CONNECT";}; }
  ];
```

Snippet 17: PostgreSQL configuration.

Notice that any connection from anywhere is trusted for convenience's sake. However, it is not an issue as the firewall is configured to open relevant ports exclusively on the lanInterface, meaning that there is only access to the database from the secondary channel, which in this case is a password protected Wi-Fi network. The presence of a "readonly" user is ensured, which can be used when one does not want to accidentally modify content of the database, "batman_testbed_db" (more on this in section 5.4 on page 47).

Chrony

Chrony is used as the NTP server and it is configured on the deploy node as shown in snippet 18 on the following page.

```
chrony = {
  enable = true;
  extraConfig = ''
    allow
    local
  '';
};
```

Snippet 18: Chrony configuration.

Observe the `allow` directive that runs chrony in server mode and the `local` directive which makes chrony use local time on the deploy node as a reference clock. [45]

5.3 Battracker

As mentioned in figure 4.4 on page 24 the tracker has an input, which is where the data is collected, and an output, which is where the data is handled. Before the data reaches the output phase it is stamped with a node id and a timestamp. The system was developed in C++, as it allows for OOP (Object-Oriented Programming), which is preferable due to having prior knowledge and experience with OOP.

5.3.1 Module Communication

As mentioned in section 4.4 on page 23 communication between the before mentioned input modules, stamper module and output modules is event-based. When an input module has data to send, it will push this data to a chain of modules with the last one being an output module.

The data produced by the input modules are represented by a class called Metric. Each input module extends the Metric class with its own data structure.

This can be seen in figure 5.2 on the next page where the PacketCapture module has its own PacketMetric which extends Metric (A full overview of the system can be seen in appendix E on page 85).

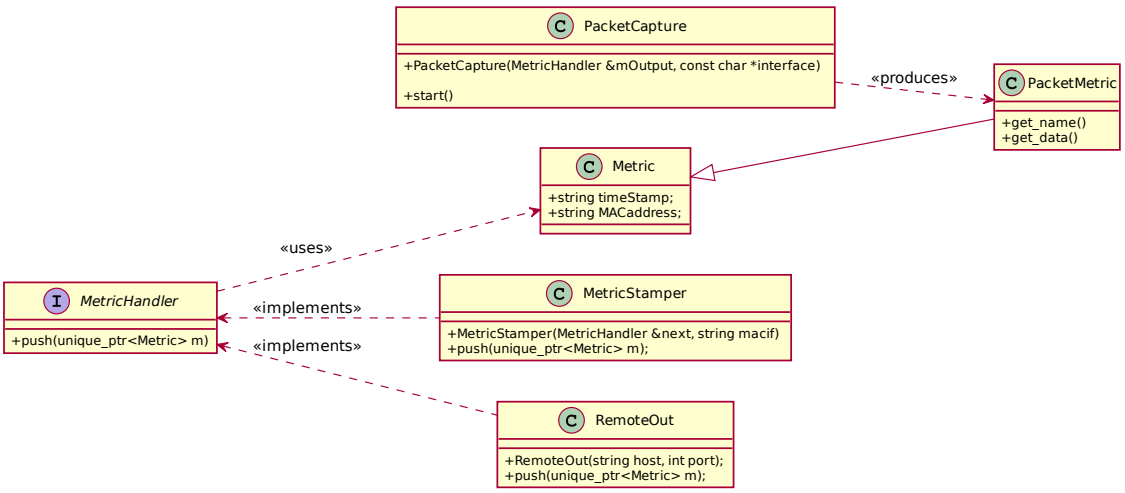


Figure 5.2: Cropped class diagram of the Tracker.

Metric defines two methods `get_name()` and `get_data()`, which classes that extend Metric must implement.

get_data() is used by output modules to get the formatted data in the Metric. get_name() returns the name of the input module.

Snippet 19 shows how PacketMetric overrides these methods. The `m_` prefixed values are set in PacketMetric's constructor.

```
1  std::string PacketMetric::get_name() {
2      return "packet_module";
3  }
4
5  std::string PacketMetric::get_data() {
6      json j;
7      j["type"] = m_type;
8      j["len"] = m_full_len;
9      j["hash"] = m_hash;
10     j["data"] = base64_encode(m_header, m_header_len);
11     return j.dump();
12 }
```

Snippet 19: PacketMetric's implementation of get_name() and get_data().

The stamper module and output modules receives these metrics when they are produced by a input module. This interaction is generalized with the MetricHandler interface, which defines the `push()` method.

All input modules have a reference to a MetricHandler which the input modules will push Metrics to.

The MetricStamper sets the public values `timeStamp` and `MACaddress` on Metric and forwards it to the next MetricHandler in the chain.

The chain of modules is configured at compile time inside the `main()` function of the program. An example of such a definition can be seen in snippet 20.

```
int main() {
    RemoteOut out("192.168.1.1", 1234);

    MetricStamper stamp(out, "wlan1");

    PacketCapture inPack(stamp, "wlan1");
    inPack.start();

    return 0;
}
```

Snippet 20: Example definition and running of module chain.

5.3.2 Input modules

As mentioned in the start of chapter 5 on page 29, the metric gathering tools, called input modules, are meant to be modular.

Packet Capture

One of the input modules implemented is the packet capture input module, which captures all packets received by a node, however every packet type except BATMAN packets are ignored.

The BATMAN packets are categorized into two main categories i.e. special packets which range from `0x00` to `0x3f` and unicast packets which range from `0x40` to `0x7f`. All BATMAN headers start with a packet-type byte which can be used to handle the packets differently, e.g. the OGM packets mentioned in section 2.1 on page 5 has the packet-type `0x00`. All the packets in the unicast category have common headers, which includes packet-type, version, TTL, and destination. [46]

The method used to find packet-type and length of the whole packet without the initial ethernet header, can be seen in snippet 21 on the next page.

```
1  PacketMetric::PacketMetric(uint8_t *packet, size_t len) {
2      // Pull fields from packet and len
3      m_full_len = len;
4      m_type = packet[0];
5      m_header_len = get_packet_size(m_type);
6
7      XXHash32 hash(0);
8      hash.add(packet, len);
9      m_hash = hash.hash();
10
11     // Copy over packet
12     m_header = (uint8_t *) malloc(m_header_len);
13     if (m_header == NULL) {
14         throw runtime_error("Could not allocate packet array");
15     }
16
17     memcpy(m_header, packet, m_header_len);
18 }
```

Snippet 21: The packet metric.

Something to notice is line 7-9, where the packet is hashed. This is done to avoid sending entire packet contents to the remote host. Furthermore, it allowed for easier differentiation of packets across nodes.

The method used to start the packet capture module, can be seen in snippet 22.

```
1  void PacketCapture :: start(){
2      const char *device = m_interface;
3      char error_buffer[PCAP_ERRBUF_SIZE];
4      pcap_t *handle;
5      /* Snapshot length is how many bytes to capture from each packet.
       ↪  This includes*/
6      int snapshot_length = 2000;
7      // capture indefinetly
8  #ifdef TEST_INPUT
9      int total_packet_count = 100;
10 #else
11     int total_packet_count = -1;
12 #endif
13     PacketCapture *my_arguments = this;
14
15     handle = pcap_open_live(device, snapshot_length, 0, 0, error_buffer);
16     pcap_loop(handle, total_packet_count, packet_handler,
       ↪  (u_char*)my_arguments);
17 }
```

Snippet 22: The function starting the packet capture module.

The important parts of the function in snippet 22 on the previous page, is the `pcap_open_live()` and `pcap_loop()`. The pcap_open_live() starts capturing on a network interface, `device`, with a `snapshot length` of 2000, to ensure large packets are handled correctly. 2000 was chosen to have a safety margin from the MTU and it doesn't seem to cause any performance issues. The pcap_loop() function loops the pcap_open_live() and makes it possible to keep capturing packets, which is set in line 8-12 (`total_packet_count`).

Batctl Input Modules

A tool for batman-adv is *batctl* which allows for multiple functionalities such as: throughput meter, neighbor table, route tracing, and much more. It was decided to parse the output of select functionalities to avoid reimplementation.

Batctl Neighbors is an input module that uses batctl for getting a nodes' neighbor nodes. This information is extracted by executing a batctl command on the node, and parsing the command output. However not all the information from this output is needed, and therefore some formatting of that output is necessary. This is be done by extending the commands by using programs that filter standard outputs, such as *awk* and *tail*.
For getting neighbors, the command shown in snippet 23 is executed on the Raspberry pi.

```
1 # batctl n | tail --lines=+3 | awk '{print $2}'
```

Snippet 23: batctl command with formatting functions to get a column of neighbors.

This command gets the node's neighbor nodes and removes the first three lines of that output using tail. This leaves only columns of data, but only the neighbors are needed, so awk is used to only print the column with neighbors. The end result of this command is a string of neighbor nodes' MAC addresses, separated by newlines.

To parse this into the tracker, `popen()` is used, which allows executing commands and reading the output as a stream of strings. This is shown in snippet 24

```
1 output = popen ("batctl n | tail --lines=+3 | awk '{print $2}'", "r");
```

Snippet 24: Using popen to execute a batctl commands.

This output stream is saved to a temporary string and when a newline is found, the string is appended into a vector, and then reset. This process repeats until the end of the stream is reached, and results in a vector containing all MAC addresses of the node's neighboring nodes. The vector with the MAC addresses is then changed to JSON format and sent to the deblobber.

The full cycle of gathering and sending data happens once every second, though with a slight delay from the time it takes to execute the module itself.

Batctl Originators is a module that works almost the same as the neighbor module. They are similar in the sense that output from a command is parsed, and sent to the data store. They are different since multiple columns is wanted from this output, and not simply one column of the output, as was the case with the neighbor table. This output is then added to a JSON object and sent to the deblobber for further formatting. The reason it is not entirely formatted on the tracker is that it is simpler to implement parsing of this command output in Python than C++. It also showcases flexibility because the place of parsing and formatting is left to the implementer. The command is shown in snippet 25.

```
1 # batctl o | tail --lines=+3
```

Snippet 25: batctl command to get originator table for a node.

Executing this command gives that node's originators, when those nodes were last seen, and the destination node for that originator. Lastly, executing and getting the data from the output of this command, as well as sending it to the deblobber is done in the same way that the neighbor module did with the same interval.

batadv-vis Input Module

An additional module was added to the tracker to get batadv-vis data, which is received just like the neighbor and originator modules. The command to get this data is executed and the output is streamed into the tracker. For this module, it is not necessary to do any formatting of the data, and therefore the raw output from the command is added to a JSON object and sent to the deblobber. The batadv-vis command is shown in snippet 26.

```
1 # batadv-vis
```

Snippet 26: batadv-vis command, returning an output to be used later.

5.3.3 Output Modules

All input modules' data is sent to one of these modules, where it is handled. The benefit of making output modules modular is to ease the process of making additions or modifications. Current output modules are *Remote host* and *Local cache*. The way the output module is chosen, is whether or not another host has been put as a command line argument. This should be changed in the future as additional output modules are added, since one can only choose between two output modules at the moment. This means that if it is not possible to setup a remote host node, the data can saved on the node itself.

Remote host

When this module is used to handle the output, it is forwarded to another host. This allows for collection of data on a single device, whereof later analysis of collected data can be done on the remote host.

When the output module is created it creates a TCP (Transmission Control Protocol) socket connection the remote host which is kept open. The metrics data is formatted as JSON and if the socket connection is open, the data is sent through the socket. If the socket has been closed it will try to reconnect `CONN_TRIES`10 number of times. This can be seen in snippet 27.

Each JSON formatted metric is separated by a newline as specified by the JSONL standard. Each line is its own JSON object, meaning they can be parsed separately on the remote host.

```
1  void RemoteOut::push(std::unique_ptr<Metric> m) {
2      json j;
3      j["node_id"] = m->MACaddress;
4      j["timestamp"] = m->timeStamp;
5      j["module"] = m->get_name();
6      j["data"] = m->get_data();
7
8      string dump = j.dump() + "\n";
9      for (int i = 0; ; i++) {
10         if (send(m_socket, dump.data(), dump.length(), MSG_NOSIGNAL) < 0)
           ↪  {
11             if (errno == EPIPE && i < CONN_TRIES) {
12                 cout << "trying to resend" << endl;
13                 sleep(1);
14                 i += reconnect(CONN_TRIES - i);
15                 continue;
16             }
17             throw system_error(errno, generic_category());
18         }
19
20         // Successfull send, break
21         break;
22     }
23 }
```

Snippet 27: The push method of the remote host module.

In snippet 27 it can be seen that JSON keys are set with the object attributes of `m`, which is passed to the function. Afterwards, the JSON object is serialized in order to be passed to the `send()` function. If the send() function fails, it retries and if this fails 10 times a system error is thrown.

This module allows for sending data live to a remote host, which was done to allow for analyzing data whilst conducting tests. This could be beneficial due to the fact that the scientist would be able to get instant feedback when e.g. moving a node.

Local Cache

When this module is used, the data is saved on the node collecting the data. As with the remote host module, metrics are written using the JSONL format to a file. The data is then collectable later. The local cache module push method can be seen in snippet 28.

```
1  void CacheData::push(std::unique_ptr <Metric> m) {
2    myFile.open ("localCache.json", std::fstream::out |
     ↪  std::fstream::app);
3
4    json j;
5    j["node_id"] = m->MACaddress;
6    j["timestamp"] = m->timeStamp;
7    j["module"] = m->get_name();
8    j["data"] = m->get_data();
9
10   string dump = j.dump() + "\n";
11
12   myFile << dump;
13   myFile.close();
14 }
```

Snippet 28: The push method of the local cache module.

In snippet 28 it can be seen that the JSON keys are set from the attributes of the passed object `m`. Lastly, the JSON object is serialized to a JSON string, which is then written to a file.

This output module was built to evaluate if sending data through a secondary channel, whilst conducting a test, would impact the results. As well as, if a secondary channel is not present on the node, it would still be possible to collect data by using this output module. Nodes may be unplugged during the test, which is an unexpected power loss from the perspective of the OS. Fortunately, this will not corrupt the local cache as appends are atomic on ext4 by default [47].

5.4 Deblobber & Storage

As seen in figure 5.1 on page 29, data is sent from the output on the node to the *deblobber*. When data is received on this module it either gets parsed or it gets inserted directly into the database. This is dependent on whether the data is already parsed, such instance is a packet from the batctl Neighbor module. The deblobber module can either be placed on the nodes or on the deploy node. However, as mentioned in chapter 5 on page 29 it was decided that it would be advantageous to place the deblobber on the deploy node in order to use a high-level language without worrying about performance and the use of dependencies without worrying about dependencies on each node.

A class diagram of the deblobber can be seen in figure 5.3 on the following page

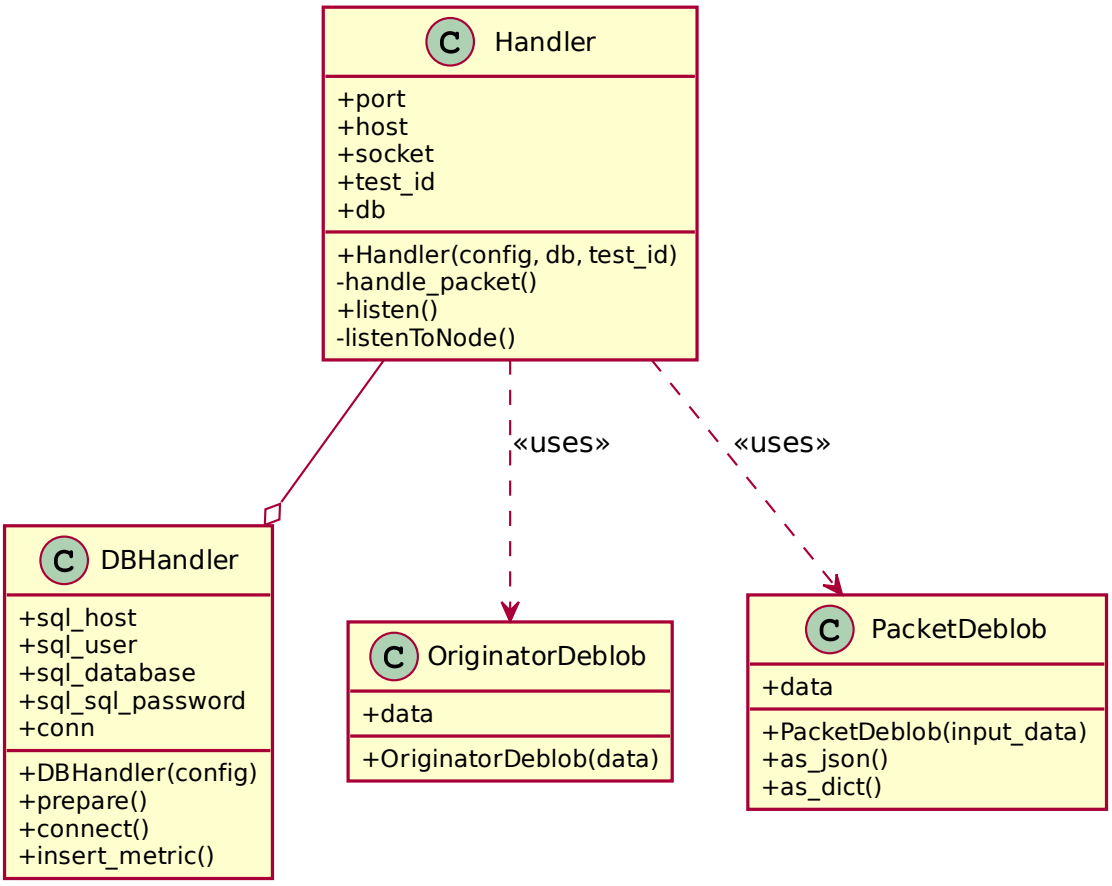


Figure 5.3: Class diagram of the deblobber.

In figure 5.3, it can be seen that the `Handler` aggregates `DBHandler` and uses the `OriginatorDeblob` and `PacketDeblob`. The reason for the aggregation is because when the handler has first identified the packet type and deblobbed the packet, it uses the instantiated DBHandler object to insert the data into the database.

As mentioned in section 5.3 on page 40 all metrics have common fields i.e. timestamp, node_mac, and module, which allows for distinguishing between the metrics and thereby handling them differently. This is done as shown in snippet 29.

```
1  def __handle_packet(self, packet):
2      if packet["module"] == "packet_module":
3          metric = json.loads(packet["data"])
4          metric = PacketDeblob(metric).as_dict()
5          packet["data"] = json.dumps(metric)
6      elif packet["module"] == "batctl_originators":
7          tmp = json.loads(packet["data"])["originator"]
8          packet["data"] = json.dumps(OriginatorDeblob(tmp).data)
9
10     self.db.insert_metric(packet, self.test_id)
```

Snippet 29: Handling packet based on its module.

If the data is from the packet capture module, it is first decoded from Base64. Afterwards, the packet is unpacked by using the *struct* library, with a predefined structure for each packet type. Unpacking is shown in snippet 30 and a predefined structure is shown in snippet 31.

```
1  def deblob(pack_fields, pack_format, packet, macs=[]):
2      ret = pack_fields._asdict(pack_fields._make(unpack(pack_format,
       ↪  packet)))
3
4      for mac in macs:
5          ret[mac] = format_mac(ret[mac])
6
7      return ret
8  [SNIP]
9          if input_data['type'] == 0:
10             # OGM
11             deblobbed = deblob(ogmfields, ogmformat, packet, ["orig",
               ↪  "prev_sender"])
```

Snippet 30: Unpacking OGMs.

```
1  common = "packet_type version "
2  ogmfields = namedtuple("OGM", common +
   ↪  "ttl flags seqno orig prev_sender tq tvlv_len")
3  ogmformat = "!BBBBI6s6sxBH"
```

Snippet 31: OGM structure.

In `ogmformat` the exclamation mark means that the network byte order is big-endian, and each letter represents a data-type defined by the struct library, which is designed to be compatible with C data types. Numbers in the same string are a number of contiguous bytes, and in this case it is used for mac addresses i.e. originator sender and previous sender. This allows for retrieving fields from all the various batman packets. However, if a different version of BATMAN is used as input module it could be so that the deblobber would have to be adjusted accordingly, which is a drawback.

PostgreSQL was chosen as the database for two reasons:

- JSON (JavaScript Object Notation) support, which means it is not needed to predefine all fields that the tracker sends, but rather predefine the ones that are common across all input modules and let the rest be inserted as unstructured JSON. In practice this is implemented using the JSONB data type of postgres, which makes postgres decompose the JSON to a binary format that does not need to be reparsed on each query [48].
- The TimescaleDB extension, which is not used, but was a possible solution in case of performance issues.

After data has been deblobbed, it is inserted into the database as shown in snippet 32.

```
def insert_metric(self, metric, test_id=0):
    parsed_time = parse_time(metric['timestamp'])
    with self.conn:
        with self.conn.cursor() as curs:
            curs.execute("""
                         INSERT INTO batman_testbed_schema.metrics (test_id,
                                                                    timestamp,
                                                                    input_module,
                                                                    node_mac,
                                                                    data)
                         VALUES(%s,%s,%s,%s,%s)
                         """,
                         (
                             test_id,
                                 parsed_time,
                             metric['module'],
                             metric['node_mac'],
                             metric['data']
                             )
                         )
```

Snippet 32: Insert metrics into db.

In snippet 32 the function `execute()` from the psycopg2 library is called, which takes SQL (Structured Query Language) queries. Furthermore, the second parameter is an optional one which takes variables in form of a tuple and maps them into the query string. It can also be seen that the function `insert_metric()` takes a parameter called `test_id`, which makes it trivial to differentiate between the tests later.

To configure which port the deblobber should listen on, which database it should insert data in, and such, a JSON configuration file is used.

Testing

Edge cases of incoming data which were not considered during parsing were predicted to be- and were encountered a few times. Therefore unit tests were written with the *unittest* framework (from the Python Standard Library) that contain such edge cases. In snippet 33 on the facing page and snippet 34 on the next page a test case is shown for different parsers.

```
class TestOriginator(unittest.TestCase):
    def test_sample(self):
        result = {
                 "10:6f:3f:eb:63:90": {
                     "nexthop": {
                         "10:6f:3f:eb:5a:c2": 128,
                         "10:6f:3f:eb:63:90": 255,
                         },
                     "last-seen": 0.030,
                   }
                 }
        data = """\
           10:6f:3f:eb:63:90    0.030s   (128) 10:6f:3f:eb:5a:c2 [     wlan1]
         * 10:6f:3f:eb:63:90    0.030s   (255) 10:6f:3f:eb:63:90 [     wlan1]"""
        dec = OriginatorDeblob(data)
        self.assertEqual(dec.data, result)
```

Snippet 33: Testing of parsing an originator table.

Observe that the incoming data is restructured quite a bit to remove redundant data and in that spirit, the asterisk (which indicates the best route) is removed, since the best route is always the one with the best TQ.

```
def test_elp(self):
    result = {"packet_type": 3,
              "version": 5,
              "orig": "c6:9b:b4:54:11:0e",
              "seqno": 391702878,
              "elp_interval": 3568022221,
              "length": 16,
              "hash": 123}
    data = self.form_packet("0305C69BB454110E1758E95ED4ABB2CD", 3, 16)
    dec = PacketDeblob(data)
    self.assertEqual(dec.as_dict(), result)
```

Snippet 34: Testing of parsing an ELP packet.

5.5 Visualizer

The visualizer exists to illustrate metrics, though metrics which are not directly inserted into the database need to be calculated first. To construct queries that do these calculations, *psql* and/or *Adminer* was used for quick iteration and in cases where visualization is not beneficial (e.g. when the result is a single number).

Grafana was thoroughly evaluated for visualization, but it was found to be incompatible with the current system.

This means the only visualizer component is a program called *httpplotter*.

5.5.1 httpplotter

httplotter is a program that sends plots over HTTP (Hypertext Transfer Protocol) and its main purpose is to render and serve images of the batadv-vis DOT output over a network. It should be accessible from other computers, i.e. the scientist's computer. The main parts of the corresponding source code is shown in snippet 35.

```
@app.get("/vis")
def vis(engine: str = 'fdp',
        filetype: str = 'svg',
        sample: bool = False,
        TT: bool = False,
        node_mac: str = "10:6f:3f:eb:63:98"):
[SNIP]
        with db.conn:
            with db.conn.cursor() as curs:
                curs.execute("""SELECT data ->> 'DOT'
                                    FROM batman_testbed_schema.metrics
                                    WHERE node_mac = %s
                                    AND input_module = 'batadv-vis'
                                    ORDER BY timestamp DESC
                                    LIMIT 1;
                                """, (node_mac,))
                dot = curs.fetchone()[0]
[SNIP]
        graph = graphviz.Source(dot, format=filetype, engine=engine)
    return Response(graph.pipe(), media_type=MEDIA_TYPES[filetype])
```

Snippet 35: Rendering and serving images of batadv-vis DOT output.

To request and view a sample image, visit e.g. `http://localhost:8000/vis?sample=true`, assuming httpplotter is configured to run on port 8000 on localhost, from a browser.

The httpplotter is written in Python with the FastAPI framework for rapid development, which turned out beneficial since useful features were added on-the-fly while testing if the testbed fulfilled the requirements set in chapter 3 on page 19. One such feature is a parameter for removing "TT" (Translation Table) nodes from the output of batadv-vis which is trivial to implement and shown in snippet 36.

```
if not TT:
    noTT = ""
    for line in dot.splitlines():
        if "TT" not in line:
            noTT += f"{line}\n"
    graph = graphviz.Source(noTT, format=filetype, engine=engine)
```

Snippet 36: Removing "TT" nodes from DOT output of batadv-vis.

While "TT" nodes can be removed from the output of batadv-vis with a command line

flag, "TT" nodes would not be present in the database at all if that flag was set.

Output without "TT" nodes is default and such plots will be present in chapter 6 on page 55, but if they are desired, one can visit e.g.

`http://localhost:8000/vis?sample=true&TT=true`.

Configuration of which database the httpplotter should query, which port it should listen on, etc. is also done in a JSON file like the deblobber.

Test 6

In order to verify that the system conform with the requirements set in chapter 3 on page 19 tests were conducted. Each test will verify a requirement and the results of all the tests will be summarized in the end.

6.1 Test setups

Test 1-4 each have their own node setup. Meanwhile, test 5 and 6 uses the node setups found in section 2.3 on page 10.

6.2 Test of Requirements

A number of tests will be run, in order to be able to verify that the requirements are fulfilled. How each test is performed will be described, as well as the success criteria and results.

6.2.1 Test of Requirement 1

The testbed should measure packet loss, delay, jitter, and throughput.

The requirement will be tested by deploying the testbed and sending some packets from on node to another.

The test steps are:

1) Turn on the Raspberry Pis with the golden image.
2) Deploy testbed.
3) Send packets between two nodes.
4) Attempt to determine packet loss.
5) Attempt to determine delay.
6) Attempt to determine jitter.
7) Attempt to determine throughput.

The test will be deemed a success if the attempts to calculate metrics are successful.

Results

In practice, the Raspberry Pis were set up with the topology shown in figure 6.1 though TQ (Transmission Quality) fluctuates a little from the labels on the edges.

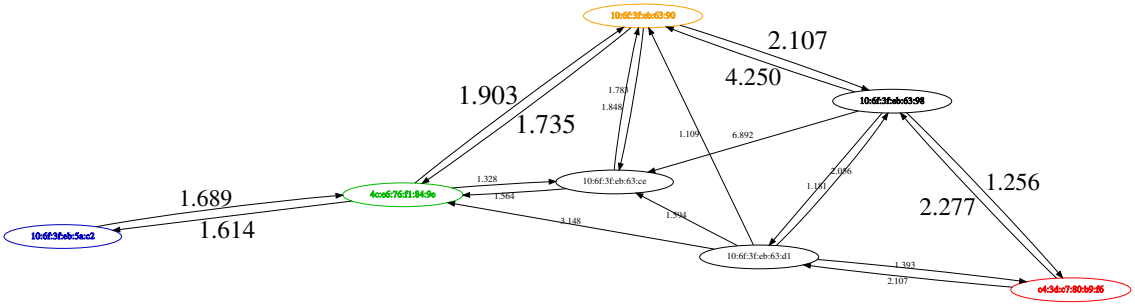


Figure 6.1: Actual test 1 setup with increased text size for relevant TQs.

All tests where done using the blue node (`10:6f:3f:eb:5a:c2`) as the source node while attempting to send data to the green node (`4c:e6:76:f1:84:9e`), yellow node (`10:6f:3f:eb:63:90`), and red node (`c4:3d:c7:80:b9:f6`).

The SQL query in section 6.2.1 is used to find packets from a source, `$src`, to a destination, `$dst`.

```
1 SELECT s.id, d.id, s.timestamp, d.timestamp, s.data FROM
  ↪ batman_testbed_schema.metrics s JOIN batman_testbed_schema.metrics d
  ↪ ON s.data ->> 'meta_hash' = d.data ->> 'meta_hash'
2 WHERE s.test_id = 52221
3 AND d.test_id = 52221
4 AND d.timestamp > (s.timestamp - INTERVAL '1 sec')
5 AND d.timestamp < (s.timestamp + INTERVAL '1 sec')
6 AND s.node_mac = $src
7 AND (s.data ->> 'dst' = $dst)
8 AND d.node_mac = s.data ->> 'dst'
9 ORDER BY s.timestamp, d.timestamp
```

Snippet 37: SQL query used to find packets from a source to a destination

The query returns metrics which match the following:

- Metrics seen by `$src` and received by `$dst`.
- Metric received by `$dst` within one second from when it is seen by `$src`.

This highlights possible problems with the way packets are captured. The query currently captures packets seen by `$src`, and thus also packets which are forwarded by it. Because the BATMAN unicast packet used for transferring data does not contain a destination field, it is impossible to check if a packet was sent by `$src`.

When running this query on the chosen nodes it was discovered that only packets to and from node `4c:e6:76:f1:84:9e` were present. This may be because of the following issues:

- According to collected metrics and figure 6.1 on the facing page, the blue node had a very poor connection to red node.
- Timestamps may not have been in sync between some nodes.

From now on, the test will use the one hop link between the blue and the green node for testing packet loss, throughput and jitter. Test packets were generated using the the *ping* tool, batctl throughput meter and *iperf3*. The associated commands that were run on each node can be seen in appendix F on page 87.

Packet loss can be measured by counting the total number of packets from `$src` to `$dst` and subtract the number of packets actually received by `$dst`. This count is calculated for each minute and plotted results in figure 6.2.

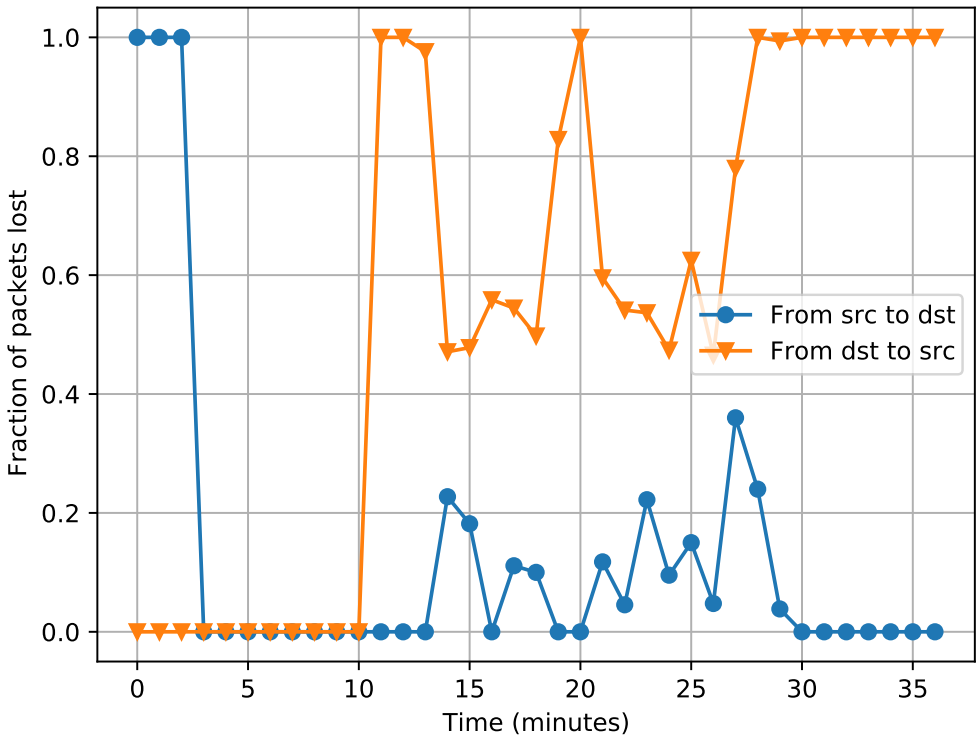


Figure 6.2: Packet loss measured from captured metrics.

It can be concluded that it is possible to measure packet loss, but not whether the given results are correct. To validate results, it would have been beneficial to use tool which reports packet loss.

Throughput can be measured by summing up the length of packets received at `$dst`. Figure 6.3 shows the throughput for each minute in the test.

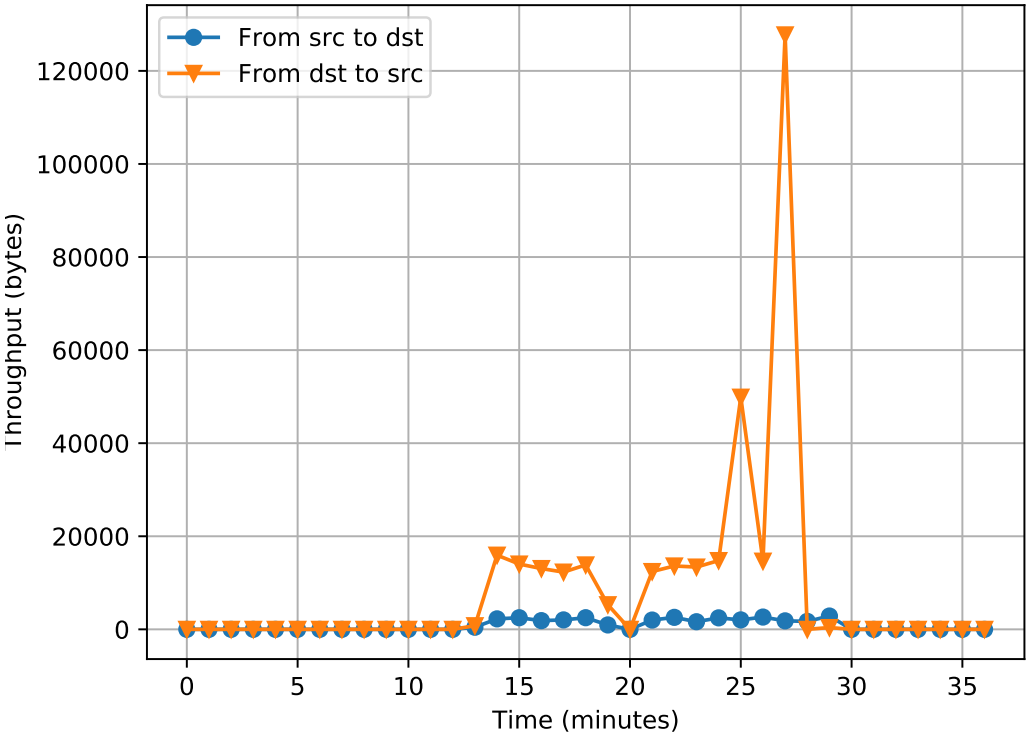


Figure 6.3: Throughput measured from captured metrics.

It can be concluded that it is possible to measure throughput, but again it is difficult to validate these results.

The throughput peaks at 120 000 bytes per minute, which is 2 KB/s. This is much lower than the iperf3 reported throughput of around 7.22 KB/s, but fits well with the batctl reported throughput of 2.99 KB/s.

Jitter can be measured by finding difference between the maximum and minimum delay. Analyzing the data showed a jitter of 996 ms from `$src` to `$dst` and a jitter of 998 ms from `$dst` to `$src` as the worst cases over the span of the whole test.

These results do not correspond to measured jitter from other works as mentioned in section 2.2 on page 7, which may indicate an issue with timestamps. It is therefore possible to measure jitter but maybe with wrong results.

Delay is found as a prerequisite to jitter and it has the same worries about correctness.

Summary Because of the 'hard to verify' results and only being able to capture traffic between two single hop nodes, this requirement is deemed partially fulfilled.

6.2.2 Test of Requirement 2

The testbed should store the measured metrics persistently.

The requirement will be tested by inserting data in the database and restarting the deploy node running the database.

The test steps will be:

1) Insert data into database.
2) Reboot the node with database.
3) Look at data.

The success criteria of this test is that the data still exists on the database after restart.

Results

The amount of rows (entries) in the database, before reboot, can be seen in figure 6.4.

Figure 6.4: Data entries before node reboot.

The amount of rows in the database, after reboot, can be seen in figure 6.5. Additional entries have been inserted, because the database receives data again immediately after the reboot.

Figure 6.5: Database test 2

This requirement is deemed fulfilled, since the database persists all the data after a shutdown.

6.2.3 Test of Requirement 3

The testbed should have adjustable link quality between nodes.

This will be tested by having two nodes connected using BATMAN, and testing their link quality. Then the link quality is decreased and again tested, to verify a difference. The node setup can be seen in figure 6.6 on the next page.

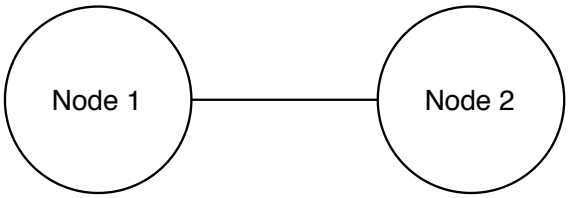


Figure 6.6: Test 3 setup.

The test steps will be:

1) Connect two raspberry pis wireless, using BATMAN.
2) Test link quality.
3) Decrease link quality.
4) Test link quality again, and compare.

It is possible to decrease the txpower (transmission power) on the Wi-Fi modules on the Raspberry Pis, which should reduce the link quality. This can be done by using SSH to one Raspberry Pi on the network and executing the `iwconfig <interface> txpower <val>dBm` command. Interface in this command is the Wi-Fi interface used for BATMAN. Txpower is changed with 'val', which is a value between 0 and 20, where 20 is the default and maximum. The current value can be seen by executing `iwconfig wlan0`. The hypothesis is that by decreasing , link quality worsens.

An example of this command is: `$ iwconfig wlan0 txpower 15dBm`

The test is a success if the link quality decreases after it is changed.

Results

Upon testing this requirement it was found that changing the txpower on the Buffalo Wi-Fi modules created unexpected results. At first the txpower was set to 15 on one of the nodes, and it was immediately apparent that this was too low. By using the batctl neighbors command on the node with adjusted txpower, it could not see any other nodes on the network. Additionally, when using SSH to interface with another node and using the same batctl command, they could not see the txpower-adjusted node either. It seemed adjusting the txpower caused either little effect or broke the connection.

Given the hardware that was used for the development of the testbed it was found that it was not possible to change the link quality satisfactorily. As such, this requirement was not fulfilled, however with other hardware it might prove possible to do so.

6.2.4 Test of Requirement 4

The testbed should have assisted deployment.

This is tested by having three Raspberry Pis with the golden image, and deploying the battracker. Thereafter changing the configuration of the battracker to store locally instead of sending to remote host.

The node setup can be seen in figure 6.7.

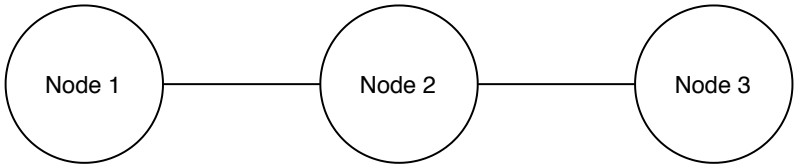


Figure 6.7: Test 4 setup.

The test steps will be:

1) Make a configuration change, in this instance making the tracker use the local cache output module.
2) Deploy the changed configuration on Raspberry Pis with the golden image.
3) Let the test run for 5 minutes.
4) Retrieve locally stored files.

The test is a success if all Raspberry Pis have a locally stored file containing updated data.

Results

It was possible to modify the configuration file that makes the tracker use the local cache output module. Hereafter, the test was run for approximately 5 minutes and the files were retrieved from the Raspberry Pis. All nodes contained a file with updated data of the test, which means that this requirement was fulfilled.

6.2.5 Test of Requirement 5

All scenarios in section 2.3 on page 10 should be able to be carried out on the testbed.

This will be tested by recreating the scenarios, found in section 2.3 on page 10, and testing whether or not they are possible with this testbed.

The test steps for each scenario will be:

1) Attempt to recreate scenario.
2) Determine if it is possible to measure the relevant metrics of the scenario.

For testing this requirement a general test setup was prepared at Niels Jernes Vej to carry out all three scenarios. A combination of the httpplotter topology and an overview of the test building is shown in figure 6.8 on the next page.

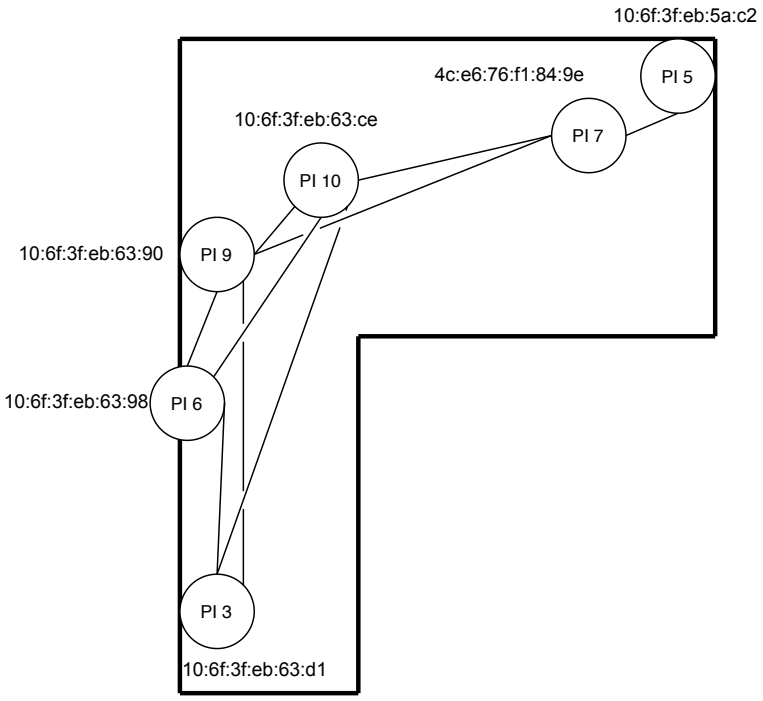


Figure 6.8: General setup, with links between nodes visualised.

This test is a success if all the scenarios are possible.

Results

Best Route. As is evident from the results of testing requirement 3 in section 6.2.3 on page 59, it was not possible to achieve asymmetric adjustable link quality either. As such, it was not possible to test this scenario.

Convergence Speed
This scenario aims to test BATMAN's ability to change the route, if one link in that route is broken. This was done by choosing a node with multiple routes to a destination node, breaking the best route and measuring how fast a new route is converged upon. The chosen nodes were those with MAC adresses: `10:6f:3f:eb:5a:c2` (PI 5) as originator node and `10:6f:3f:eb:63:98` (PI 10) as destination node. One node on the route to the destination node must then be turned off, so BATMAN can find an alternative one. This node was chosen to be `10:6f:3f:eb:63:98` (PI 9).

In order to trace the route that a packet would take through the network the `batctl tr <MAC>` command was executed. When given a valid MAC address as a destination on the network, it prints out a table in which a packet would travel to that destination. This command is executed on the originator node over SSH.

Executing `# batctl tr 10:6f:3f:eb:63:98` on PI 5 gave the output seen in snippet 38 on the next page.

```
# batctl tr 10:6f:3f:eb:63:98
traceroute to 10:6f:3f:eb:63:98 (10:6f:3f:eb:63:98), 50 hops max, 20 byte
 ↪  packets
 1: 4c:e6:76:f1:84:9e  5.425 ms  0.741 ms  0.458 ms
 2: 10:6f:3f:eb:63:90  14.083 ms  1.466 ms  1.444 ms
 3: 10:6f:3f:eb:63:98   *  3.165 ms  2.310 ms
```

Snippet 38: batctl tr executed on PI `10:6f:3f:eb:5a:c2` (PI 5) showing before turning off `10:6f:3f:eb:63:90` (PI 9).

This output in snippet 38 shows three packets traveling the same route through the network, with their latency. It is also seen that the route that these packets took three hops, and the MAC addresses of the nodes it traveled through, with the last one being the destination. The '*' represents a packet loss.

This route can be visualised as shown in figure 6.9

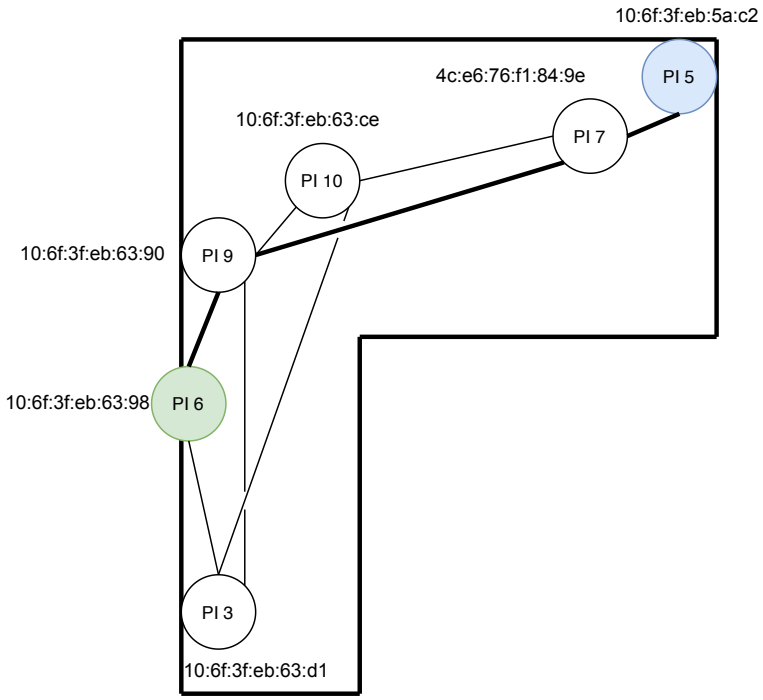


Figure 6.9: Original best route for `10:6f:3f:eb:5a:c2` (PI 5) to `10:6f:3f:eb:63:ce` (PI 10).

After turning off `10:6f:3f:eb:63:90` (PI 9) a new route should be chosen. The same command was executed again on the originator node and its output can be seen in snippet 39 on the following page.

```
# batctl tr 10:6f:3f:eb:63:98
traceroute to 10:6f:3f:eb:63:98 (10:6f:3f:eb:63:98), 50 hops max, 20 byte
↪  packets
 1: 4c:e6:76:f1:84:9e  10.222 ms  7.047 ms  0.915 ms
 2: 10:6f:3f:eb:63:ce  6.649 ms  9.809 ms  7.379 ms
 3: 10:6f:3f:eb:63:d1  8.446 ms  21.526 ms  22.713 ms
 4: 10:6f:3f:eb:63:98  5.359 ms  58.083 ms  12.382 ms
```

Snippet 39: batctl tr executed on PI `10:6f:3f:eb:5a:c2` (PI 5) showing after turning off `10:6f:3f:eb:63:90` (PI 9).

In figure 6.10 this new route is shown.

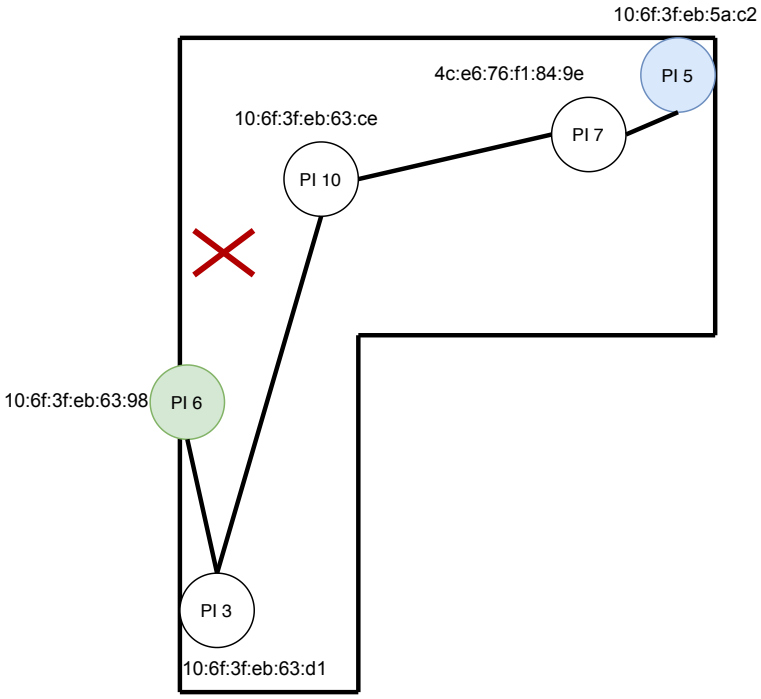


Figure 6.10: New route from `10:6f:3f:eb:5a:c2` (PI 5) to `10:6f:3f:eb:63:ce` (PI 10).

This proves that it is possible to enact the convergence speed scenario.

The topology of the network can be seen on appendix D on page 83 as created by httpplotter.

Mobility

Testing the movement scenario, one node was selected to be moved from one place to another. This node to be moved was decided to be the node with MAC address `10:6f:3f:eb:5a:c2` (PI 5). Using the `batctl ping <MAC>` from node `10:6f:3f:eb:63:ce` (PI 10) to the moving node, it was possible to track when the moving node was disconnected and reconnected, after being moved. The movement of the moving node can be seen in figure 6.11 on the next page.

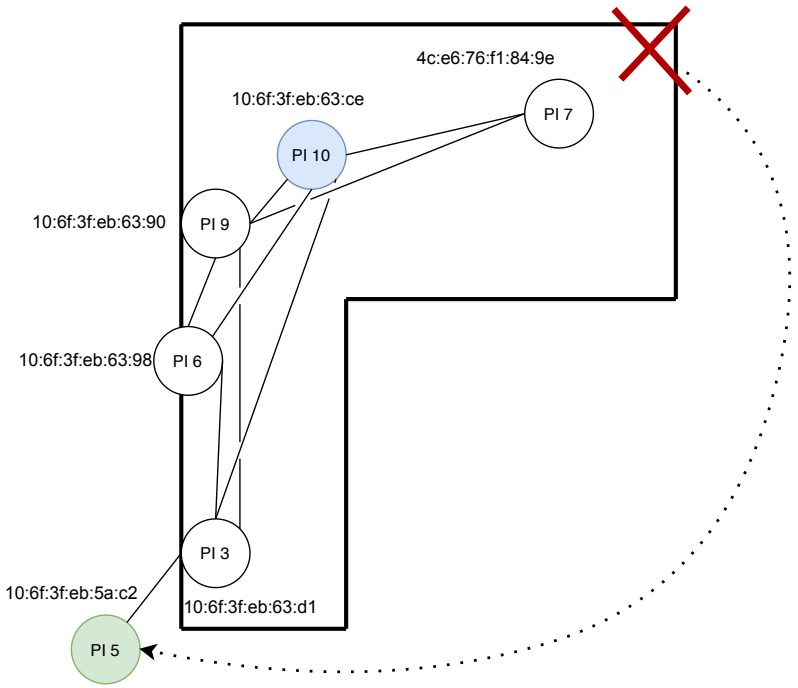


Figure 6.11: Moving `10:6f:3f:eb:5a:c2` (PI 5) to a new location, with updated link(s).

This change in movement can also be seen when looking on the network topology from batadv-vis in figure 6.11.

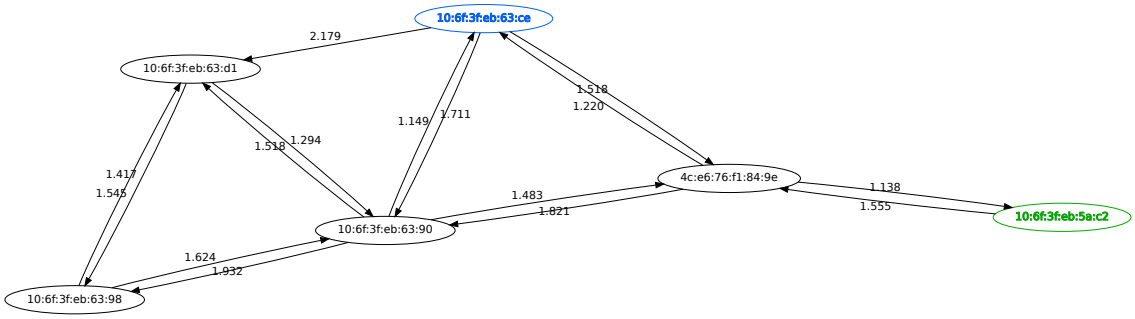


Figure 6.12: Topology before moving `10:6f:3f:eb:5a:c2` (PI 5).

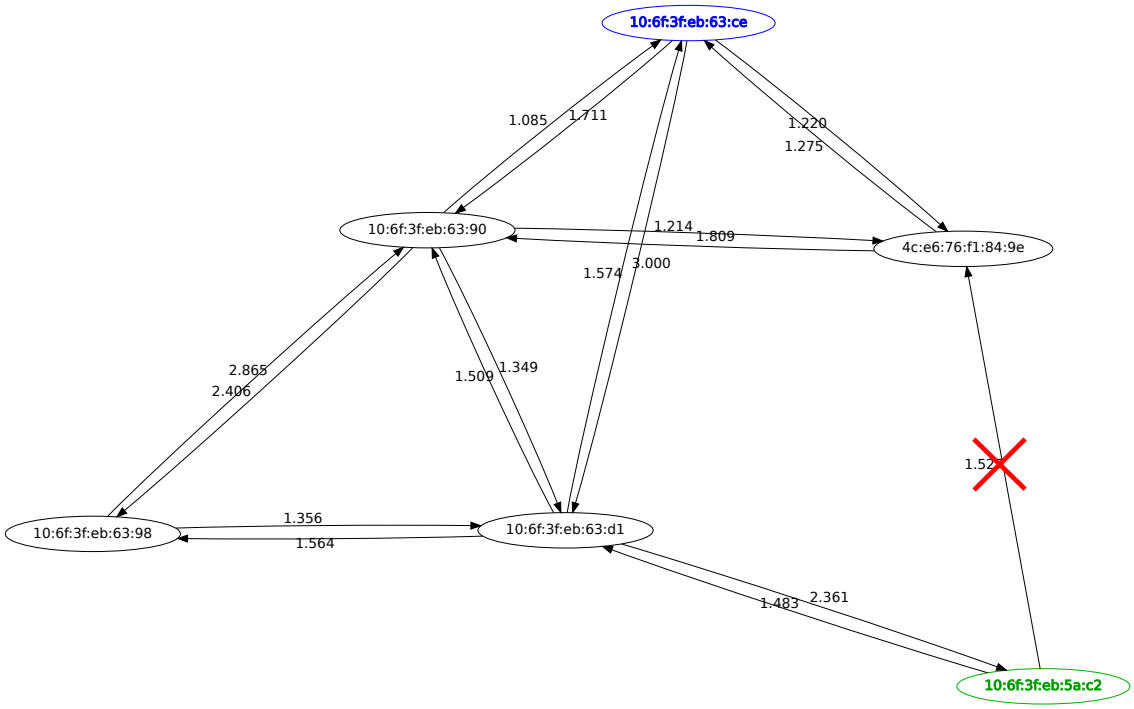


Figure 6.13: Topology after moving `10:6f:3f:eb:5a:c2` (PI 5).

It can be seen in figure 6.12 on the previous page and figure 6.13, that the links have adjusted to the new placement of the moving node. In figure 6.13 a one-way link is visible between the nodes `4c:e6:76:f1:84:9e` and `10:6f:3f:eb:5a:c2`. This does not mean that there is a one way communication between them, but because the node, which the batadv-vis data is from, still thinks that `10:6f:3f:eb:5a:c2` can see the other. This ultimately has no effect on the performance of the network, as this is a batadv-vis drawback and eventually corrects itself.

It can be concluded that the mobility test scenario is possible.

Summary of test 5

This requirement is partially fulfilled. The *Convergence Speed* and *Mobility* tests were successful, however since the requirement from section 6.2.3 on page 59 is not fulfilled, the *Best Route* scenario could not be recreated.

6.2.6 Test of Requirement 6

The results of carrying out the scenarios on the testbed should be replicable.

This will be tested by using the same procedure as 6.2.5 again, however only after the testbed has been torn down.

This test is a success if all the scenarios gives the same results as in 6.2.5.

Results

The testbed was not torn down and set back up, for this requirement to be tested. This was mostly due to lack of resources. However, the convergence scenario was conducted

twice, and the results did not deviate significantly. Having results that do not deviate significantly was how replicability was defined in section 2.5 on page 16. Keeping in mind that a full tear down and set up was not performed, this requirement is deemed partially fulfilled.

6.2.7 Test of Requirement 7

The testbed should collect the local BATMAN state of nodes, using existing tools.

This will be tested by deploying the testbed. Then the batctl input module will collect neighbour table, and originator table and save the data.

The test steps for each scenario will be:

1) Deploy testbed.
2) Have a test run for 5 minutes.
3) Look in the database.

This test is a success if valid neighbor/originator table data are inserted into the database from each node.

Results

These two input modules were run simultaneously with the other input modules, and were set to send every second. It was possible to parse the output of the batctl neighbor and originator commands and format it as JSON to be stored in the database. An example of an originator table in the database can be seen in snippet 40 and an example of a neighbor table can be seen in snippet 41 on the next page.

```
1 {
2 "10:6f:3f:eb:5a:c2": {"nexthop": {"4c:e6:76:f1:84:9e": 155}, "last-seen":
  ↪  1.44},
3 "10:6f:3f:eb:63:90": {"nexthop": {"10:6f:3f:eb:63:90": 199,
  ↪  "4c:e6:76:f1:84:9e": 110}, "last-seen": 0.67},
4 "10:6f:3f:eb:63:98": {"nexthop": {"10:6f:3f:eb:63:90": 60,
  ↪  "4c:e6:76:f1:84:9e": 0}, "last-seen": 0.09},
5 "10:6f:3f:eb:63:ce": {"nexthop": {"10:6f:3f:eb:63:90": 35}, "last-seen":
  ↪  2.17},
6 "10:6f:3f:eb:63:d1": {"nexthop": {"4c:e6:76:f1:84:9e": 129}, "last-seen":
  ↪  0.91},
7 "4c:e6:76:f1:84:9e": {"nexthop": {"10:6f:3f:eb:63:90": 123,
  ↪  "4c:e6:76:f1:84:9e": 224}, "last-seen": 0.34}
8 }
```

Snippet 40: An example of an originator table as JSON.

```
{
"neighbors": ["c4:3d:c7:80:b9:f6", "10:6f:3f:eb:5a:c2",
↪  "10:6f:3f:eb:63:98", "4c:e6:76:f1:84:9e"]
}
```

Snippet 41: An example of a neighbor table as JSON.

It was found possible to retrieve data of the node's local state and the requirement can therefore be deemed fulfilled.

6.3 Summary of tests

The test results are collected in a table and can be seen in table 6.1.

#	Requirement	Result
1	The testbed should measure packet loss, delay, jitter, and throughput.	Partially fulfilled
2	The testbed should store the measured metrics persistently.	Fulfilled
3	The testbed should have adjustable link quality between nodes.	Not fulfilled
4	The testbed should have assisted deployment.	Fulfilled
5	All scenarios in 2.3 should be able to be carried out on the testbed.	Partially fulfilled
6	The results of carrying out scenarios on the testbed should be replicable.	Partially fulfilled
7	The testbed should collect the local BATMAN state of nodes, using existing tools.	Fulfilled

Table 6.1: Results of all tests.

7 Discussion

Whilst analyzing the data of the various tests it was evident that tests of the requirements needed to be conducted again. However, due to lack of resources this was not possible which meant this did not happen.

One of the faults that was found was that time synchronization was not achieved amongst the nodes. This could have been due to the usage of the pcap library in the packet capture input module as seen in section 5.3 on page 40. It was found that libpcap buffers packets to deliver them in bulk, which means that the time stamped by the tracker is not necessarily the time the packets were received [49]. This could have been avoided if the timestamps provided by libpcap were used instead.

During the design of the system (chapter 4 on page 21) it was found that time synchronization would be necessary for it to work. This time synchronization was meant to be achieved through the secondary channel. Therefore two access points were set up, which the nodes could connect to and get synchronized time with the other nodes on the network. However, if a node was not able to reach the secondary network to get synchronized the system would not detect it. Therefore, examination on how to better monitor nodes through the secondary channel should have been implemented. This would increase the ease of deploying the testbed and running tests.

In regards to deploying the system, Ansible was chosen as the deployment tool. This was chosen due to experience and the knowledge of its capabilities which were deemed as satisfactory for the testbed. Be that as it may, similar tools that have the same functionalities as Ansible exists, and it could be beneficial to investigate the potential of using a different deployment tool. It should be mentioned that no flaws of Ansible were found during the development of the system.

Another output module was developed that could solve the lack of a secondary channel called the local cache input module. It was intended to be used as an alternative if no secondary channel were available, to test whether the tracknet interfered with the batnet, and to be used as a failover in case that a node temporarily lost connection to the tracknet and the tracker's buffer gets full. The latter implementation was never realized due to its low priority, however if further development were to be made on the testbed this functionality would be recommended to avoid packets to be lost due to bad Wi-Fi signal. Furthermore, another functionality that was not implemented but considered, was one that could ease the retrieval of local cache files and their insertion into the database.

The testbed was developed with the intent to also accommodate the newer version of the BATMAN protocol called BATMAN V. BATMAN V was never tested and it is therefore

unknown if this version would work on the testbed. Furthermore, one of the reasons for not testing BATMAN V is because it was suspected that batadv-vis would not provide correct TQ values due to an old unresolved feature request of exactly this [50]. batadv-vis was found to be a valuable tool when combined with httpploter, due to the fact that during tests it was easy to debug and have a live overview of how the network acted.

Therefore, it might have been beneficial to allocate more resources to the implementation and extension of existing tools such as batadv-vis and batctl. This could potentially have allowed for better functionalities of the testbed, but packet metrics were prioritized. This prioritization was done because the potential yield of the data is higher if the raw data is analyzed, compared to data being limited by existing tools. Consequently this meant that the usage of some of the tools in this system was unsatisfactory. For instance, the output of batctl and batadv-vis are often unchanged from the previous output. It seems trivial to implement a feature such that these outputs are only sent when it has changed and an effort was started to do so. However, it was not completed since bugs that requires comprehensive tests to reproduce were discovered.

Other pain points of the testbed are that making changes to the golden image requires reflashing memory cards which takes a while. A solution to this would be the network boot, but this would require adequate hardware. Furthermore, a solution for cross-compilation of the tracker software was not found, which would have allowed for compiling the software on the deploy node and it being executable on the nodes. In hindsight it would have been easier if the deploy node was chosen with the same architecture as other nodes. This means source code is transferred to a node, compiled, transferred back, and then sent out to all other nodes via Ansible. Therefore, it could be valuable to investigate the possibility of cross-compilation. Moreover, it could also be beneficial to have static compilation, which would mean that the nodes would not have dependencies managed by Ansible.

Lastly, user experience was mostly disregarded in terms of deployment and the tracker. While it is expected of the scientist to be familiar with CLIs (Command Line Interface), it would be nice to implement a proper CLI. For instance such that the tracker prints instructions instead of segfaulting when presented with an unexpected argument.

Conclusion 8

Wireless ad-hoc networks could possibly be both an alternative and an addition to the internet. However, one of the big issues with these types of networks is the routing of data. One potential solution to this problem is BATMAN. In order to aid the research of BATMAN at Aalborg University a testbed was developed. This gave the final problem statement:

"How can a testbed be made for measuring packet loss, delay, jitter, and throughput of the BATMAN under open-mesh's- and potentially additional routing scenarios at Aalborg University?"

It was found that all metrics could be calculated, although they may be incorrect since it appeared from tests that proper timestamping is not implemented correctly. Three out of four of selected open-mesh's scenarios could be recreated. This was done by using an automated deployment system, which works well, to configure nodes in combination with a program that serves live overviews of the network topology.

While performing open-mesh's scenarios, a tracker program sends metrics to a remote host where metrics are further processed and inserted in a database. Given that the timestamping issue gets fixed and more packet header data is collected, metrics can be used in combination with manually executed programs to calculate additional metrics and generate insight about BATMAN. Not just packet loss, delay, jitter, and throughput, but also:

- The given network topology at any time;
- any local BATMAN state;
- BATMAN's convergence speed;
- BATMAN's ability to select the best route;
- BATMAN's ability to converge on a new route after a previous route has been broken;
- BATMAN's ability to handle mobile nodes.

While several solutions to adjust link quality of the nodes were examined it was not managed to do satisfactorily. As such, one of Open-mesh's scenarios could not be recreated. It also follows that this testbed might be unfit for Aalborg University to recreate

Open-mesh's scenarios due to requirements for physical space, which is not necessarily available in a lab setting.

In conclusion, a testbed that fulfills all set requirements was not created, but it is quite possible that this testbed is a good starting point for further development towards fulfilling all the requirements.

Bibliography

[1] andre. *Freifunk - Free and Open Wireless Community Networks*. 2016-08. URL: https://www.emfcamp.org/schedule/2016/236-freifunk-free-and-open-wireless-community-networks.

[2] A. Neumann, C. Aichele, M. Lindner, and S. Wunderlich. *Better Approach To Mobile Ad-hoc Networking (B.A.T.M.A.N.)* Draft. 2008-03. URL: https://tools.ietf.org/html/draft-openmesh-b-a-t-m-a-n-00.

[3] *B.A.T.M.A.N.* URL: https://wiki.freifunk-franken.de/w/B.A.T.M.A.N. (visited on 2020-11-10).

[4] *About the Battle of the Mesh Organisers*. URL: https://www.battlemesh.org/AboutUs (visited on 2020-11-10).

[5] *Freifunk*. URL: https://wiki.p2pfoundation.net/Freifunk (visited on 2020-11-10).

[6] *B.A.T.M.A.N. V*. URL: https://www.open-mesh.org/projects/batman-adv/wiki/BATMAN_V (visited on 2020-10-16).

[7] Marek Lindner and Simon Wunderlich. *B.A.T.M.A.N meshing protocol kconfig*. 2020. URL: https://github.com/open-mesh-mirror/batman-adv/blob/f2a2e0310dc1c570bdd1439553e897649b000292/net/batman-adv/Kconfig#L26 (visited on 2020-10-14).

[8] Linus Lüssing and Marek Lindner. *bat_v_elp.c*. 2020. URL: https://github.com/open-mesh-mirror/batman-adv/blob/f2a2e0310dc1c570bdd1439553e897649b000292/net/batman-adv/bat_v_elp.c (visited on 2020-10-14).

[9] *Originator Message version 2 (OGMv2)*. URL: https://www.open-mesh.org/projects/batman-adv/wiki/Ogmv2 (visited on 2020-10-16).

[10] "IEEE Standard for Local and Metropolitan Area Networks: Overview and Architecture". In: *IEEE Std 802-2014 (Revision to IEEE Std 802-2001)* (2014), pp. 1–74. DOI: 10.1109/IEEESTD.2014.6847097.

[11] *B.A.T.M.A.N. advanced*. URL: https://www.open-mesh.org/projects/batman-adv/wiki/Wiki (visited on 2020-10-22).

[12] *Tweaking B.A.T.M.A.N. Advanced*. URL: https://www.open-mesh.org/projects/batman-adv/wiki/Tweaking (visited on 2020-10-16).

[13] E. Kulla, M. Hiyama, M. Ikeda, and L. Barolli. "Comparison of Experimental Results of a MANET Testbed in Different Environments Considering BATMAN Protocol". In: *2011 Third International Conference on Intelligent Networking and Collaborative Systems*. 2011, pp. 1–7.

[14] Leonard Barolli, Makoto Ikeda, Giuseppe De Marco, Arjan Durresi, and Fatos Xhafa. "Performance Analysis of OLSR and BATMAN Protocols Considering Link Quality Parameter". In: *2009 International Conference on Advanced Information Networking and Applications* (2009). DOI: 10.1109/aina.2009.28.

[15] Elis Kulla, Masahiro Hiyama, Makoto Ikeda, and Leonard Barolli. "Performance comparison of OLSR and BATMAN routing protocols by a MANET testbed in stairs environment". In: *Computers & Mathematics with Applications* 63.2 (2012). Advances in context, cognitive, and secure computing, pp. 339–349. ISSN: 0898-1221. DOI: https://doi.org/10.1016/j.camwa.2011.07.035. URL: http://www.sciencedirect.com/science/article/pii/S089812211100589X.

[16] Jerry Chun-Ping Wang, Brett Hagelstein, and Mehran Abolhasan. "Experimental Evaluation of IEEE 802.11s PathSelection Protocols in a Mesh Testbed". In: *International Conference on Signal Processing and Communication Systems*. Vol. 4. IEEE, 2010-12, pp. 1–3.

[17] E. Chissungo, E. Blake, and H. Le. "Investigation into Batman-adv Protocol Performance in an Indoor Mesh Potato Testbed". In: *2011 Third International Conference on Intelligent Networking and Collaborative Systems*. 2011, pp. 8–13.

[18] K. Kiran, N. P. Kaushik, S. Sharath, P. D. Shenoy, K. R. Venugopal, and V. T. Prabhu. "Experimental Evaluation of BATMAN and BATMAN-Adv Routing Protocols in a Mobile Testbed". In: *TENCON 2018 - 2018 IEEE Region 10 Conference*. 2018, pp. 1538–1543.

[19] J. Xu, L. Wang, Y. Li, Z. Qin, and M. Zhu. "An Experimental Study of BATMAN Performance in a Campus Deployment of Wireless Mesh Networks". In: *2011 Seventh International Conference on Mobile Ad-hoc and Sensor Networks*. 2011, pp. 341–342.

[20] P. Gupta and P. R. Kumar. "The capacity of wireless networks". In: *IEEE Transactions on Information Theory* 46.2 (2000), pp. 388–404.

[21] D. Seither, A. König, and M. Hollick. "Routing performance of Wireless Mesh Networks: A practical evaluation of BATMAN advanced". In: *2011 IEEE 36th Conference on Local Computer Networks*. 2011, pp. 897–904. DOI: 10.1109/LCN.2011.6115569.

[22] M. S. Singh and V. Talasila. "A practical evaluation for routing performance of BATMAN-ADV and HWMN in a Wireless Mesh Network test-bed". In: *2015 International Conference on Smart Sensors and Systems (IC-SSS)*. 2015, pp. 1–6. DOI: 10.1109/SMARTSENS.2015.7873617.

[23] Elis Kulla Makoto Ikeda, Tetsuya Oda, Leonard Barolli, Fatos Xhafa, and Aleksander Biberaj Michael Keating. "Experimental results from a MANET testbed in outdoor bridge environment considering BATMAN routing protocol". In: *Computing*. Ed. by Spring. Vol. 95. 2013, pp. 1073–1086. DOI: https://doi.org/10.1007/s00607-012-0225-9.

[24] *Routing scenarios*. URL: https://www.open-mesh.org/doc/open-mesh/Routing_scenarios.html (visited on 2020-10-19).

[25] *Graphviz - Graph Visualization Software*. URL: https://www.graphviz.org/ (visited on 2020-11-03).

[26] Marek Lindner. *Translation table in a nutshell*. URL: https://www.open-mesh.org/news/38 (visited on 2020-11-09).

[27] *batctl-0.2.x: Update README's vis section.* 2010-03-22. URL: https://github.com/open-mesh-mirror/batctl/commit/15870b582afd781b41954b6621f523e9544f16d4 (visited on 2020-11-03).

[28] Simon Wunderlich. *vis.c.* 2020. URL: https://github.com/open-mesh-mirror/alfred/blob/dece44cf5626f921b5803273c8e05de7684fb6bf/vis/vis.c (visited on 2020-11-03).

[29] *generic_netlink_howto.* URL: https://wiki.linuxfoundation.org/networking/generic_netlink_howto (visited on 2020-11-09).

[30] *A.L.F.R.E.D - Almighty Lightweight Fact Remote Exchange Daemon.* URL: https://github.com/open-mesh-mirror/alfred/blob/112788d77f54f4779715d40485e2f17130631f51/README.rst (visited on 2020-11-09).

[31] *How NTP Works.* URL: https://www.eecis.udel.edu/~mills/ntp/html/warp.html (visited on 2020-12-14).

[32] *Introduction to Linux Traffic Control.* URL: https://tldp.org/HOWTO/Traffic-Control-HOWTO/overview.html (visited on 2020-12-17).

[33] Shweta Bhandare Sagar Sanghani Timothy X Brown and Sheetalkumar Doshi. *EWANT: The Emulated Wireless Ad Hoc Network Testbed.* Tech. rep. 2003-03-16. URL: https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=1200667&tag=1.

[34] *Raspberry Pi documentation glossary.* URL: https://www.raspberrypi.org/documentation/glossary/ (visited on 2020-12-17).

[35] *Raspberry Pi 4 Computer Model B.* URL: https://static.raspberrypi.org/files/product-briefs/Raspberry-Pi-4-Product-Brief.pdf (visited on 2020-12-17).

[36] *A-Profile Architectures.* URL: https://developer.arm.com/architectures/cpu-architecture/a-profile (visited on 2020-12-17).

[37] *Raspberry Pi OS.* URL: https://www.raspberrypi.org/documentation/raspbian/ (visited on 2020-11-12).

[38] *Network booting.* URL: https://www.raspberrypi.org/documentation/hardware/raspberrypi/bootmodes/net.md (visited on 2020-11-12).

[39] *Automatic detection of hardware interface.* URL: https://bugs.debian.org/cgi-bin/bugreport.cgi?bug=101728 (visited on 2020-11-12).

[40] *Predictable Network Interface Names.* URL: https://www.freedesktop.org/wiki/Software/systemd/PredictableNetworkInterfaceNames/ (visited on 2020-11-12).

[41] *systemd.net-naming-scheme — Network device naming schemes.* URL: https://www.freedesktop.org/software/systemd/man/systemd.net-naming-scheme.html (visited on 2020-11-17).

[42] *Dnsmasq - network services for small networks.* URL: http://www.thekelleys.org.uk/dnsmasq/doc.html (visited on 2020-11-12).

[43] *hostapd: IEEE 802.11 AP, IEEE 802.1X/WPA/WPA2/EAP/RADIUS Authenticator.* URL: https://w1.fi/hostapd/ (visited on 2020-11-12).

[44] Oleg Obleukhov. *Building a more accurate time service at Facebook scale.* 2020-03-18. URL: https://engineering.fb.com/2020/03/18/production-engineering/ntp-service/ (visited on 2020-12-13).

[45] *chrony.conf - chronyd configuration file*. URL: https://chrony.tuxfamily.org/doc/3.4/chrony.conf.html (visited on 2020-12-11).

[46] *Packet Types*. URL: https://www.open-mesh.org/projects/batman-adv/wiki/Packet-types (visited on 2020-12-08).

[47] *ext4 General Information - The Linux Kernel documentation*. URL: https://www.kernel.org/doc/html/v5.4/admin-guide/ext4.html (visited on 2020-12-16).

[48] *PostgreSQL: Documentation: 11: 8.14. JSON Types*. URL: https://www.postgresql.org/docs/11/datatype-json.html (visited on 2020-12-14).

[49] *pcap - Packet Capture library*. URL: https://linux.die.net/man/3/pcap (visited on 2020-12-17).

[50] *batadv-vis: Add support for B.A.T.M.A.N. V throughput*. URL: https://www.open-mesh.org/issues/251 (visited on 2020-12-17).

Additional test scenarios A

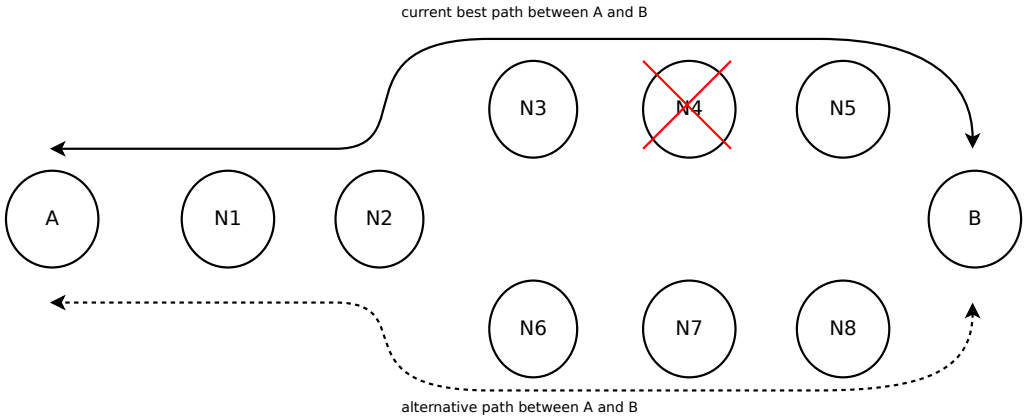


Figure A.1: Alternative to the expanded broken link setup. [24]

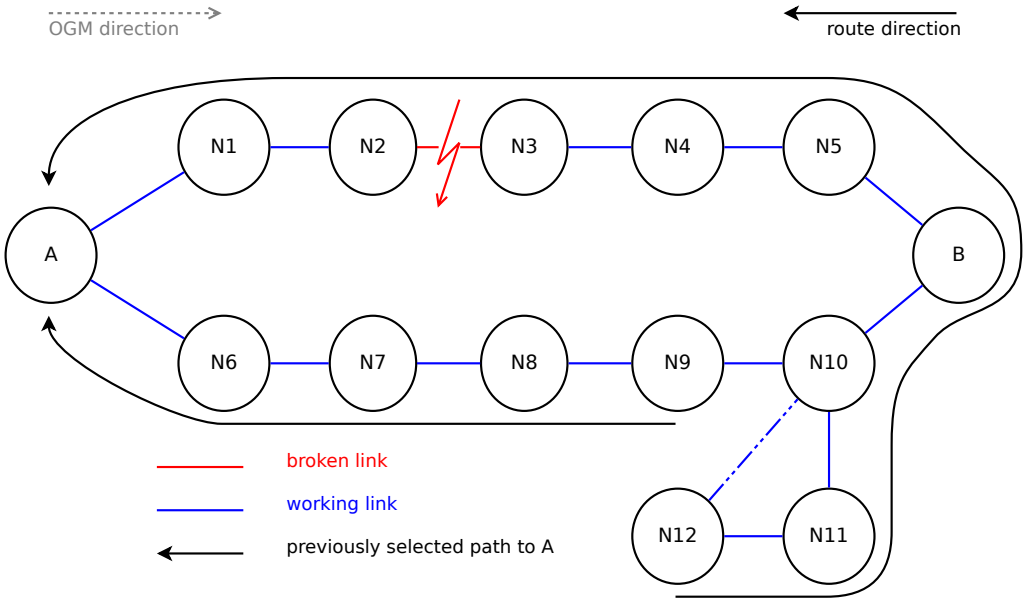


Figure A.2: Alternative to the expanded broken link setup. [24]

batadv-vis DOT output B

```
digraph {
        subgraph "cluster_10:6f:3f:eb:64:06" {
                "10:6f:3f:eb:64:06"
        }
        "10:6f:3f:eb:64:06" -> "c4:3d:c7:80:b9:f6" [label="1.037"]
        "10:6f:3f:eb:64:06" -> "10:6f:3f:eb:63:98" [label="1.032"]
        "10:6f:3f:eb:64:06" -> "be:6b:8d:a5:88:19" [label="TT"]
        "10:6f:3f:eb:64:06" -> "33:33:ff:a5:88:19" [label="TT"]
        "10:6f:3f:eb:64:06" -> "33:33:00:00:00:fb" [label="TT"]
        "10:6f:3f:eb:64:06" -> "01:00:5e:00:00:01" [label="TT"]
        "10:6f:3f:eb:64:06" -> "01:00:5e:00:00:fb" [label="TT"]
        "10:6f:3f:eb:64:06" -> "33:33:00:00:00:01" [label="TT"]
}
```

batadv-vis DOT output C

```
digraph {
        subgraph "cluster_c4:3d:c7:80:b9:f6" {
                "c4:3d:c7:80:b9:f6"
        }
        "c4:3d:c7:80:b9:f6" -> "10:6f:3f:eb:63:98" [label="1.000"]
        "c4:3d:c7:80:b9:f6" -> "10:6f:3f:eb:64:06" [label="1.049"]
        "c4:3d:c7:80:b9:f6" -> "33:33:00:00:00:fb" [label="TT"]
        "c4:3d:c7:80:b9:f6" -> "01:00:5e:00:00:01" [label="TT"]
        "c4:3d:c7:80:b9:f6" -> "1a:fb:92:65:0e:94" [label="TT"]
        "c4:3d:c7:80:b9:f6" -> "01:00:5e:00:00:fb" [label="TT"]
        "c4:3d:c7:80:b9:f6" -> "33:33:ff:65:0e:94" [label="TT"]
        "c4:3d:c7:80:b9:f6" -> "33:33:00:00:00:01" [label="TT"]
        subgraph "cluster_10:6f:3f:eb:64:06" {
                "10:6f:3f:eb:64:06"
        }
        "10:6f:3f:eb:64:06" -> "c4:3d:c7:80:b9:f6" [label="1.032"]
        "10:6f:3f:eb:64:06" -> "10:6f:3f:eb:63:98" [label="1.054"]
        "10:6f:3f:eb:64:06" -> "be:6b:8d:a5:88:19" [label="TT"]
        "10:6f:3f:eb:64:06" -> "33:33:ff:a5:88:19" [label="TT"]
        "10:6f:3f:eb:64:06" -> "33:33:00:00:00:fb" [label="TT"]
        "10:6f:3f:eb:64:06" -> "01:00:5e:00:00:01" [label="TT"]
        "10:6f:3f:eb:64:06" -> "01:00:5e:00:00:fb" [label="TT"]
        "10:6f:3f:eb:64:06" -> "33:33:00:00:00:01" [label="TT"]
}
```

Convergence Speed Topologies

D

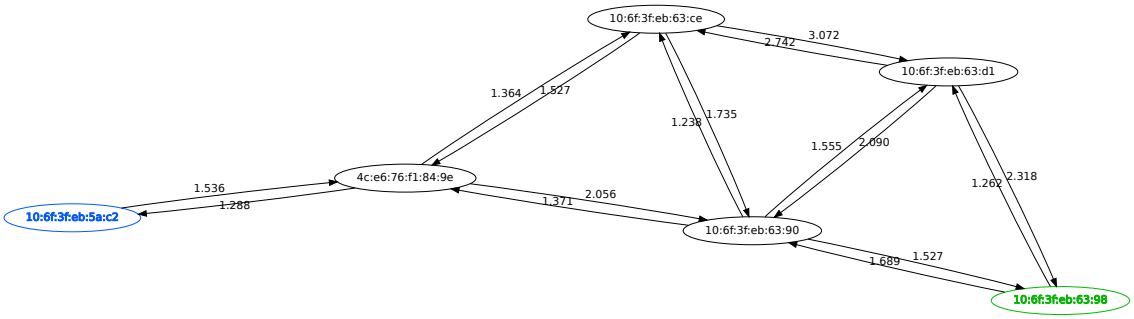


Figure D.1: Before removing node

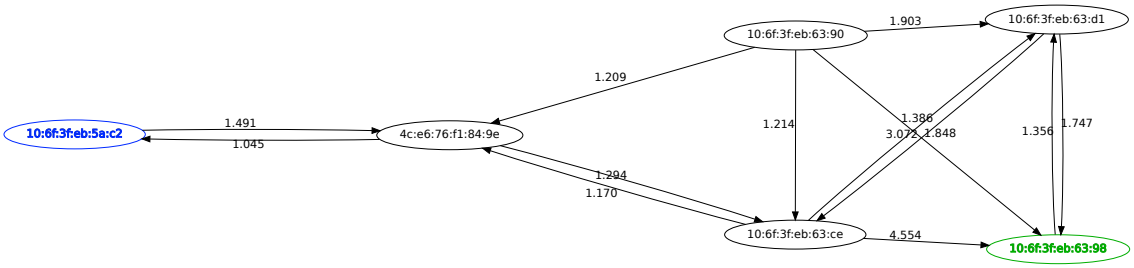


Figure D.2: After removing node

Class Diagram Of Tracker E

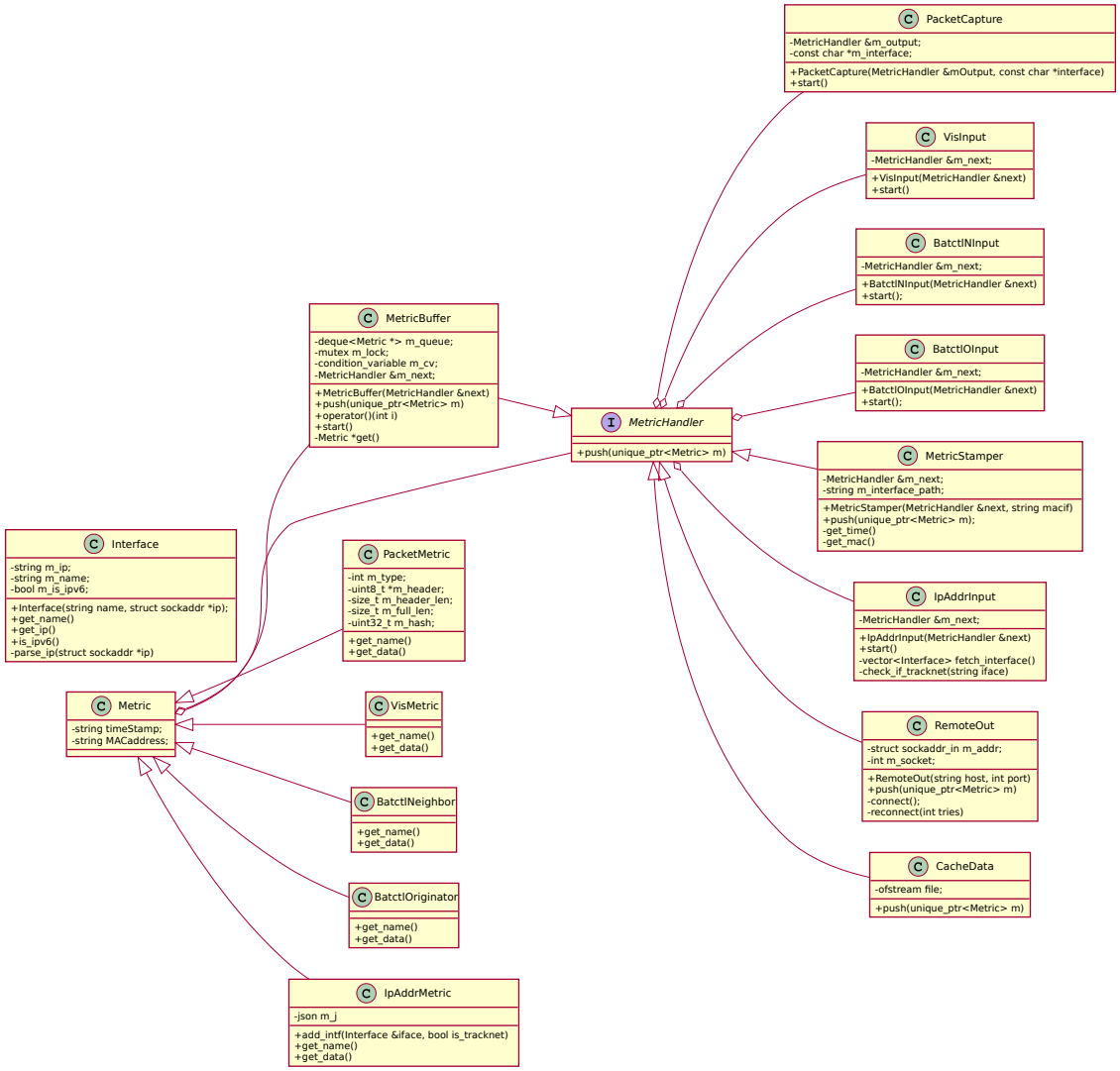


Figure E.1: Class diagram of system

Commands F

F.1 Source node

```
pi@raspberrypi:~ $ ip a
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state
   UNKNOWN group default qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
       valid_lft forever preferred_lft forever
2: eth0: <NO-CARRIER,BROADCAST,MULTICAST,UP> mtu 1500 qdisc
   mq state DOWN group default qlen 1000
    link/ether dc:a6:32:82:34:e3 brd ff:ff:ff:ff:ff:ff
    inet 10.46.0.11/24 brd 10.46.0.255 scope global eth0
       valid_lft forever preferred_lft forever
3: wlan0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc
   pfifo_fast state UP group default qlen 1000
    link/ether dc:a6:32:82:34:e4 brd ff:ff:ff:ff:ff:ff
    inet 192.168.1.42/24 brd 192.168.1.255 scope global wlan0
       valid_lft forever preferred_lft forever
    inet6 fe80::dea6:32ff:fe82:34e4/64 scope link
       valid_lft forever preferred_lft forever
4: wlan1: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1532 qdisc mq
    master bat0 state UP group default qlen 1000
    link/ether 10:6f:3f:eb:5a:c2 brd ff:ff:ff:ff:ff:ff
    inet6 fe80::126f:3fff:feeb:5ac2/64 scope link
       valid_lft forever preferred_lft forever
5: bat0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc
   noqueue state UNKNOWN group default qlen 1000
    link/ether 4a:75:0b:ac:be:53 brd ff:ff:ff:ff:ff:ff
    inet 169.254.8.39/16 brd 169.254.255.255 scope link bat0:
       avahi
       valid_lft forever preferred_lft forever
    inet6 fe80::4875:bff:feac:be53/64 scope link
       valid_lft forever preferred_lft forever

pi@raspberrypi:~ $ iperf3 -s -f K
```

```
-----------------------------------------------------------
Server listening on 5201
-----------------------------------------------------------
Accepted connection from 169.254.6.118, port 54308
[  5] local 169.254.8.39 port 5201 connected to 169.254.6.118
    port 54310
[ ID] Interval           Transfer     Bitrate
[  5]   0.00-1.00   sec  5.66 KBytes  5.66 KBytes/sec
[  5]   1.00-2.00   sec  5.66 KBytes  5.66 KBytes/sec
[  5]   2.00-3.00   sec  8.48 KBytes  8.48 KBytes/sec
[  5]   3.00-4.00   sec  8.48 KBytes  8.48 KBytes/sec
[  5]   4.00-5.00   sec  9.90 KBytes  9.90 KBytes/sec
[  5]   5.00-6.00   sec  7.07 KBytes  7.07 KBytes/sec
[  5]   6.00-7.00   sec  7.07 KBytes  7.07 KBytes/sec
[  5]   7.00-8.00   sec  0.00 Bytes  0.00 KBytes/sec
[  5]   8.00-9.00   sec  12.7 KBytes  12.7 KBytes/sec
[  5]   9.00-10.00  sec  7.07 KBytes  7.07 KBytes/sec
[  5]  10.00-10.97  sec  7.07 KBytes  7.28 KBytes/sec
- - - - - - - - - - - - - - - - - - - - - - - - -
[ ID] Interval           Transfer     Bitrate
[  5]   0.00-10.97  sec  79.2 KBytes  7.22 KBytes/sec
                    receiver
-----------------------------------------------------------
Server listening on 5201
-----------------------------------------------------------
```

F.2 Destination node

```
pi@raspberrypi:~ $ ip a
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state
   UNKNOWN group default qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
       valid_lft forever preferred_lft forever
2: eth0: <NO-CARRIER,BROADCAST,MULTICAST,UP> mtu 1500 qdisc
   pfifo_fast state DOWN group default qlen 1000
    link/ether b8:27:eb:f1:af:f2 brd ff:ff:ff:ff:ff:ff
    inet 10.46.0.11/24 brd 10.46.0.255 scope global eth0
       valid_lft forever preferred_lft forever
3: wlan0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc
   pfifo_fast state UP group default qlen 1000
    link/ether b8:27:eb:a4:fa:a7 brd ff:ff:ff:ff:ff:ff
    inet 192.168.1.53/24 brd 192.168.1.255 scope global wlan0
       valid_lft forever preferred_lft forever
```

```
16      inet6 fe80::ba27:ebff:fea4:faa7/64 scope link
17         valid_lft forever preferred_lft forever
18  4: wlan1: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1532 qdisc mq
       master bat0 state UP group default qlen 1000
19      link/ether 4c:e6:76:f1:84:9e brd ff:ff:ff:ff:ff:ff
20      inet6 fe80::4ee6:76ff:fef1:849e/64 scope link
21         valid_lft forever preferred_lft forever
22  5: bat0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc
       noqueue state UNKNOWN group default qlen 1000
23      link/ether 5e:b3:47:63:a1:1b brd ff:ff:ff:ff:ff:ff
24      inet 169.254.6.118/16 brd 169.254.255.255 scope link bat0
           :avahi
25         valid_lft forever preferred_lft forever
26      inet6 fe80::5cb3:47ff:fe63:a11b/64 scope link
27         valid_lft forever preferred_lft forever
28
29  pi@raspberrypi:~ $ sudo batctl ping 10:6f:3f:eb:5a:c2
30  PING 10:6f:3f:eb:5a:c2 (10:6f:3f:eb:5a:c2) 20(48) bytes of
       data
31  20 bytes from 10:6f:3f:eb:5a:c2 icmp_seq=1 ttl=50 time=3.76
       ms
32  20 bytes from 10:6f:3f:eb:5a:c2 icmp_seq=2 ttl=50 time=2.06
       ms
33  --- 10:6f:3f:eb:5a:c2 ping statistics ---
34  2 packets transmitted, 2 received, 0% packet loss
35  rtt min/avg/max/mdev = 2.056/2.908/3.759/0.852 ms
36  ^Cpi@raspberrypi:~ $ sudo batctl tr 10:6f:3f:eb:5a:c2
37  traceroute to 10:6f:3f:eb:5a:c2 (10:6f:3f:eb:5a:c2), 50 hops
       max, 20 byte packets
38   1: 10:6f:3f:eb:5a:c2  10.781 ms  63.305 ms  35.904 ms
39  pi@raspberrypi:~ $ sudo batctl tp 10:6f:3f:eb:5a:c2
40  Test duration 10110ms.
41  Sent 30996 Bytes.
42  Throughput: 2.99 KB/s (24.52 Kbps)
43
44
45  pi@raspberrypi:~ $ iperf3 -c 169.254.8.39 -f K
46  Connecting to host 169.254.8.39, port 5201
47  [  5] local 169.254.6.118 port 54310 connected to
       169.254.8.39 port 5201
48  [ ID] Interval           Transfer     Bitrate         Retr
       Cwnd
49  [  5]   0.00-1.00   sec  39.6 KBytes  39.6 KBytes/sec    0
       14.1 KBytes
50  [  5]   1.00-2.00   sec  0.00 Bytes  0.00 KBytes/sec    0
       14.1 KBytes
```

```
51 [  5]   2.00-3.00   sec  45.2 KBytes  45.3 KBytes/sec    0
   14.1 KBytes
52 [  5]   3.00-4.00   sec  0.00 Bytes  0.00 KBytes/sec    0
   17.0 KBytes
53 [  5]   4.00-5.00   sec  0.00 Bytes  0.00 KBytes/sec    0
   17.0 KBytes
54 [  5]   5.00-6.00   sec  0.00 Bytes  0.00 KBytes/sec    0
   17.0 KBytes
55 [  5]   6.00-7.00   sec  0.00 Bytes  0.00 KBytes/sec    0
   17.0 KBytes
56 [  5]   7.00-8.00   sec  0.00 Bytes  0.00 KBytes/sec    2
   5.66 KBytes
57 [  5]   8.00-9.00   sec  45.2 KBytes  45.3 KBytes/sec    0
   11.3 KBytes
58 [  5]   9.00-10.00  sec  0.00 Bytes  0.00 KBytes/sec    0
   11.3 KBytes
59 - - - - - - - - - - - - - - - - - - - - - - - - -
60 [ ID] Interval           Transfer     Bitrate         Retr
61 [  5]   0.00-10.00  sec   130 KBytes  13.0 KBytes/sec    2
               sender
62 [  5]   0.00-10.97  sec  79.2 KBytes  7.22 KBytes/sec
                     receiver
63
64 iperf Done.
```