JOHANNES KEPLER
UNIVERSITY LINZ

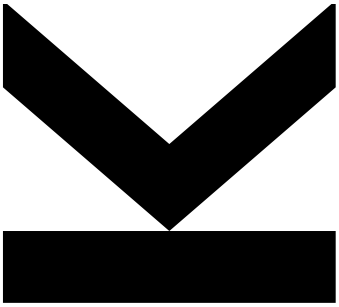
Submitted by
Peter Wagenhuber

Submitted at
Institute of Networks and Security

Supervisor
Michael Sonntag

April 2022

A Decentralized, Resilient and Secure Sensor Network



Master Thesis
to obtain the academic degree of
Diplom-Ingenieur
in the Master's Program
Computer Science

JOHANNES KEPLER
UNIVERSITY LINZ
Altenbergerstraße 69
4040 Linz, Österreich
www.jku.at
DVR 0093696

Abstract

Sensor networks are an essential building block for the so-called “Internet of things”. Many of these networks establish their own OSI layer 2 or 3 network or use a central infrastructure, such as web servers or cloud services, to control the network and collect the data.

In the course of this master thesis a sensor network was implemented that builds on the existing Internet infrastructure without the need for a central instance. The entire functionality of the network is formed by the participating nodes. They are responsible for network management as well as collecting data at one of the participating nodes. A major focus was placed on robustness, which means that the nodes can be reached and participate in the network even in restricted networks, such as through NAT or firewalls. Furthermore, the failure of one or more nodes can be compensated. Finally, all network traffic is encrypted and authenticated.

Zusammenfassung

Sensornetzwerke sind ein wesentlicher Baustein für das sog. “Internet of things”. Viele dieser Netzwerke etablieren ein eigenes OSI Schicht 2 oder 3 Netzwerk oder verwenden eine zentrale Infrastruktur, wie Webserver oder Cloud-Services, zur Steuerung des Netzwerks und dem Sammeln der Daten.

Im Zuge dieser Masterarbeit wurde ein Sensornetzwerk implementiert, dass auf die vorhandene Internetinfrastruktur aufbaut und ohne jegliche zentrale Instanz auskommt. Die gesamte Funktionalität des Netzwerks wird durch die teilnehmenden Knoten gebildet. Diese sind sowohl für die Steuerung des Netzwerks als auch das Sammeln der Daten an einem der beteiligten Knoten verantwortlich. Ein wesentlicher Fokus wurde auf Robustheit gelegt, das heißt, dass die Knoten möglichst auch in eingeschränkten Netzwerken, wie z.B. durch NAT oder Firewalls, erreichbar sind und am Netzwerk teilnehmen können. Des Weiteren kann der Ausfall eines oder mehrer Knoten kompensiert werden. Schließlich wird auch noch der gesamt Netzwerkverkehr verschlüsselt und authentifiziert.

Contents

1. Introduction

Networked sensors are already an integral part of our everyday life. They are the basis for applications such as smart homes, smart cities, smart energy grids, IoT in medicine and agriculture or industry. According to the "Global IoT Forecast Insight Report 2020" by Transforma Insights, there are 8.74 billion IoT connected devices installed worldwide and these numbers will grow to about 25.44 billion in 2030[13].

Sensors and sensor-networks form the foundation of most IoT applications. The figures from the aforementioned report suggest a significant importance of such sensor networks. Especially environmental sensors, such as temperature, humidity, luminosity or gas sensors are vital for smart city or agricultural and industrial applications.

This thesis presents such a sensor network, built from autonomous nodes, focusing on robustness and security, without the need for extra infrastructure like central servers collecting the data or controlling and maintaining network operation.

1.1 Design goals and Definitions

The goal of this work is to build a robust, resilient and scalable network of autonomous nodes for collecting and transferring sensor data. It is built with a strong focus on security, considering the data transferred and the software itself. The network should be used for the exchange and collection of sensor data. It does not create its own OSI layer 1-3 infrastructure but relies on existing underlying TCP/IP network infrastructure.

The focus of the present sensor network lies on numerical and textual sensor data and does not include high bandwidth datastreams like video or audio. Apart from that it does not make any assumptions about the structure of transferred data or enforce any kind of syntax, nor does it provide any semantics. Furthermore, the presentation of the collected data is not part of this work.

The network must be built and managed only by the nodes itself without introducing any kind of extra infrastructure like servers needed for connection establishment, central identity providers or the like.

There should be as little configuration as possible. A node only needs to know its neighbor, that is the neighbors' IP-address or its onion service address and the neighbors' certificate as well as the shared secret for authentication, to join a network. If those informations are not given, the node will start a new network.

The sensor data is collected by a single node that is known to all other nodes and can be retrieved from there. The actual sensing and the processing of the sensor data is not part of this work.

In case of failure the network must be able to recover. Possible failures are for example a downtime of the collecting node or a network split.

The properties already mentioned can be summarized with the following functional and non-functional properties.

1.1.1 Functional Requirements

In order to build and maintain a sensor network every node must:

- Collect sensor data. This means that usually one or more sensors are attached to the node. Since the actual implementation focuses on building and maintaining the network the actual sensor values don't play a vital role, thus a node only produces random values simulating a temperature sensor. Details of the random data generated are presented in section 2.3.

- Send sensor data to the current collector node.

- Keep track of the current collector node and participate in finding a new collector node, if necessary.

- Be capable of acting as collector node, where all other nodes send their data to and the data can be retrieved from there.

In summary this means, that a node sends the data it collects from its attached sensors to a distinguished node in the network, the "collector node". This node must be unique at any time in a connected network. Whenever this collector node changes, the data must be sent to a newly determined collector node. Each node must be able to fulfill the role of a collector node.

1.1.2 Non-Functional Requirements

The non-functional aspects of this sensor network are:

Topology: The network is built and maintained by the participating nodes only. It is paramount to avoid building extra infrastructure alongside the participating nodes, for example running dedicated servers/services for collecting sensor data or network management.

Robustness/Resilience: The network must continue to work regardless of node or TCP/IP network failures. In case of recovery the sensor network must resume normal operation. e.g. A failing router causing a network split recovers, the sensor network must reunite and not continue in a split state.

Security: The application must support the three main security objectives confidentiality, integrity and availability. This means that the sensor data must not be read, understood and altered undiscovered by a third party in transport. Furthermore, the application must be programmed in a way to be failure resistant.

Performance/Scalability: Since sensors are often attached to CPU- and memory-constrained devices and network bandwidth can also be a limiting factor, the application must focus on a low network overhead and a low CPU and memory footprint.

Usability: The configuration effort for a single node should be minimal.

Even though the application shall be designed to be performant and use as little resources as possible, it is not designed to run on “bare metal”, like a microcontroller or similar. The actual implementation relies on the presence of an operating system[1].

The security and usability requirements are partially contradictory. For authentication, confidentiality and integrity it is necessary to configure crypto-keys, shared secrets or the like, which requires additional configuration effort. Concerning network stability and availability it’s a good idea to configure more than one way to access the network which also increases the configuration effort.

There will always be a tradeoff between different requirements, so it’s necessary to prioritize them. The order in which the non-functional requirements were presented reflects the priority. The most important requirement is “topology”, which means that if a requirement of lower priority, like “performance” or “security” would require a centralized server or service it would be dropped in favor of the higher prioritized requirement. The “robustness”, “security” and “performance” requirements are on the same priority level, whereas “usability” is the least important. Whenever an actual decision has been made in the implementation based on this prioritization, reference is made to this prioritization at that point in the text.

1.2 Comparison with existing Systems

The International Organization for Standardization (ISO) defines a sensor network as “system of spatially distributed sensor nodes interacting with each other and, depending on applications, possibly with other infrastructure in order to acquire, process, transfer, and provide information extracted from its environment with a primary function of information gathering and possible control capability”[3]. This definition perfectly matches the sensor network presented in this paper. The term “sensor network” is often used synonymously with “wireless sensor net-

[1] It should run on any UNIX-like OS but was only compiled and tested to run on GNU/Linux and FreeBSD

work"[2].

As already mentioned in the previous section, in contrast to "wireless sensor networks" (WSN), as often described in relevant literature, the present "sensor network" just operates at the application-layer of a TCP/IP network. There are no network operations in the sense of OSI layers 2 and 3. However, it is of course possible to integrate a WSN into the existing network. A node could for example be connected to a WSN and use its sensors like as data sources like the directly connected ones.

One example for a sensor network, that like the one developed for this thesis, just operates on the application layer is OpenSensorHub or OSH for short.

OpenSensorHub is a framework for building sensor networks. It is built primarily on the Open Geospatial Consortium (OGC) Sensor Web Enablement standards. Compared to the system developed in this work, it has a different focus. OSH also provides metadata, like calibration, location, etc., which is not covered by the present work. Additionally, OSH provides a user interface, supports datastreams like video and has several distinct roles for nodes like storage or processing[1], whereas the nodes in the present work are all functionally equal. On the other hand, OSH relies on all nodes being able to communicate with its "Event Bus", a generic bus which transits all events coming from and going to the connected sensors[1]. This means there is need for a central, reachable service realizing this bus. This bus is conceptually similar to the MQTT broker running on the collector node as presented in this work.

A major difference is that OSH does not care about how this bus can be reached from a networking perspective. In contrast to the network developed in the course of this work, OSH offers no built-in way to reach this bus behind a firewall or NAT. Furthermore, OSH does not offer capabilities to recover from network or node failures, like for instance network splits, but the network presented in this paper is designed for that.

[2] e.g. Wikipedia redirects from "sensor network" to "wireless sensor network", a Google search for "sensor network" gives 12 out of 18 top results containing "wireless sensor network" in the title, a search on IEEEXplore gives 40,387 hits searching for "sensor network" but only 4,644 excluding the term "wireless".(the search was performed on a blank browser on Apr. 9th, 2021)

1.3 Architecture and System Overview

As stated in section 1.1, the network consists of autonomous nodes that perform measurements, send the sensor data to a collector node, and track and respond to any change in the sensor network.

The sensor network is built on top of TCP/IP and uses MQTT as OSI layer 7 protocol to collect the sensor data and exchange data about the network itself to build and maintain the sensor network.

MQTT is a lightweight client-server publish-subscribe network protocol, designed for machine to machine communication in constrained environments [9]. These properties were the reason to choose MQTT for this project.

The nodes must be able to directly communicate with at least one neighbor node, since there is no central message relay server. This raises the need for NAT traversal techniques, because it cannot be guaranteed that every node is directly reachable on the network layer. However, the standardized protocols for NAT traversal, that is ICE [21] in conjunction with TURN [17] and STUN [18], all need a central server or service to communicate with. Such servers are not a ubiquitous part of the internet infrastructure and must be run by the sensor network operator. Therefore, this would contradict the most important non-functional requirement as outlined above.

The solution to the NAT traversal problem introduced for the present sensor network is to use the Tor network. The Tor network is an overlay network with "thousands of relays[3] run by volunteers and millions of users worldwide" [8]. This makes the Tor network a ubiquitous part of the internet. Its main purpose is providing anonymization. This is achieved by routing the traffic through several relays and using encryption to ensure every relay has limited access to information about the communication. A relay only knows about its direct neighbors in the communication, the previous and the next relay. Tor also offers the so-called "onion services", that even hide the servers IP address. Besides its main purpose, anonymization, Tor onion services are an effective way to establish a communication despite NAT. The Tor website states "Onion services don't need open ports because they punch through NAT. They only establish outgoing connections."[7]

[3] see https://metrics.torproject.org/

Since the Tor network is an infrastructure everyone can use, and it offers services that help traverse NAT it is suitable for the project at hand. Furthermore, the Tor network is a decentral network and has proven to be quite robust.

1.3.1 Network Operations Overview

To join a sensor network, a node must know at least one neighbor, either via its IP-address and/or Tor onion service address. Furthermore, the neighbors TLS-certificate and a shared secret, used for authentication, must be known. Each node calculates a priority value at startup. This value determines the collector node. The node with the lowest priority value will be the collector node. The priority value calculation details are laid out in Section 2.3.1.

Figure 1.1a shows the first node of a network. It is called node A and has a priority value of 4 and no configured neighbor node. The first node will typically have no neighbor configured, although it's technically possible. The arrows in the graphic mean that an MQTT client (shaft) subscribes to a topic on the broker (arrowhead). Additionally, every node subscribes to the network control topic on the local broker, but this black arrow to the node itself is not shown in the graphic for the sake of readability. The collector node is the node to which all MQTT clients sending sensor values connect. In the graphs, these are the nodes to which all green arrowheads point.

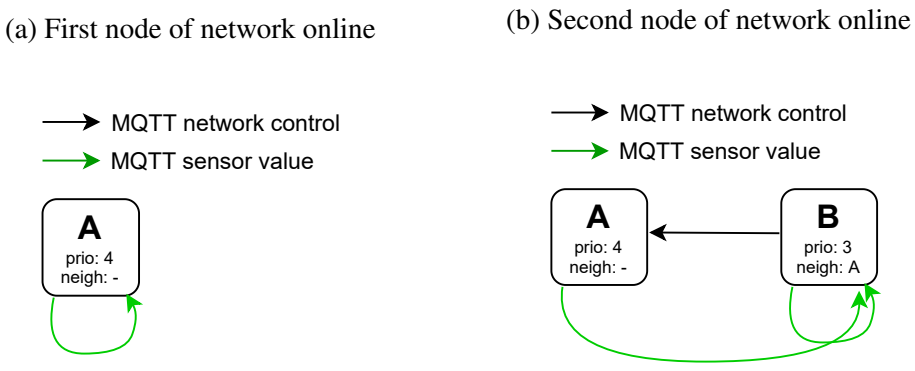


Figure 1.1: The first two steps of building a network

As soon as the second node B with a priority value of 3 and node A as con-

figured neighbor, joins the network, an MQTT client on node B subscribes the network control topic on the broker. This connection is used to exchange information about the nodes and the network the nodes know of. So node A learns that node B has a lower priority value and should therefore be used as collector node. Node A then sends its sensor data to the broker on node B.

Figure 1.2: Third node of network online

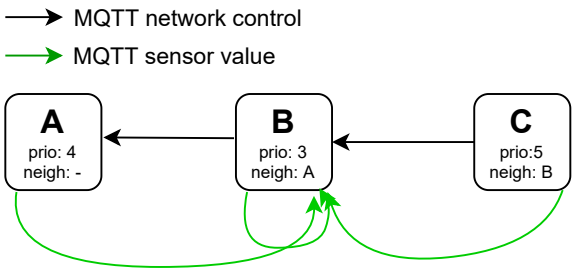


Figure 1.2 shows the network after node C joined the network and exchanged information with node B. Node C comes online and contacts its neighbor node B. It sends information about itself to node B and node B replies with information about the whole network, in this case node A and node B. So node C learns that node B has the lowest priority value in the network, and it should send its sensor data to the broker on node B. Furthermore node C knows that node A has the second lowest priority, so if node B fails it should send its sensor data there.

After node B learns about node C, it informs its neighbors, in our case node A, about the new node in the network. In the end all nodes know the whole network.

2. Networking

Section 1.3 already gave an overview on how the network is built and maintained, as well as how data is collected. In this chapter a detailed insight on all these tasks will be given.

2.1 Building, Maintaining and Managing the Network

This section explains in detail how the autonomous nodes form a network and how this network is able to maintain its operation despite failures, as well as how the network is able to recover from errors. In chapter 4, measurements and their results are presented to support or refute the theoretical assumptions made during the implementation described in this chapter.

2.1.1 Building the Network

After startup and some initial checks, explained in section 3.2, the Tor daemon and the MQTT broker are started. Since the network depends on a working MQTT broker, the program will exit if it isn't able to start the broker.

The successful initialization is followed by the calculation of the priority value, since the collector node is primarily determined by this value. Section 2.3.1 describes the process of calculating the priority in detail.

Now the node has all information needed to start populating its central data structure, the "node list". This is a simple list containing all nodes the node knows of. A node in this list is represented as a "struct" that is defined as follows:

```
pub struct Node {
    onion_url: String,
    broker_port: u16,
    iplist: Option<Vec<IpAddr>>,
    priority: Option<u128>,
    configured_neighbor: bool,
    connection: Option<Connection>,
    is_self: bool,
    ca_cert: String,
    broker_password: String,
    reachable: Option<Reachability>,
}
```

Most fields are self-explanatory, the others are explained below, especially the role they play in building and maintaining the network.

- `onion_url`: Besides obviously being the address on which the node can be contacted via the Tor network, this field has two further features due to its uniqueness. First it acts as a unique ID to find the node in the node list, and second it is used as ultimate decision criterion in determining the collector node. As mentioned in section 2.3.1, there is still a very small chance of having two nodes with the same priority value. In that case the nodes are sorted in lexicographical order of the onion service hostname.

- `configured_neighbor`: If a collector node is not reachable, it will be removed from the node list unless `configured_neighbor` is set to true. This is because nodes that are configured as neighbors in the configuration file will be contacted indefinitely.

- `connection`: The `Connection` type is an enum that encodes via which of the different connection possibilities the node is or was connected. In case the connection succeeded to one of the other node's IP addresses this address is also stored in the field. This field is useful because all possible connections, including the connections via the Tor network, to nodes are initiated in parallel to speed up connection establishment. As soon as one of the threads successfully establishes a connection it sets this field's value and all other threads stop their connection attempts.

- `is_self`: This field has two purposes. First, it serves a similar purpose as

the `configured_neighbor` field, not to stop connecting to the broker, even in case of a (temporary) failure. The broker software used in this project is the `moquitto` MQTT broker[1] and not implemented as part of this project. All communication with the broker on the local host is done via TCP/IP. The broker is started by the node-network daemon and the process is monitored for failure, in which case the broker will be restarted. Although a failure is very unlikely, it is not impossible.

Second, in a received MQTT message containing a node list, the sender can be determined based on the value of this field. This is needed in case the communication is done via the Tor network or the message was relayed via one or more nodes.

- `ca_cert`: This is the certificate (not the file, its contents) in PEM format as defined in RFC7468.

- `broker_password`: This password is needed to authenticate on the broker running on that node.

- `reachable`: This field is especilly important in handling "asymmetric" failures, where a node may be reachable by some nodes in the network but not by others. Detailed information on how it helps in such situations is presented in section 2.1.2. It encodes if and how a node is reachable. It represents five different values:

  - Not yet contacted: This is encoded by the `None` variant of the `Option` enum.
  - Directly reachable: This means that the target node is reachable directly, either via its IP-address or by using the Tor network.
  - Unreachable: This means that the node in the node list can't be contacted.

    A timestamp since when the node is unreachable is stored alongside with this variant of the `Reachability` enum. If a node is unreach-

[1] https://mosquitto.org/

able for more than 10 minutes it will be removed from the node list, unless it is a configured neighbor or the node itself.

- Local: A node also stores a reference to itself in the node list and connects to the broker running on the local host. Therefore the reachability of this node is “Local”.
- Via: This indicates that a node, if not reachable directly, may be reachable through a different node in the network.

After the initialization is complete the node connects as client on the local MQTT broker and subscribes to the network control topic. The network control topic is always `networkl/ctrl` on every broker. It is hardcoded to make sure all clients subscribe the same topic and are able to communicate with each other. If it is the first node of the network, that means that it has no neighbors configured or none of the configured neighbors are reachable, it either just waits for messages coming in, indicating a new node joining, or if a neighbor is configured it additionally regularly tries to connect to the neighbors' broker. As long as the node is the only node in the network, all sensor data is published on the local MQTT broker.

As soon as a second node comes online, the two nodes start exchanging data. In the following, the formation of a small 4 node network will be explained in detail, especially focusing on the messages exchanged and the information each node stores in its node list.

Figure 2.1 shows some parameters of the 4 nodes used in the example illustrating the building of the network.

Figure 2.1: Example configuration of the 4 nodes building the network

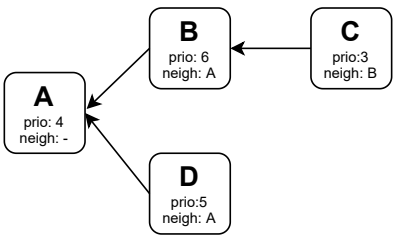
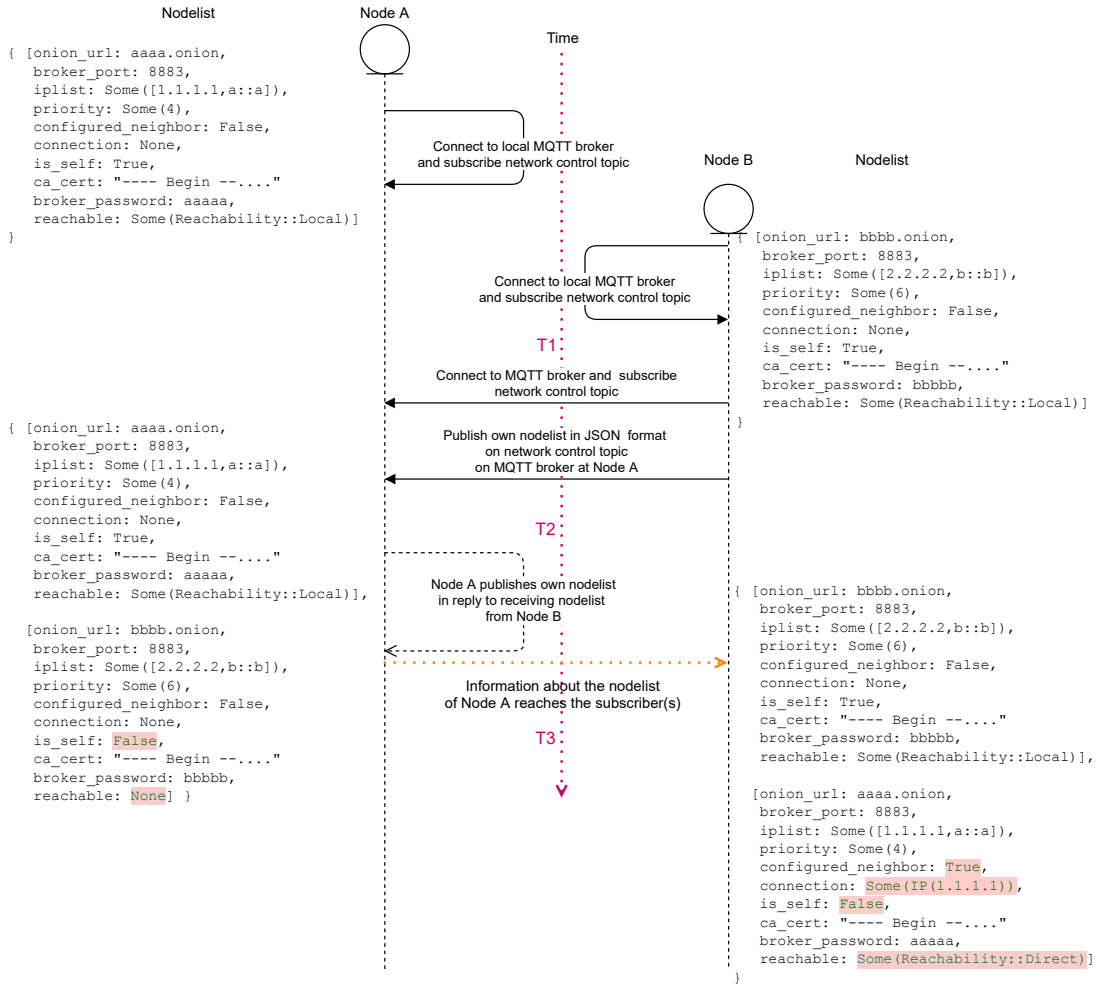


Figure 2.2 shows the first step of building up a network. Node A has already come online and finished its initialization process. Some time later Node B joins the network. The diagram shows the node list in detail to further illustrate the usage of the nodes-structs fields. The black arrows represent MQTT connections and messages, where the dotted line represents a reply to a previous message. The dotted orange line illustrates information flowing from one node to the other.

Figure 2.2: Interaction details of one node joining the network consisting of just one node



At time T1, Node B has come online and already subscribed the network con-

trol topic on the local MQTT broker. Immediately after that, it connects to the MQTT broker running on Node A, because Node A is the neighbor configured for Node B.

On successful connection and subscription to the network control topic on Node A, Node B publishes its node list, at that point in time only containing information about itself, on the subscribed topic. The node list is transmitted serialized in JSON format. Node B keeps being subscribed on the network control topic in order to get informed by others about changes in the network, or to inform other subscribers about such changes.

Node A already subscribed to this topic on its local MQTT broker and receives the information Node B published. It parses the JSON string, changes the `is_self` flag to `False` and the `reachable` field to `None`, indicating that it did not try to connect to this node. After that it adds the information about the new node to its own node list. The `connection` field is still `None` as Node A does not connect to the broker at Node B.

At time T2 Node A already added information about Node B to its node list. In reply to receiving the node list from Node B, Node A publishes its node list on the network control topic on its local MQTT broker. Since Node B is still subscribed to this topic it receives the node list information, and adds the information to its list. It changes the `configured_neighbor` to `True` and sets `connection` to `Some(IP(1.1.1.1))`[2], which is a way to express, that a successful connection was established to a broker with IP-address 1.1.1.1. Additionally it also sets the `reachable` field to `Some(Reachability::Direct)` because it successfully connected to the MQTT broker at Node A.

Finally, at time T3 both nodes know about each other. Since the priority value of Node A is lower than the one of Node B, all sensor data will be sent to the MQTT broker on Node A.

For the sake of readability, the next figures, showing the nodes C and D join the network, are presented with a reduced degree of detail, containing only the relevant information for supporting insight in the process.

[2]This is Rusts `Option` type. It can hold data in its `Some` variant. For detailed information, see `https://doc.rust-lang.org/std/option/enum.Option.html`. The data in this case is again an enum, indicating a connection via IP-address or Tor. How enums work in Rust is explained in `https://doc.rust-lang.org/std/keyword.enum.html`

The sequence of figures 2.3 a-d show the process of a third node joining the network. Figure 2.3a shows the status of the network before Node C comes online. The semantics of the black and green arrows are consistent with the description given in section 1.3.1.

Figure 2.3b shows the network right after Node C coming online. Node C already contacted its neighbor node B, and successfully subscribed the network control topic on this node. The annotation at the arrow shows the publication of Node Cs node list.

Node B incorporates the new information in its node list and answers with the information stored about the network, by publishing its node lists on its local broker, where Node C subscribed the network control topic. This is shown in figure 2.3c. Furthermore, Node B publishes the newly received information on the broker running at Node A. Node B already knows about Node C and its lower priority value than Node A has, so it publishes its sensor values to the broker at Node C, whereas Node A just receives the information and has not incorporated it yet, so Node A still sends the sensor data to its local broker. So for a very small timeframe, not all sensor-data is collected by a single node. This timeframe lasts from Node B starting to send its sensor-data to the broker on Node C, to Node A receiving the information about Node C and adapting the information it has about the network, so it can start sending its data to Node C too.

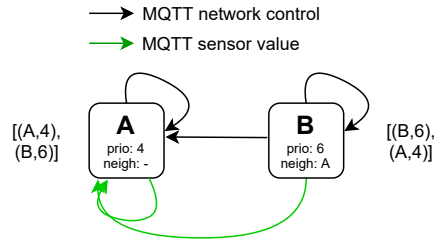
Step 4 in figure 2.3d shows that Node A already added Node C to its node list and then immediately starts to send its sensor-data to the broker on Node C. Because Node A received new information, it answers with its own node list. But Node B already has all the information Node A sent. Now all 3 nodes have the same information about the network and all nodes publish their sensor-data on Node C.

The figures 2.4 and 2.5 illustrate the joining of a fourth node and the propagation of information through the network.

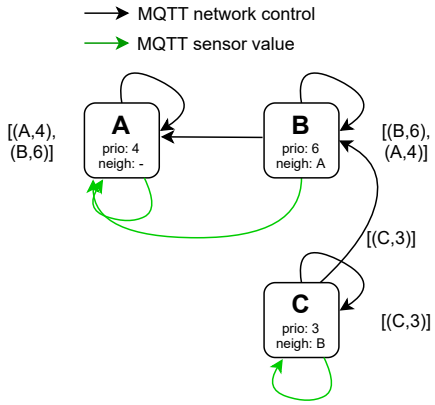
The first step is shown in figure 2.4a and shows Node D, that has already successfully subscribed the network control topic on the broker at its neighbor node A, and publishes its node list.

Figure 2.4b shows Node A answering to the information received at its local MQTT broker by also publishing its node list on the local broker. Furthermore,

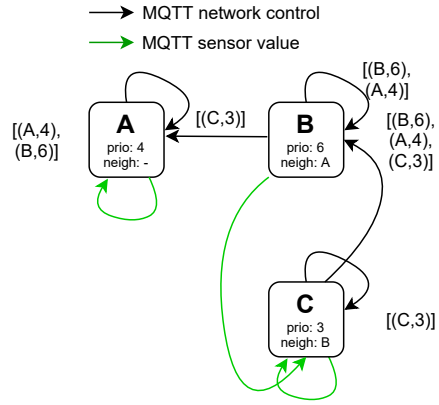
(a) 2 node network



(b) C sends its node list to MQTT broker on B



(c) B answers with its node list and published new node list elements on broker at A



(d) Final 3 node network

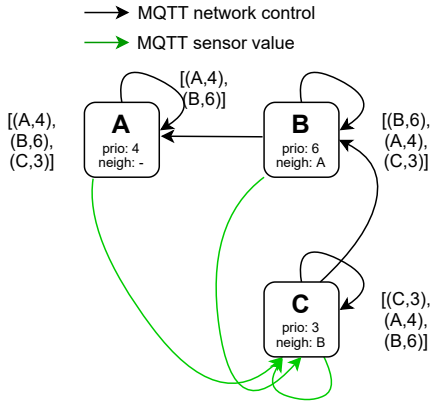


Figure 2.3: Node C joining the network

Node B publishes the new information received from its neighbor on its local MQTT broker. At this point in time Node A and Node B already stored the new information in their node lists. Since the new node has a higher priority value, there is no need to send the sensor-data to a different node. Node D still doesn't know anything about the network, and thus it sends the sensor-data to its local broker.

The final figure in this example shows that Node D got the information about

(a) D joins and connects to neighbor A (b) A answers B redistributes new information

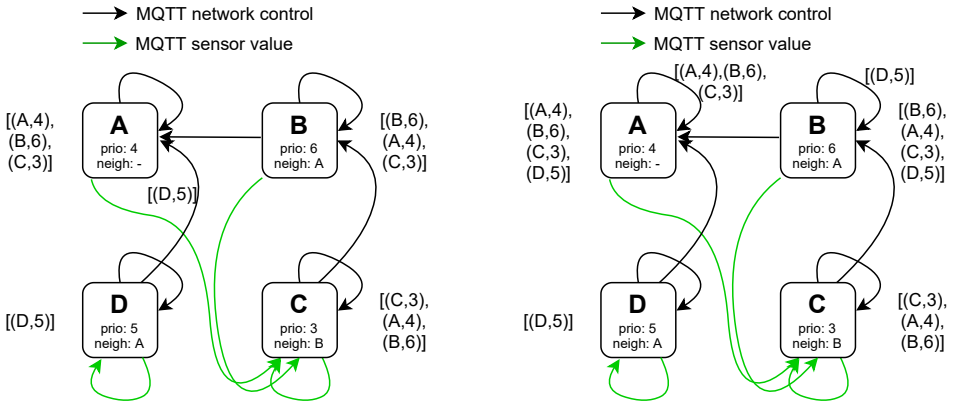


Figure 2.4: Node D joining the network

the whole existing network via its subscription on the broker residing at Node A. It learned that there is a node with a lower priority value, so it starts to send its sensor-data to Node C. Node C learned about the existence of Node D, but no further action must be taken for Node C, because it is still the one with the lowest priority value.

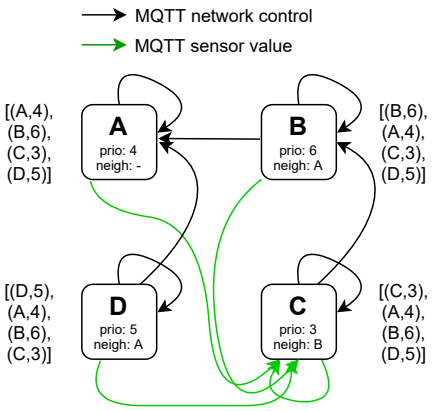


Figure 2.5: Final 4 node network

2.1.2 Maintaining Operations

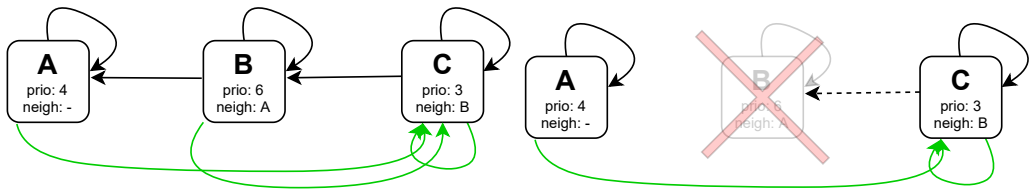
This section describes the behavior of the nodes, and thus the whole network, in case of failures. It describes the prevention of a network split caused by a single node failure and how the nodes react in case the collector node is unreachable.

Network Split

The previous section on building the network gave an overview of how the information of new nodes joining the network propagates through the neighbor links. So all information regarding the network itself is transferred using the direct neighbor links. This makes the network vulnerable to network splits. The network can be represented by a graph with nodes as vertices and the neighbor links as edges. If a node or a link fails the graph could be split in two separate graphs. To avoid a split caused by a single node or link failure additional links must be introduced. In the following this problem and the chosen solution are discussed in detail.

The following sequence of figures illustrate such a network split scenario.

Figure 2.6: 3 node network with failing node



(a) Intact 3 node network

(b) Node B fails

Figure 2.6a shows an intact 3 node network with the black arrows as MQTT network control connections and the green arrows as the MQTT sensor-data connections. The nodes also show their priority values and their configured neighbor nodes.

At some point Node B fails, effectively cutting the information flow between Node A and Node C via the MQTT network control connections. The sensor-data

connection is still active, since Node A directly sends its data to Node C, as both nodes still have the same top priority node in their node list. This changes as soon a node with a lower priority value than the present collector node joins the network on any side of the split.

Figure 2.7 shows such a case, where Node D with a priority value of 2 joined the network by contacting its neighbor Node A.

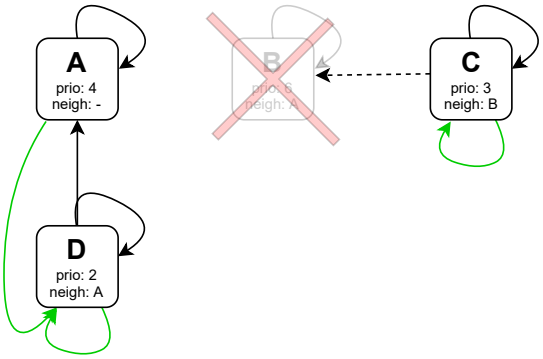


Figure 2.7: Two distinct networks

Node A learns about the new collector node with low priority value, and also redistributes the information on the local MQTT broker. Since no other node is connected to this broker, the information will not be passed on to Node C. Node C still transmits its sensor-data to its local broker, whereas Node A and Node D send their data to the broker on Node D, which leads to a complete network split.

This problem of a single node being able to split the whole network in case of failure can be addressed by guaranteeing that the network graph is at least 2-connected. This means, that at a minimum 2 nodes must fail in order to split the network. There are algorithms to check whether a graph is 2-connected, even quite efficient ones, like described in [22]. Even algorithms for constructing 2-connected graphs exist, like one introduced in [11].

But all this means computing and in a steadily changing network a possibly heavy CPU load on the nodes with typically little computing capabilities and possibly higher network traffic in a bandwidth or latency constrained environment. The solution chosen for this problem doesn't require extra computation. It just

uses an extra network control connection made to the collector node.

Every node in the network not just connects to the MQTT broker at the neighboring node, but additionally to the broker on the collector node, and subscribes the network control topic. This also has the advantage that information about new nodes don't have to propagate node by node through the network, but are distributed to all nodes, subscribed to the network control topic on the collector node, simultaneously. This measure does not necessarily make the graph 2-connected, because it is still possible that one node connects 2 parts of the network and is the collector node. In that case the 2-connectedness is not given. But if that node fails all nodes will recognize that and connect to the node with the next higher priority value.

The following set of figures show the difference in network behavior using this approach compared to the situation described before. Figure 2.8 shows the additional network control connection to Node C.

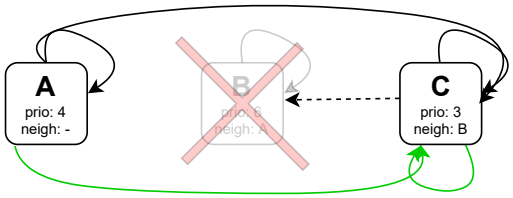


Figure 2.8: Node A with the extra network control connection

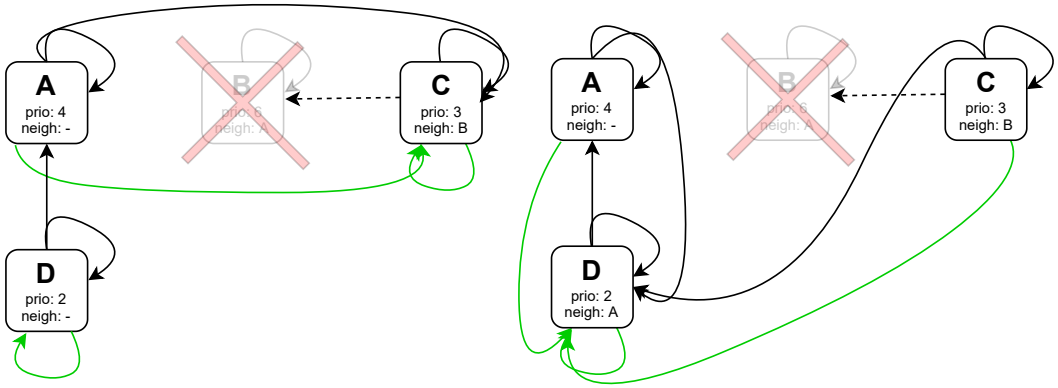
As Node D joins the network, Node A is still connected to Node C and transfers the information of the new node joining to it.

As a consequence Node A and Node C know about the new Node D and learned that it has the lowest priority value, and thus connect to the MQTT broker on Node D, subscribing to the sensor-data topics, as well as to the network control topic.

So this measure successfully prevents network splits triggered by failing nodes that prevent information flowing through the network.

This works even if the failed node B in the previous example was the collector node. Because all nodes connected to the collector node recognize the failure and

Figure 2.9: Node D with lowest priority value joins the network



(a) Node D joins

(b) All nodes connect to Node D

connect to the new collector node and join the network control topic there.

Collector Node Unreachable

If the collector node appears unreachable for some node in the network, the collector node is either unreachable for all others as well, or unreachable for only a part of the network.

It is quite trivial to respond to a collector node failure that makes it unreachable for all nodes in the network. All nodes in the network will immediately recognize the failure. The priority value of this failed node will be set to a higher value. This value is formed from the original value with its most significant bit set to 1. All nodes will then connect to the node with the next higher priority value, which is the lowest priority in the node list then. Furthermore, the failed node will be marked as unreachable with a timestamp since when it is unreachable. If it is unreachable for more than 10 minutes, it will be removed from the node list. (unless it is a configured neighbor node or the node itself)

It gets more complex if the collector node is only reachable by a subset of nodes in the network. This can happen due to firewall restrictions or routing issues. To effectively address this problem two measures are necessary. The first one, the dynamic adjustment of the priority value, has been described in the pre-

vious paragraph. The second one involves sending the message not directly to the collector node, but via a neighboring node acting as a kind of proxy. These two measures and their interaction are described in detail below.

If the nodes that can't reach the collector node just set the priority to a higher value and use the node with the next lower priority as new collector, two, or possibly more, collectors would exist in the network.

A simple measure would be just to inform the other nodes that the collector node cannot be reached from one of the nodes. This can be done by propagating the higher priority value through the network. And this is exactly what is implemented in the network. If one node can't reach the collector all others would then switch to a collector that is reachable by all nodes because they update the priority value stored in their node lists with the higher one.

This is sufficient in most cases, but will fail miserably in case every node in the network is unreachable by at least one other node. In this case all nodes will have higher priority values, but the original order of priority values is still intact. This means no commonly reachable collector node can be found.

To overcome this problem the measure just described can be combined with some sort of "routing" the messages through neighbor nodes that are able to reach the collector either directly or those neighbors relay the messages to another neighbor until it finally reaches the collector. For that the nodes must be able to relay the MQTT messages and be able to inform their neighbors that they can reach the collector.

Relaying the message is quite simple. This is done by a thread that listens on the local MQTT broker subscribing the wildcard topic `sensors/#`. It then sends the message to the collector node or the next "neighbor hop" keeping the original topic intact to be able to identify the source of the message. Section 2.3.2 describes the topic scheme used to transport messages.

The "routing" is implemented using the nodes' reachability information. The basic mechanism works as follows: If a node receives the information from its neighbor that another node is reachable directly from it, the node marks the new node as reachable via the neighbor. This is done with the `reachable` attribute in the node struct introduced in section 2.1.1.

Table 2.1 shows how the information about a new node is processed regarding

the reachability information.

Table 2.1: Reachability handling on information about new node

learned reachability	reachability in node list	action
None	Via(neighbor ID)	inform neighbors
Direct	Via(neighbor ID)	
Local	None	
Via(X)	Via(X)	
Unreachable(Y)	Unreachable(Y)	

The first column "learned reachability" represents the information sent by the neighboring node. The second column "reachability in node list" provides information about the value that will be stored in the node list after receiving information about a new node with the reachability given in the first column. With this information a node can send the sensor information to the newly learned one via the neighbor which the information originated from, if the node is not reachable directly. In case the "learned reachability" is `None`, which means that the node that sent the information did not contact the node in question yet, the reachability information stored in the local node list is `Via(neighbor ID)`. This gives the node an additional opportunity to reach the node in question. Because a node with reachability `Via(neighbor ID)` will be contacted directly first and only in case of a failure it will be contacted via its neighbor. If the entry in the node list would be just `None` the possibility to reach the node via a relay would be lost.

After storing the node information in its node list a node informs its neighbors, depending on where the information originated from. It will inform the neighbors where it is connected remotely, if the information was received on the local broker or vice versa.

If the information about a node is already stored in the node list, the reachability values that will be updated depend on the values that are already present. Table 2.2 shows the details of this update process:

If the reachability stored in the node list is *Local* or *Direct* the information regarding reachability contained in the message will be ignored. This is because *Local* in the node list always refers to the node itself and this does not change, even if the local MQTT broker goes down. In that case it will be started again. If

Table 2.2: Reachability handling on updated information about node

entry in node list	learned reachability	updated reachability	action
None	None	None	
	Local	None	
	Direct	Via(neigh ID)	
	Via(X)	Via(X)	
	Unreachable	Unreachable	set priority high + inform neighbors
Unreachable	None	Via(neigh ID)	
	Unreachable	Unreachable	
	Via(X)	Via(X)	set priority high + inform neighbors
	Local	None	
	Direct	Via(neigh ID)	
Via(X)	None	Via(X)	
	Local		
	Via(Y)		
	Direct		
	Unreachable	see description in text	

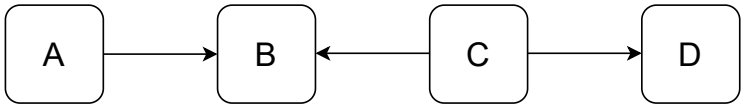
the reachability of a node is stored as *Direct* this means that a successful connection to this node has been established. This will only be changed in case such a connection fails.

The case that can't be described in a meaningful way in this table is when the value stored in the node list is *Via(X)* and the message from a neighbor says this node is *Unreachable*. If the originator of the *Unreachable* message is Node X as stored in the *Via(X)* reachability information, the reachability in the node list will be set to *Unreachable*. Otherwise the information will be ignored. Additionally, the priority value will be set to a higher value, using the already explained mechanism, if the value in the node list was changed from *Via(X)* to *Unreachable*.

The following four node example will illustrate this. Figure 2.10 shows a four node network, where Node B is the node that will receive the updates on the reachability of Node D.

Node B stored the reachability of Node D as *Via(C)* in its node list. If it receives a message stating Node D is *Unreachable* from Node A, it will simply ignore it, because Node A is not the node to that will relay messages for Node

Figure 2.10: Four node network for explaining the *Via(X)* update process



D. If however Node C sends a message containing "Node D is *Unreachable*" to Node B, it will update its node list with the new information and inform Node A about it.

After the introduction of the concepts to solve the problem of the collector node being only reachable for a subset of the networks nodes, the following two example networks demonstrate how these two concepts work together to build a functional network.

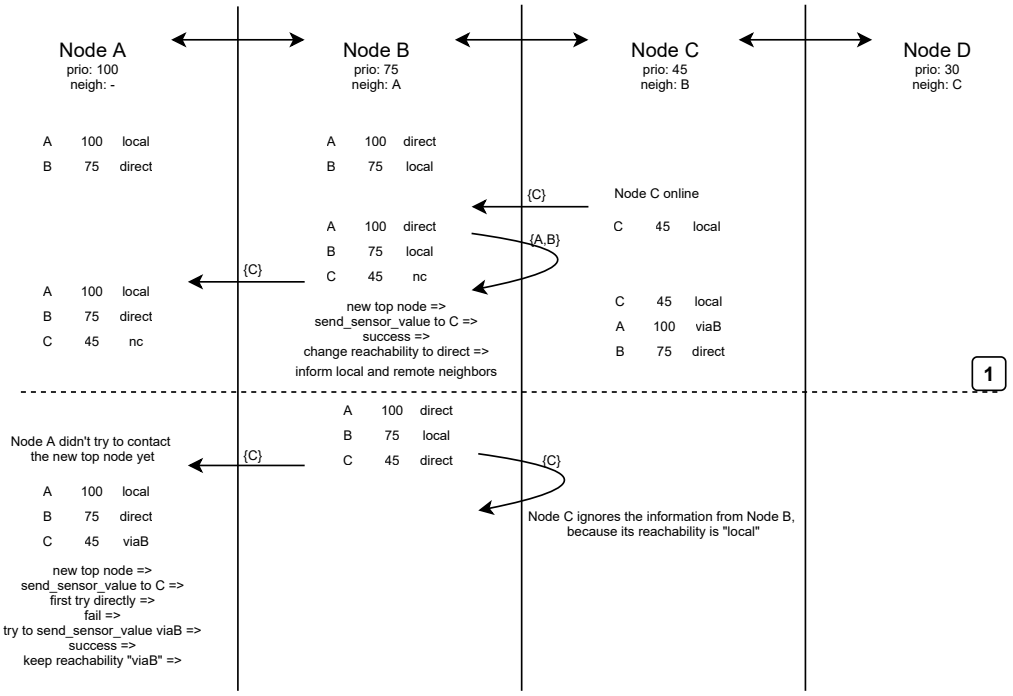
The diagram in figure 2.11 represents a four node network, where the nodes are only able to reach their neighbors. This works in both directions as indicated by the arrows on top of the diagram. For example Node B is able to reach its neighbor Node A and vice versa. For each node the most relevant parts of the node list are shown.

The first state of the network shown is after Node B contacted Node A and both agree on Node B as collector node. After that Node C, with a lower priority value than Node B, joins the network. It sends its node list, only containing information about itself, to Node B. Node B adds this information to its node list but sets the reachability to `None`[3], which means that there was no connection attempt yet. Node B answers with its node list on its local MQTT broker, informing Node C about the presence of Node A and the details about Node B. Node C now knows that it can contact Node B directly and Node A via Node B, because Node A is reachable directly from Node B.

After Node B incorporated the information about Node C in its node list it forwards the information to Node A. Node A also stores the information about the new node in its list. At the same time Node B informs Node A about the newly joined node, it tries to send its sensor data to Node C, because of the lower priority value. This will succeed, so Node B can set the reachability to "direct".

[3]In the diagram this reachability is written as "nc" which should express "not connected yet"

Figure 2.11: Four node network, where the nodes are only able to reach their neighbor



At time [1] two different further courses are possible. The first possible course is shown in black, the second one, in figure 2.12 in blue.

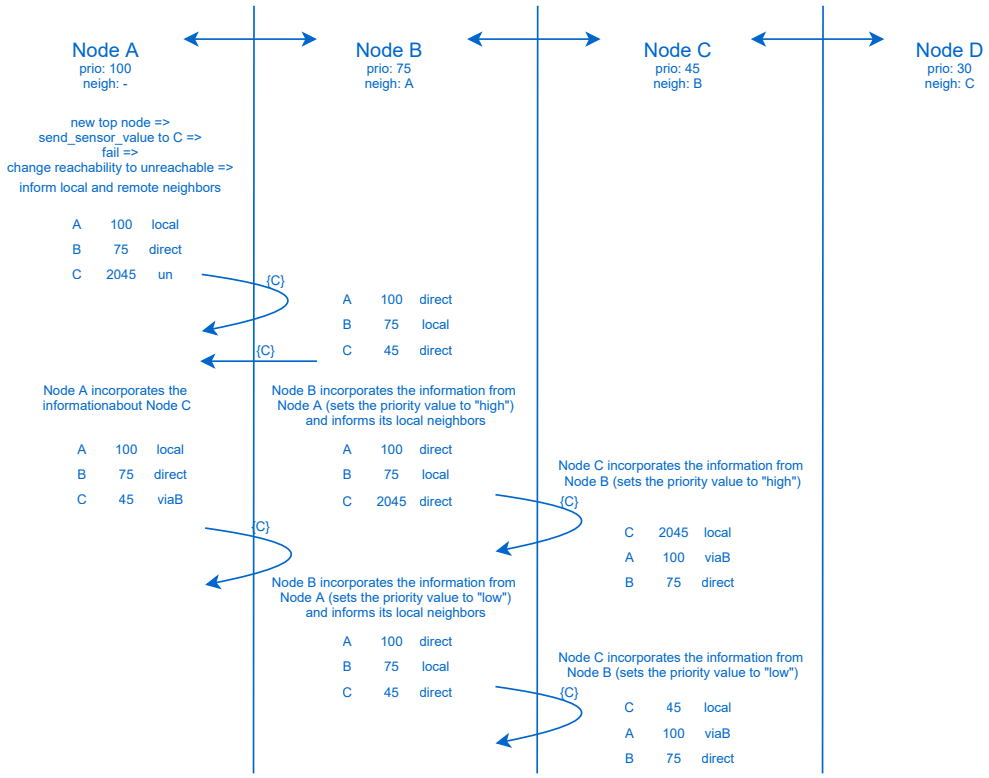
If Node B is faster connecting to Node C and informing Node A about the new reachability of Node C in its node list, Node A will have the information that Node C is reachable “viaB” immediately.

Node A tries to contact Node C directly, via its IP-addresses or via the Tor network, first, but this fails, and therefore keeps using Node B as proxy to send its sensor data to Node C.

The alternate course is a bit more complex but leads to the exact same result as the one just shown.

In that case Node A was faster contacting Node C and realizing that it is not reachable for it, than Node B was able to inform it that Node C can be reached using Node B as proxy. So Node A first sets the reachability to “unreachable” (denoted as “un” in the diagram) and increases the priority value as already de-

Figure 2.12: Four node network, where the nodes are only able to reach their neighbor - alternate course



scribed[4]. As a consequence Node A informs Node B about this new state and Node B also increases the priority and informs its neighbors in turn. This process is then also repeated on the neighboring nodes.

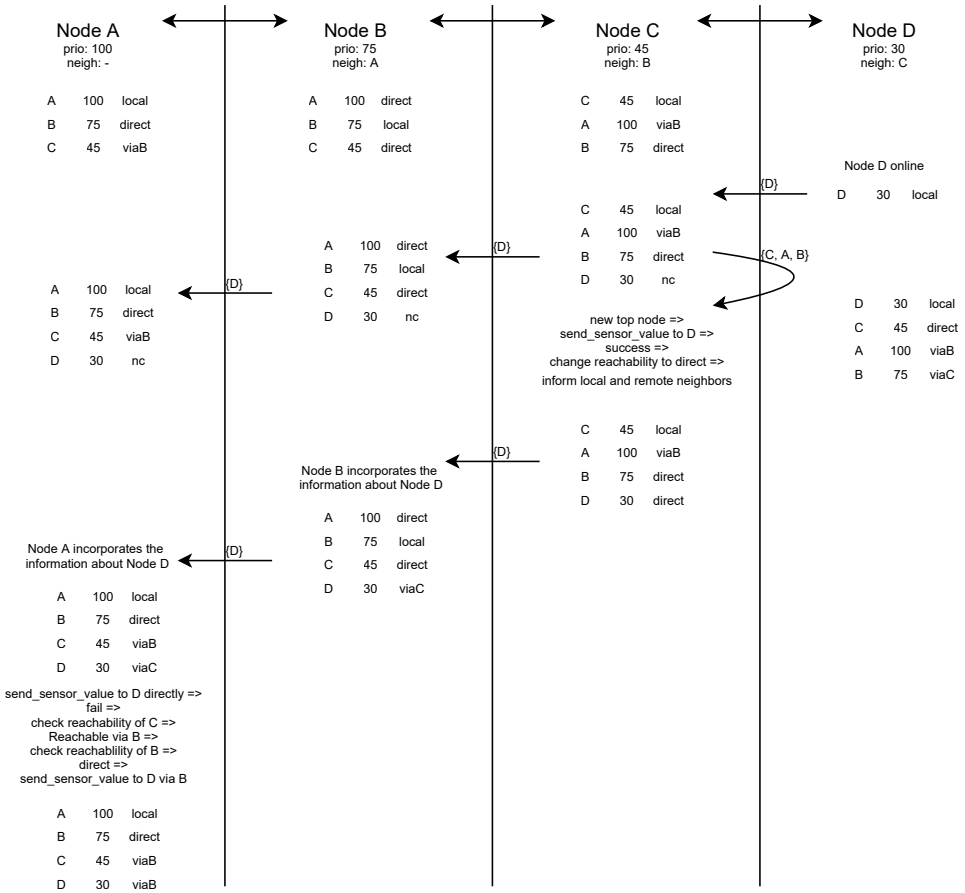
Short after Node A informed Node B about the unreachability of Node C it learns that Node B is able to contact Node C directly. So it resets the priority value to the original one and informs its neighbors. This value propagates through the whole network and in the end all nodes have agreed on Node C being the collector.

The process of Node D joining the network is similar to the previously described process of Node C joining, so it is not described in the same detail.

A notable difference is the behavior of Node A after learning that it can reach

[4]In this diagram this is illustrated by adding 2000 to the priority value because a 128bit integer with the most significant bit set to 1 would not really fit nicely in that diagram

Figure 2.13: Node D joins

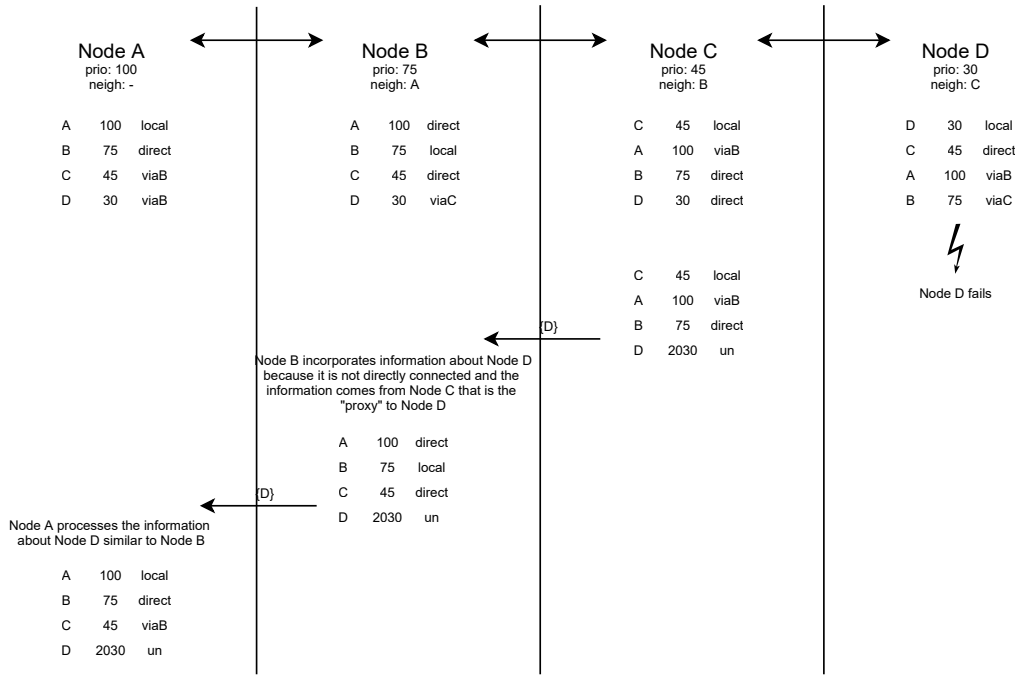


Node D via Node C. First the node tries to contact Node D directly, this fails because it can only reach Node B. But before contacting Node D via Node C it looks up the reachability information for Node C in its node list. Node C is also not reachable directly but via Node B. So the next step is to look up the reachability of Node B. This node is directly reachable, so it will send the sensor values destined for Node D to Node B.

In case Node D fails Node C will recognize it, update the reachability and priority and send this information to its neighbors. For Node B, Node D can be reached via Node C. Now Node C informs Node B of the failing Node D thus Node B also updates reachability and priority in turn. The same applies to the information about the failure that Node B sends to Node A.

To complete this first example, the last diagram, figure 2.15, shows the rejoin-

Figure 2.14: Node D fails



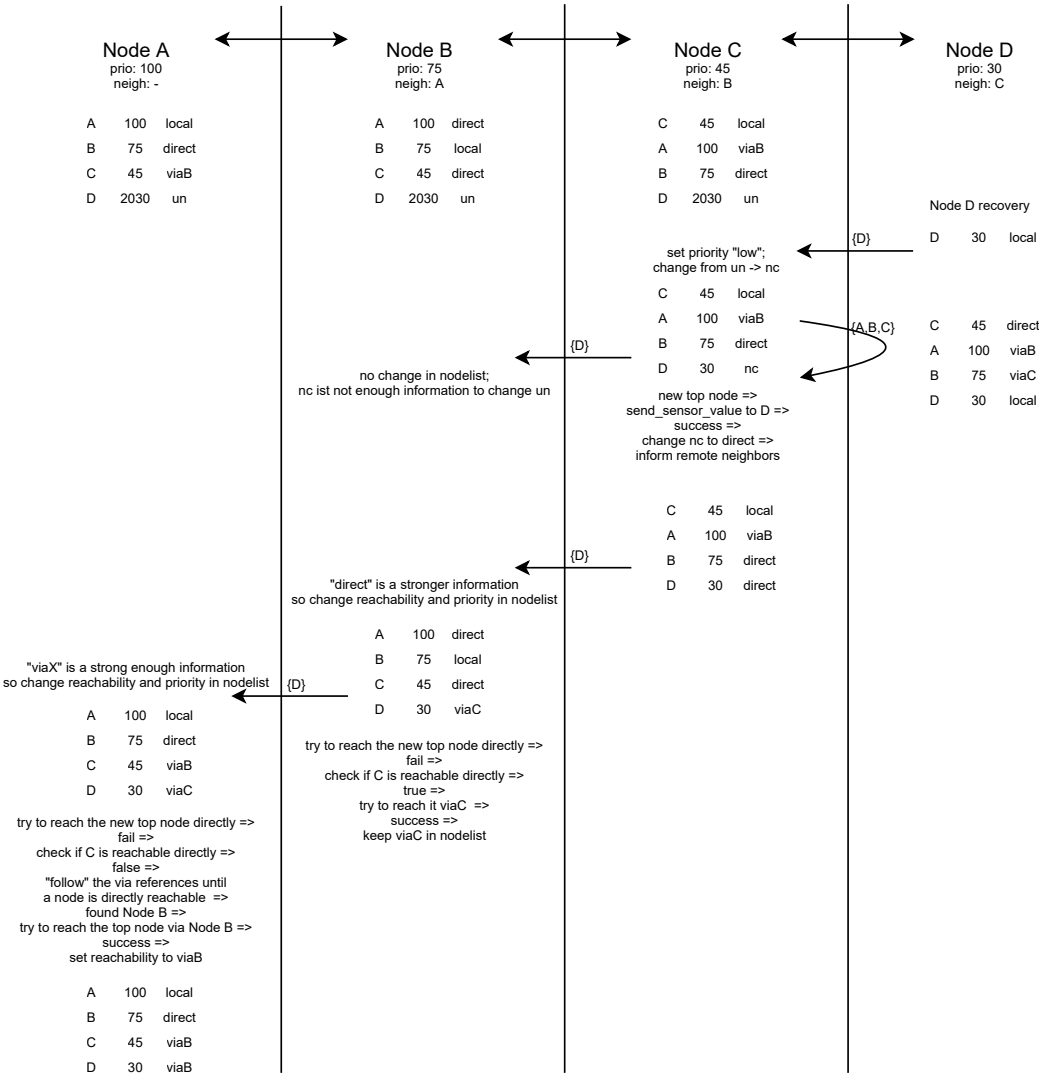
ing of Node D less than 10 minutes after its fail, so the information about the node is still in the node list. If it was more than 10 minutes ago Node D failed, it would be the same scenario as joining the network for the first time.

The examples discussed so far didn't really require the priority value updating in case a node gets unreachable. This mechanism is only needed in case a node can be reached by other nodes, but this node is unable to reach other nodes of the network. The following example shows, similar to the first one, a four node network building up and the collector node failing and recovering. The main difference is that now Node D can only reach Node C but not the other way round. The same applies to the nodes C, B and A similarly. This is expressed by the arrows at the top pointing in only one direction.

Figure 2.16 shows the first step of the buildup of the network where Node B joins Node A to form a 2-node network.

This small network already shows the necessity of dynamically adapting the priority value. The relay and proxy mechanism does not help in such situations,

Figure 2.15: Node D rejoins

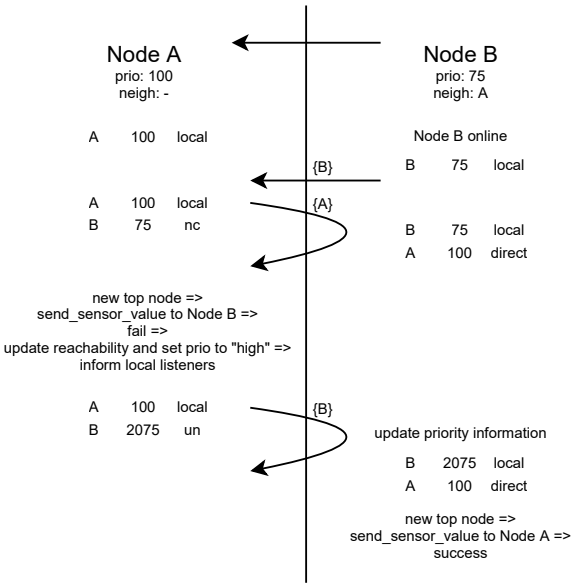


because the two nodes won't be able to agree on a collector node.

In the case shown in figure 2.16 Node A receives the information about Node B because Node B is able to contact Node A. Node A then tries to contact Node B, which fails. This leads to Node A increasing the priority value and informing its neighbor B. Node B updates the priority accordingly and both nodes agreed on Node A being the collector node.

The joining of Node C to the network follows a similar pattern. Now both

Figure 2.16: Network buildup where Node B can reach Node A, but not vice versa



mechanisms, the dynamically updated priority value and the relaying of messages by the neighbor nodes, that enable the network to function in such asymmetric situations work together. The dynamic update of the priority value alone would not be sufficient to build a working network in such a situation with three nodes. This is because Node C is unable to contact Node A directly and would set its reachability to “unreachable” and the priority value to “high”. This in the end would lead to a situation where every node in every node list has a higher priority value, which in turn would give the same priority ranking as in the beginning.

The diagram in figure 2.17 shows Node C joining the network. Finally, the three nodes agree on Node A as being the collector node and Node C sends its sensor data to Node A via Node B.

The process of Node D joining the network is similar and therefore not covered in a separate diagram.

Figure 2.18 shows the node lists of each node after Node D successfully joined the network.

If Node A fails, Node Bs node list only contains entries with a raised priority value and the lowest priority is the one of Node D. But this node is marked as unreachable. To solve that, Node B must reset the priority value to the original

Figure 2.17: Node C joins

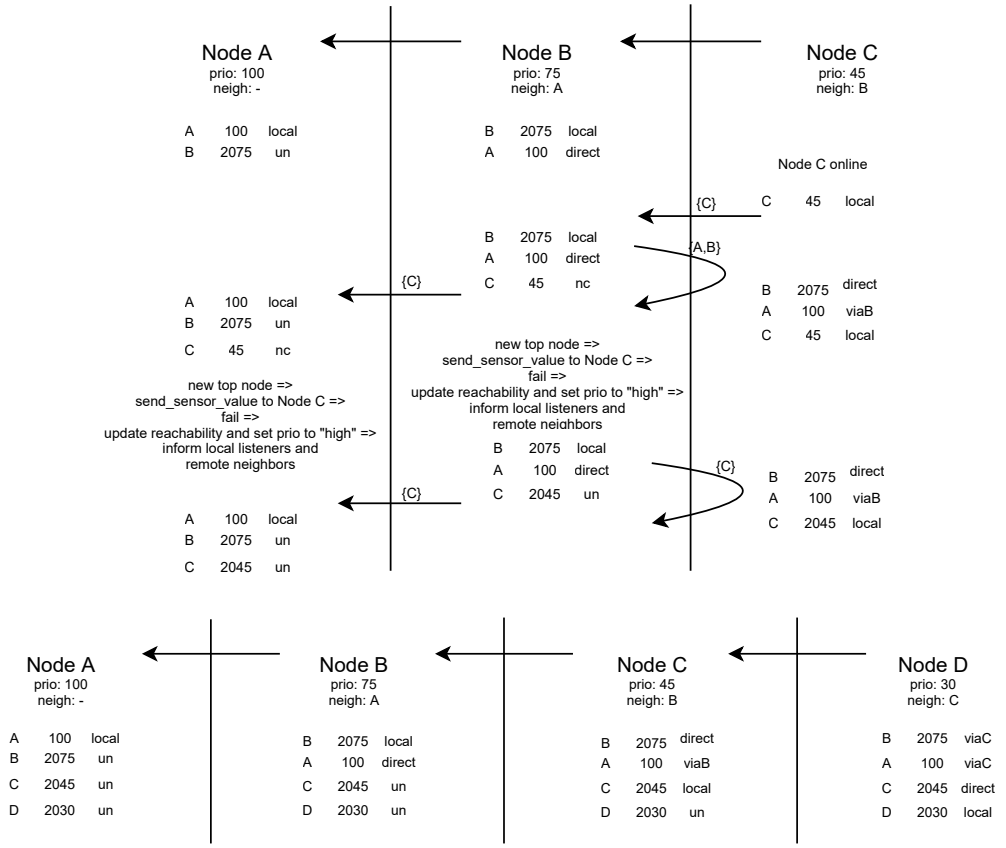


Figure 2.18: Network after Node D joined

value of all nodes in its list that are not marked as unreachable and inform its neighbors.

In this case Node B only has to reset its own priority to 75 and inform Node C. Node C then in turn informs Node D. This leads to Node B being the new collector node.

2.2 Configuration

The non-functional requirements demand a minimal configuration effort. Therefore, there are not many configuration parameters and all of them are optional. Although this might not always be useful, depending on what the node should do (joining a network or creating a new one). The main configuration is done via a configuration file, whereas logging can be configured via environment variables. The configuration file is expected to be called "`rssd.conf`" and must reside in the home-directory of the user running the program. If there is an error reading the file, e.g. the file is not found, not readable or alike the node will start with default values and generated values. This will be laid out in detail explaining all configuration parameters.

In case of a failure reading the configuration file the node will not be able to participate in an existing network but start a new network. Configuration values that are generated upon this failure will be logged, so that new nodes are able to join the newly created network. Of course the nodes wanting to join the network still need to have a copy of the TLS-certificate of the node they want to connect to.

For debugging and troubleshooting purposes the logging verbosity can be controlled via environment variables. The implementation uses the Rust crate `pretty_env_logger`. This crate uses the `RUST_LOG` environment variable to control logging[2]. Because this crate is used without modification, the best way to inform about the logging configuration is to have a look at the official documentation at `https://docs.rs/env_logger/0.8.3/env_logger/`.

2.2.1 Configuration File Format and Parameters

The configuration file format chosen for this project is TOML. TOML is very similar to the INI file format and aims to be a minimal configuration file format that's easy to read and to parse into data structures[20]. Furthermore, it is well-supported by the Rust programming language.

The configuration file is divided into 2 sections, called `nodeconf` and `contacts`. The sections are identified by square brackets. Each section has its

own set of possible configuration paramters.

A sample configuration file, showing all possible configuration parameters is presented below. The meaning of each parameter will be explained after the sample configuration.

Listing 2.1: sample configuration file

```
[nodeconf]
broker_port = 8883
onion_url = "tzfg35dbdmtida12.onion"
priority = 123
broker_password = "malaria"
sensor_interval = 1500

[contacts]

[contacts.1]
broker_port = 8883
onion_url = "b37zrcfebtma7ids.onion"
iplist = [ "192.168.142.51", "fe80::2da4:3456:9876:1", "10.12.23.45" ]
ca_cert = /path/to/certfile
broker_password = "barbara"

[contacts.2]
broker_port = 8883
onion_url = "c37zrcfebtma7ids.onion"
iplist = [ "192.168.142.52", "fe80::2da4:3456:9876:2", "10.12.23.46" ]
ca_cert = /path/to/certfile2
broker_password = "barbara2"
```

The `nodeconf` section provides configuration parameters for the node itself. The section as a whole is completely optional, as well as all parameters in it. Its parameters are:

- `broker_port`: Tells the MQTT broker on which port to listen on. The default value is 8883 and is applied if this parameter is not given.
- `onion_url`: This is the URL of the local onion service through which this node can be reached over the tor network. If this parameter is not given the program tries to read the file `$HOME/tor-data/hs/hostname`. If this also fails, the program will stop, because every node needs to know its own onion address.
- `broker_password`: This is the shared secret needed to connect to the lo-

cal broker. If this parameter is not present a shared secret will be generated, and shown in the log, so that other nodes are able to contact the broker on this node. Because this secret gets generated every time a node starts up if the configuration parameter is missing, it is a good idea to set this parameter, otherwise all other nodes connected to this node will have to reconnect and be reconfigured every time this node restarts with the newly created shared secret.

- `priority`: This parameter is for testing purposes only. It allows constructing predefined networks. Usually this parameter is generated as outlined in section 2.3.1.

- `sensor_interval`: Determines the interval in which the sensor-data is sent to the collector node. The interval is given in milliseconds and is an unsigned 64-bit integer. The default if no number is given is 250ms.

The `contacts` section contains information needed to contact other nodes and join existing networks. Like the `nodeconf` section explained above this section is technically completely optional too. If the node should actively join an existing network this section must exist and at least one contact must be specified. The contacts section can be followed by one or more `contacts.X` sections. The X is an arbitrary string. The only constraint on this string is that it has to be unique in the config file, because it will be parsed into a "HashSet", that uses this string as key and the following configuration parameters as the corresponding value.

The parameters in a `contacts.X` section refer to the remote node(s) that act(s) as "entry point" to the sensor network. The `broker_port`, `onion_url` and `broker_password` have a similar meaning as in the previous configuration section. In this section they are parameters used to connect to remote nodes rather than configuring local services. There are two additional parameters not present in the `nodeconf` section:

- `iplist`: An array of IP-addresses where the neighbor can be reached. This can be a mix of IPv4 and IPv6 addresses.

- `ca_cert`: The absolute path to the file containing the certificate the remote

broker users for TLS. The certificate file must be in PEM format as defined in RFC-7468 [5].

The parameters `ca_cert` and `broker_password` are mandatory if a section `contacts.X` exists. This is because the MQTT-connection to other brokers is done using TLS unexceptionally, and the authentication is always mandatory.

The parameters `onion_url` and `iplist` are both optional, but at least one of them is required to be able to contact the neighbor. If both are present, the neighbor will be contacted first by IP, and if not successful via Tor. If none of the two is present, a warning will be logged and the neighbor will not be contacted, since it is not possible. In case the only `contacts.X` section contains neither of these parameters, the node will start a new network, because there is no neighbor that can be contacted and therefore, the node is not able to join an existing network.

2.3 Collecting Sensor Data

As mentioned in the introduction, the focus of the implementation is on network operations. The sensor values sent to the broker on the collector node are just random floating point values in the range [17.0;45.0] representing a temperature sensor. As already explained in the section on configuration, the default interval for sending sensor data is 250ms, but this value is configurable.

To collect all sensor data on the same broker, a dedicated collector node must be determined. Furthermore, a MQTT-topic schema is developed that allows the sensor data to be evaluated in a meaningful way.

The details of determining the collector node and the MQTT-topic schema are laid out in the following sections.

2.3.1 Determining the Collector-Node

Unlike other distributed systems, no leader election in the true sense is performed in this sensor network. Other protocols like Raft [19] or Paxos [14] based protocols use multiple iterations to elect a leader. This is not done in this network.

Determining the collector-node is simply based on a precalculated priority value. This value is calculated by every node for itself. The node with the lowest priority value will be the collector-node.

The priority value takes several metrics into account:

- The number of global IPv4 or IPv6 unicast addresses.
- The time the node went online.
- The amount of available RAM when the node goes online.
- A random number as tie-breaker.

The priority is encoded as an unsigned 128-Bit integer where the most significant bit is used to dynamically update the priority value if a node gets unreachable or reachable again as described in section 2.1.2. The two most important metrics affecting the priority are the number of global IP addresses and the time the node

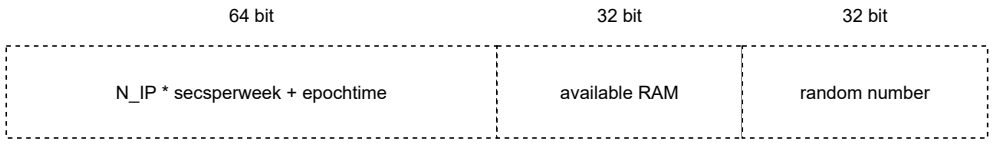
went online. Those two numbers make up the most significant 64-Bit of the priority. The amount of available RAM accounts for next 32-Bit of priority value. The 32 least significant Bits are randomly generated by a cryptographically secure RNG to raise the chance of a unique priority. So the amount of RAM is only important in case of two or more nodes coming online at the same second with the same amount of global unicast addresses.

The whole priority value is calculated as follows:[5]

```
!((ipcount * secsperweek + !epochtime) << 64
  + available_ram << 32
  + random_int)
```

Figure 2.19 illustrates the anatomy of the priority value.

Figure 2.19: Anatomy of the priority value



The number of global IPv4 or IPv6 unicast addresses serves as heuristic for the *reachability* of a node. Where reachability means the node is directly reachable without the need for traversing NAT or blocked by some network-filtering device, like a firewall. The sole presence of a global IP address is neither necessary nor sufficient to guarantee direct reachability. It is not sufficient because new incoming connections could still be blocked by firewalls. On the other hand it is not necessary because some NAT mechanisms could be in place to enable direct network communication with the node despite only having "private" IP addresses. It is not possible for a node to determine its reachability on its own, so this heuristic is simple but sufficiently accurate. At a first glance the "NAT Behavior Discovery Using Session Traversal Utilities for NAT (STUN)" as defined in RFC5780 looks promising for determining the reachability of a node because the abstract promises to discover "the presence and current behavior of NATs and

[5]The ! means bitwise inversion in Rust.

firewalls between the STUN client and the STUN server." [15]. But having a closer look at the protocol reveals that it does not detect a stateful packetfilter blocking requests to the node.

Since there is no protocol, and therefore no public services that can be used, that allows a reliable determination of a nodes reachability, it would require extra infrastructure to run a custom service. Unfortunately it's not possible to state a probability for a node being put behind a 1:1 NAT or getting a global unicast IP address but still being restricted by a firewall. Therefore, the number of global unicast IP addresses is a viable heuristic.

The time the node went online is the POSIX time, as it is defined in [10], captured at the moment the program starts.

The two numbers are combined by multiplying the number of IP addresses by the number of seconds per week and adding the bitwise inverted time the node went online. $x = N_{IP} * secsperweek + !epochtime$

Thus, one IP address is equivalent to one week of uptime. So a node without a global IP address, that is online for more than one week has a lower priority value than one with one global IP address that came online more than one week later. The following paragraphs will give a concrete example how the priority value, as a combination of time coming online and number of IP addresses, is generated.

The generation of the priority value, specifically highlighting the interaction of the number of global unicast IP addresses and the time a node comes online, will now be illustrated by an example using five nodes. The example uses symbolic values instead of actual ones just to illustrate the working of the algorithm.

Table 2.3: Priority value depending on the amount of IPs and time of coming online only.

	Node A	Node B	Node C	Node D	Node E
# global unicast IPs	0	0	1	1	5
Online since	T	T+1s	T+1w	T+1w+1s	T+4w
Priority value[1]	100	99	100	99	101

[1] Derived from the number of global unicast IPs and the time the node came online. Higher value means higher priority, since the whole value combined with the memory metric and the random number will be bitwise inverted

Table 2.3 illustrates the combination of the number of global unicast IPs and the time a node comes online to the priority value. From this it is clear that node C that came online one week after node A, but has one global unicast IP address, gets the same base priority value. Whereas node D, even with its global unicast IP gets a lower priority value as nodes A and C because it comes online over a week later (one second after Node C). The node with the highest priority in this case will be Node E, even though it comes online 4 weeks after Node A, but has 5 global unicast IP addresses configured.

This shows how the priority value reflects the reachability (number of global unicast IP addresses) and the stability (time the node came online) of a node.

The amount of available RAM when the node goes online is given in kilobytes. The documentation of the library used for obtaining the available memory states: "Generally, "free" memory refers to unallocated memory whereas "available" memory refers to memory that is available for (re)use." [6] This means that this number includes memory that is already mapped for caches etc. but is generally available for applications.

A random number as tie-breaker is used in case two nodes have the same amount of global unicast IP addresses, came online in the same second and have the same amount of available RAM, which isn't too likely by itself. The random number is a 32-Bit unsigned integer, generated in a cryptographically secure way, so it is extremely unlikely to generate conflicting priority values. A 32-Bit random value means that 2932 numbers must be generated to exceed a collision probability of 0.1%.[6] The chance of two nodes having the same priority value is extremely low.

Nevertheless, there is a further measure to determine a unique collector node that is already described in section 2.1.1. The onion URL[7] under which the node is reachable acts as ultimate decision criterion if the calculated priority values of two or more nodes are the same. In that case the priority is given by the lexicographical

[6]This number can be calculated with the following formula: $n \approx \lceil \sqrt{-2 * \ln(1 - p_m) * N} \rceil$, where p_m is the desired probability, in our case 0.001, of a match and N is the amount of possible random numbers, in the case of 32-Bit numbers 2^{32}.

[7]The onion URL is derived from the 256 bit Tor public key that is generated based on a cryptographically secure random number. Thus, chances are extremely high for it to be a unique value in our networks.

order of the onion service hostname.

2.3.2 MQTT-Topic Schema for collecting Sensor-Data

To be able to collect the sensor data in a useful way, it's necessary to define a topic schema all clients use, to publish their data. Usually an MQTT message just contains a small string, often representing a numerical value, and the topic encodes metainformation. Such a topic could look like this: `house1/room-101/ceiling/temp/celsius`. This only works if you know all publishing sensors in advance, so you can subscribe to every single topic.

If you have a network of nodes and don't know what sensors will provide data it's not possible to collect all sensor data by subscribing to every single topic a node publishes the data, because the topics are not known in advance. It is still possible though to collect all sensor data using so-called wildcard subscriptions.

These wildcards are defined in the MQTT specification by the Organization for the Advancement of Structured Information Standards (OASIS). This specification defines two types of wildcards. The "Single-level wildcard" represented by the "+" sign, and the "Multi-level wildcard", represented by the "#" sign[9].

Here are some examples how to filter data using these wildcards assuming the following topic schema: `sensors/<nodeID>/<measurementtype>`.

- `sensors/#` subscribes to all child topics of the `sensors` topic.
- `sensors/ID/+` subscribes to all topics one level below `sensors/ID` topic. In our example this would mean all measurements of a node.
- `sensors/+/TEMP` subscribes to the `TEMP` topic of all nodes.

A major problem with those simple filters is, that metadata encoded in the topic is lost. This means that the meta information must be transported by other means. So if you subscribe to the topic `sensors/ID/+` and the node has different sensors connected it's not possible to say if a value represents a temperature or relative humidity for example.

There are various ways in which this problem could be solved, but this is beyond the scope of this project. In the section "outlook" three possible solutions are outlined.

3. Security

One of the non-functional requirements is security. The security requirements demand that the application must support the three main security objectives confidentiality, integrity and availability. Furthermore, authentication is vital to be able to support confidentiality and integrity. How availability is maintained, has already been described in detail in section 2.1.2.

3.1 Threat Model and Adversary Model

Before implementing security measures, it should be made clear against what they are intended to help and in which cases the security measures will only have a limited effect or will even be ineffective. The following sections provide an overview of possible threats, followed by sections that explain the measures taken to meet the threats.

General Threats

General threats are threats that are not specific for the sensor network. These threats are possibly many. There are a lot of possible vulnerabilities in an operating system and its network stack, or users that run different, possible vulnerable, software on a sensor node. These threats can't be realistically dealt with, so they are not considered further.

There are still enough general threats that can be dealt with. The network relies on the `mosquitto` MQTT broker and the Tor daemon. These two could be vulnerable, but this can be mitigated, for example by using some form of virtualization or simply following the principle of least privilege or even combine those

measures. Other threats are programming errors, including logical mistakes, improper error handling and of course memory corruption issues.

How all the above threats are addressed is explained in section 3.2.

Information Disclosure or Alteration

Since all communication is done over an insecure medium, it may be possible for an attacker to read sensitive information, like the sensor values or the network management information, or even alter the data in transit. Even more subtle attacks, like inserting or removing sensor data and network control information are possible. This can lead to a total malfunction of the network or, even worse, to subtle changes in the network structure or sensor data that are difficult to detect. In case the sensor-data is used to operate some actuator this may even be catastrophic.

This kind of threats are addressed using cryptography to ensure confidentiality, integrity and authentication. How this is implemented in detail is explained in the respective sections.

Despite all cryptography, which makes all the above attacks impossible, it is still possible to maliciously prevent a node from sending messages or joining the network. This is like a network problem and therefore next to impossible to detect.

Rogue Sensor/Node

If a node is able to participate in the network with malicious intent, it is called a rogue node. Authentication should prevent such a node from participating, but it could be still possible for dedicated attackers to introduce a rogue node to the network.

This can be done either by stealing authentication information, or by somehow taking control of a node (hacking it). Another way may be to gain trust by a party using social engineering.

Such a rogue node scenario is not addressed by the security measures implemented.

DoS and brute force on password

It may be possible to perform a DoS attack on the network, leveraging the fact that authentication is done at the application layer. Even unsuccessful authentication needs resources, and extra resources are needed for TLS, which may be a way to exhaust them especially on resource constrained devices. The MQTT broker used in this project is the `mosquitto` MQTT broker. This broker does not offer protection against brute force password attacks nor against previously described versions of DoS attacks.

Measures against these kinds of attack could be implemented using software like `fail2ban` that evaluates the applications (mosquitto) log file and based on that decides whom to block for how long. Other possible measures are using OS specific mechanisms for controlling the number of connections or implement a custom MQTT broker for this project that includes measures against DoS and brute force attacks.

3.2 Security during Development / Design

Google, in a report about the security in the chromium project, states that 70% of "serious security bugs are memory safety problems"[4]. The same figure is given by Microsoft, that "70 percent of all the vulnerabilities in Microsoft products addressed through a security update each year are memory safety issues"[12].

To address these memory corruption issues, Rust was used as programming language. It guarantees memory-safety and thread-safety, while being a natively compiled language without a garbage collector or a runtime, thus being a good choice to run even on small embedded devices[1].

In order to prevent or minimize programming errors, unit testing and integration tests have been implemented. Section 4 provides further information about integration tests.

Furthermore, the program drops privileges if ran as user `root`. To be able to drop privileges a user named `rssd` must exist. If ran as unprivileged user the program does not drop privileges and just runs under the user it got started. The

[1] see `https://www.rust-lang.org/`

Tor and MQTT daemon are also started with the same privileges as the sensor-network program.

3.3 Confidentiality and Integrity

The MQTT broker used in this project, the `mosquitto` MQTT broker, supports MQTT over TLS. This is used to encrypt the communication between the client and the broker and perform the integrity checks. The TLS maximum version supported by the client library used and by the broker software is 1.2 and is enforced by configuration on the broker and client to prevent downgrade attacks. The MQTT client library uses OpenSSL for cryptographic routines. To prevent OpenSSL from using a cipher-suite supported by TLSv1.2 only providing authentication and integrity but no encryption, the cipher suite is specified as follows: `'TLSv1.2:!COMPLEMENTOFALL'`[2].

3.4 Authentication

TLS relies on X.509 certificates for establishing trust as the basis of authentication. This standard defines hierarchy of certificate-authorities, that results in a centralistic system of CAs. This on the one hand collides with the non-functional requirements defined in 1.1.2, and on the other hand it does not really fit the peer-to-peer approach used in this network. Section 5.1.1 gives some ideas how this problem could be solved.

The implementation relies on TLS because all components already support it, even though not being the perfect choice for this particular network.

The certificates used by the broker are all self-signed certificates. In order to verify the authenticity of the broker the client must know the brokers certificate in advance.

Because the broker can't know the clients that will connect to it, clients authenticate with a shared secret. This is also supported by the broker and client library. The whole mutual authentication process is implemented as follows:

[2]The OpenSSL version used is 1.1.1d. A documentation of how the ciphers are specified can be found at `https://www.openssl.org/docs/man1.1.1/man1/ciphers.html`

1. The client knows the shared secret and the certificate of the broker, of all its configured neighbors, in advance by configuration[3].

2. The client contacts its configured neighbors, verifying the neighbors certificate with the one it has configured. In case the verification fails, the connection is aborted.

3. In case the verification is ok, the client sends its configured shared secret to the broker, over an already encrypted connection.

4. If the shared secret is ok the broker accepts the connection and information about the network can be exchanged. Else the connection is terminated by the broker and the node is unable to participate in the network.

5. The information about the network also contains the shared secrets and the certificates of all brokers in the network[4].

As soon as a node successfully authenticated to a neighbor, it has access to the whole network and all other nodes in the network will trust that node. This means that the configured neighbors act as kind of gatekeepers to the network. As soon as a node is accepted by such a gatekeeper it will be accepted by the whole network.

[3] see section 2.2

[4] see the node struct in section 2.1.1

4. Tests and Measurements

In this chapter, the tests and measurements carried out in selected networks and their results are presented. The tests in section 4.1 are intended to check whether the functional requirements defined in section 1.1 have actually been met and verify the robustness and resilience of the network. The tests in section 4.2 focus on security measures.

4.1 Testing Network Stability and Resilience

In this section, the networks are tested to see if they can form despite underlying difficulties, such as NAT or asymmetric reachability conditions. Furthermore, the time it takes for all nodes to agree on a collector node is measured.

The tests are all done using virtual machines in VirtualBox, version 6.1. All tests, except those in section 4.2 which use a more memory constrained system, use debian 10.11[1] as the operating system on the nodes. The respective network configuration is explained in the specific test scenarios.

To find out which node sends its sensor data to which one, a custom MQTT client was written, which connects to all nodes in the network at the same time, logging the information on which node it received a sensor value and the sender of that message together with a timestamp. This makes it possible to see whether all nodes send to the same collector node and how long it took from starting a node to all nodes agreeing on the same collector. All following tests ran script-controlled and were performed 100 times. This makes the tests more meaningful in terms of stability but also convergence times.

[1] The current stable version at the time of the tests

The convergence time is the time it takes all nodes to agree on the same collector node. This is measured as the time difference between the moment a new collector must be chosen and the point in time, where all nodes send their sensor values to the new collector. The moment a new collector must be chosen can either be the moment it comes online or the point in time it failed. The sensor data send interval in the tests is 1s, the measured convergence time is therefore at most 1s longer than it could theoretically be.

All tests in this section use 4 nodes. The initial priority values and the neighbor nodes are shown in the following table. A neighbor node means that a node establishes a connection to this neighbor node in order to exchange control messages concerning the sensor network and has nothing to do with a neighborhood in the network in the sense of the lower OSI or TCP/IP layers. The logical topology of a chain is formed. This topology was chosen because it is the most fragile in terms of stability.

Table 4.1: Node configuration for testing

	Node A	Node B	Node C	Node D
priority	7	5	3	2
neighbor nodes	-	A	B	C

After every step in each test scenario, which is either a node joining the network or a simulated node or network failure, the test evaluates if the collector node is the expected one. In case the observed collector node, or the collector nodes in case a split is expected, does not meet the expectation it will be considered an error, even if a functional network with a different collector node than expected emerges. The errors that occurred are counted and at the end of the tests their total number is output. If errors occur, they are explained in the sections on the respective scenarios.

Scenario 1

This scenario is for testing the network without restrictions in the underlying TCP/IP network infrastructure. All nodes are simply on the same LAN, where they all can reach each other. The steps in this test are:

1. Build the network: The starting sequence is A, C, D, B at 10 second intervals. This means the network starts in a split state, since no information about the network can pass from A to C, because this would need Node B forwarding messages between Node A and C. After Node B is online all nodes are expected to send their sensor data to Node D. The following table 4.2 shows the four network buildup stop in more detail. A dash between two nodes indicates a control connection and a frame around a node name indicates that this node is a collector node.

Table 4.2: The 4-step network buildup

Action	Network(s)
Node A started	[A]
Node C started	[A] [C]
Node D started	[A] C–[D]
Node B started	A–B–C–[D]

2. Simulate failure of collector node: The nodes D, C and B are stopped in exactly this order, again at 10 second intervals. The nodes are expected to change the collector node as the current collector fails. Table 4.3 shows the simulated failure steps and the expected networks after these failures.

Table 4.3: Node failures and expected network

Action	Network(s)
Node D fails	A–B–[C]
Node C fails	A–[B]
Node B fails	[A]

3. Recovery of the previously failed nodes: The nodes recover in the order C, D and B. This again leads to a split network that should be resolved as soon as Node B comes online. The behavior expected is the same as illustrated in table 4.2.

Table 4.4 shows the measurements of the convergence times for each step in the tests. In total 100 tests have been performed. This makes 100 relevant convergence events in step 1. The relevant convergence event in step 1 is when Node B joins the network and the split gets resolved. The same applies to step 3. The formula for calculating the convergence time is

$$max(T_{AD} - T_{startB}, T_{BD} - T_{startB})$$

where T_{AD} and T_{BD} are the times, where Node A or Node B send their sensor value to Node D for the first time and T_{startB} is the time when Node B comes online. For the calculation the maximum time difference is taken, because after this period all nodes in the network agree on the same collector node.

In step 2 there are 3 convergence events relevant for this measurement. Each time a node goes offline is such an event. Therefore, there are 300 such events overall. The convergence time formula is very similar to the one used in step 1 and 3

$$max(T_A - T_{stopTop}, T_B - T_{stopTop}, T_C - T_{stopTop})$$

where $T_{stopTop}$ is the time the collector node gets stopped, and $T_{[node-name]}$ is the time the respective node sends its sensor data to the new collector node. If there are only 3 nodes left, only 2 nodes are used for calculation.

The $\overline{x}$ column shows the average convergence time for each step and the s column the corresponding standard deviation.

Table 4.4: Convergence time scenario 1

	convergence time [ms]	
$N = 100$	$\overline{x}$	s
step 1	5013	1769
step 2	3901	667
step 3	3947	1074

In this scenario six errors occurred, all in the same test run. All these six errors had the same cause, it was Node A failing because of a "segmentation fault" in the MQTT-library[2]. The rest of the network without Node A worked as expected

[2]The library is actually written in C and just a Rust wrapper around the C code

and formed a functioning network.

This first test scenario shows the basic functionality of the network. It shows the adequate reaction to a failure of the collector node. Furthermore, it demonstrates the ability of the network to reunite after a split.

The convergence times reveal that a reunion (step 3) is significantly faster than the initial union of the network (step 1). This is because all nodes are already known to Node A. It especially knows which connection it previously used and does not need to try all other connection possibilities.

Scenario 2

This scenario and the tests performed are similar to the first one. The main difference is, that the nodes are now all in separate LANs and all "hidden" behind a NAT router. This makes it necessary to communicate via the Tor network.

The expected resulting network state is the same as in scenario 1. The main expected difference is the time it takes to contact the neighbors via the Tor network and therefore the convergence times should be higher than in the first scenario. So, in contrast to scenario 1, the nodes are not started and stopped in 10-second intervals, but in 90-second intervals. Secondly more errors are expected, because the connection over the Tor network might take longer than the connection timeouts set in the application. This could result in a different but connected and functioning network than expected or lead to a split network, which both would be considered an error.

Table 4.5: Convergence time scenario 2

	convergence time	
$N = 100$	$\overline{x}$	s
step 1	42007	9145
step 2	23441	8697
step 3	46187	8898

Convergence times are approximately almost a factor of 10 higher than in scenario 1.

Of the 100 tests performed, deviations from expectation were found in 17 cases. They all had the same root cause. It was always a connection timeout to the

Tor network that caused the errors. Depending on which connection failed and in which test step this happened two different error manifestations with two different characteristics each have been observed.

- Functioning network, but different from expectation. This happens if the connection to the Tor network failed when trying to send the sensor data, but all neighbor connections in the chain are intact. In that case one of the nodes was not able to send its sensor data to the collector node, so it marked it as unreachable, set the priority value to "high", as described in section 2.1.2, and informed the neighbors. This led to a different collector node than expected, but is still a functioning network, because all neighbor connections are intact. In total this kind of error occurred 9 times. The two different characteristics are:
  - Temporary failure: This means the error occurs in one of the intermediate steps and does not propagate further, so that a network is created again in the following step that meets the expectations. The reason for the recovery is either a new node joining the network with a lower priority value and the connection to that node succeeds, or the expected collector node leaves the network and therefore another node is the expected collector and the connection to that new expected collector succeeds. In total this temporary failure was observed 8 times. The recovery occurred 5 times on a new node joining and 3 times because the expected collector went down.
  - Permanent failure: This means that the resulting network keeps being different as expected. This happened only once, because the error was detected in the last step of the test. Therefore, it was categorized as a permanent failure, because it lasted until the end of the test.
- Split networks: This means that at least two distinct networks emerged, although it should be possible to form a single network. This happened because a network control connection to one of the neighbors failed. Here the same two characteristics can be observed. In total 8 times a split network was detected.

- Temporary failure: This type of error was observed 7 times. Two different events led to the observation matching the expectation again. The first one was caused by the connection to the neighbor succeeding after the test interval timeout of 90s, the time between two test steps. In five cases it took more than these 90s to establish a successful network control connection to a neighboring node. In two cases the expectations where met again because the collector node went down and in subsequent test steps the control connection succeeded in time.
- Permanent failure: In contrast to the permanent failure mentioned above, the permanent failure in this case persisted over several test steps. Such a permanent failure was only detected once.

Scenario 3

This test scenario resembles the first example given in section 2.1.2 where the nodes are only able to reach their neighbor and vice versa. In this scenario the Tor network access was prevented, because it is impossible to only allow specific connections via Tor. The nodes are started in the order A, B, C and D. After D successfully joined the network (step 1) the expected collector node, Node D, gets stopped (step 2). Finally, Node D rejoins the network (step 3)

The network restrictions have been realized using the operating systems' firewall infrastructure.

The example already mentioned also sets the expectations. After the first step in the test, the "buildup" phase, and at the end of the test, Node D should be the collector node and the expected relevant parts of the node list are shown in the table 4.6.

The actual node lists of every node after step 1, taken from the debug output of Node A, is given below. The debug output node list was reduced to the most relevant parts.

Listing 4.1: Debug output of node list taken from Node A after step 1

```
node_list: [
Node {
    onion_url: "hl7ecb7runfwsik6oqwbd65fh5tjxthoa2bpywrs6fgtp74zlj5isaqd.onion",
    priority: Some(2),
```

Table 4.6: Expected node list after test step 1

Node	Node list entry	Priority	Reachable
Node A	Node A	7	local
	Node B	5	direct
	Node C	3	viaB
	Node D	2	viaB
Node B	Node A	7	direct
	Node B	5	local
	Node C	3	direct
	Node D	2	viaC
Node C	Node A	7	viaB
	Node B	5	direct
	Node C	3	local
	Node D	2	direct
Node D	Node A	7	viaB
	Node B	5	viaC
	Node C	3	direct
	Node D	2	local

```
    connection: Some(Ip(192.168.56.32)),
    reachable:
      Some(Via("y5n7meyrmackkklfoficwgvirgfromp4gxusryvkmhxtiibcqw5mjoqd.onion")),
},
Node {
    onion_url: "uy3pfz726lj5dckmslhxfwwr5v7o2en7p7rec52xel2drtin5fkcxqyd.onion",
    priority: Some(3),
    connection: Some(Ip(192.168.56.32))
    reachable:
      Some(Via("y5n7meyrmackkklfoficwgvirgfromp4gxusryvkmhxtiibcqw5mjoqd.onion")),
},
Node {
    onion_url: "y5n7meyrmackkklfoficwgvirgfromp4gxusryvkmhxtiibcqw5mjoqd.onion",
    priority: Some(5),
    connection: Some(Ip(192.168.56.32)),
    reachable: Some(Direct),
},
Node {
    onion_url: "brylkh3qatb4u47xnp6t2pfrj3m65nkhacydbbtu4e77e2uk6uouwyad.onion",
    priority: Some(7),
    connection: Some(Ip(127.0.0.1,))
    reachable: Some(Local),
},]
```

The following table is a mapping of node names shown in table 4.6 to their

.onion and IP addresses.

Table 4.7: Node Name to .onion and IP address map

Node A	Node B	Node C	Node D
brylkh3…	y5n7mey…	uy3pfz7…	hl7ecb7…
192.168.56.31	192.168.56.32	192.168.56.33	192.168.56.34

This output corresponds well with the expected values shown in table 4.6. The `connection` value shows that all connections from Node A are made through Node B.

Table 4.8: Convergence time scenario 3

	convergence time	
$N = 100$	$\overline{x}$	s
step 1	4120	998
step 2	2611	720
step 3	3852	1389

No errors occurred in this scenario.

This test scenario demonstrates the networks' ability to function in a restricted network environment. In this scenario some nodes are unable to send their sensor data directly to the collector node, but need to send it via their neighbors that act as "proxy nodes".

Compared to the convergence times from scenario 1, the significantly shorter convergence time in step 2 is striking. This is because the nodes that are further away in the logical chain don't need to change their sensor data sending target that often. They do update their node lists, but they still send the sensor data to their neighbors because this is the only node they can send it to directly.

Scenario 4

This test should check the second example introduced in section 2.1.2, where the nodes are only able to reach their neighbors but this time only in one direction. This kind of restriction is actually quite common in IP networks protected by a firewall, where the devices in the protected network should be able to initiate

a connection to other devices on the internet but no machine from outside the protected network should be able to initiate a connection to a device inside the protected network.

Like in scenario 3 `iptables` was used to set the network restrictions in such a way, that Node D is allowed to initiate a connection to no other IP address than the one of Node C. Node C is only allowed to contact Node B and this in turn may only initiate a connection to Node A. This also means that the access to the Tor network was prevented for the same reason it was in scenario 3.

Although such a restricted chain of nodes is not quite a realistic scenario, this test aims to show the functioning of the network even in such extreme situations.

The test steps are a bit different in this example, because the test should show the behavior of the network in case of a collector node failure. The "buildup" phase is the same as in scenario 3. But the expected collector node after all 4 nodes are online is Node A, so the sequence the nodes are turned off is A, B and C (step 2).

In this phase the expected collector nodes are B after turning off A, C after B failing and finally Node D as the remaining node in the network after Node C went down.

The sequence the nodes are brought online again in step 3 is B, A and C. This is for testing a network split and reunite.

In this test scenario the nodes were started and stopped in a 40-second interval.

Table 4.9[3] shows the expected node lists after the "buildup" phase and at the end of the test, and the node list how it should look like after the collector node A failed.

The following listings show the debug output of the node list taken from Node A after step 1 and from Node B after Node A failed. Again, the debug output was reduced to the most relevant parts of the node list. The mapping of node names to the `onion_url` is the same as in test scenario 3 shown in table 4.7.

Listing 4.2: Debug output of node list taken from Node A after step 1

```
node_list: [
```

[3]Like in the example given in section 2.1.2, the "high" priority values in the table are built by adding 2000 to the initial priority value, because a 128bit integer with the most significant bit set to 1 would not really fit nicely.

Table 4.9: Expected node lists in “one way” scenario

Node	Node list entry	after steps 1 and 3 Priority	after steps 1 and 3 Reachable	after Node A failed Priority	after Node A failed Reachable
Node A	Node A	7	local		
	Node B	2005	un		
	Node C	2003	un		
	Node D	2002	un		
Node B	Node A	7	direct	2007	un
	Node B	2005	local	5	local
	Node C	2003	un	2003	un
	Node D	2002	un	2002	un
Node C	Node A	7	viaB	2007	un
	Node B	2005	direct	5	direct
	Node C	2003	local	2003	local
	Node D	2002	un	2002	un
Node D	Node A	7	viaC	2007	un
	Node B	2005	viaC	5	viaC
	Node C	2003	direct	2003	direct
	Node D	2002	local	2002	local


```
Node {
    onion_url: "brylkh3qatb4u47xnp6t2pfrj3m65nkhacydbbtu4e77e2uk6uouwyad.onion",
    priority: Some(7),
    connection: Some(Ip(127.0.0.1,))
    reachable: Some(Local),
},
Node {
    onion_url: "hl7ecb7runfwsik6oqwbd65fh5tjxthoa2bpywrs6fgtp74zlj5isaqd.onion",
    priority: Some(170141183460469231731687303715884105730),
    connection: None,
    reachable: Some(Unreachable(1645212452)),
},
Node {
    onion_url: "uy3pfz726lj5dckmslhxfwwr5v7o2en7p7rec52xel2drtin5fkcxqyd.onion",
    priority: Some(170141183460469231731687303715884105731),
    connection: None,
    reachable: Some(Unreachable(1645212451)),
},
Node {
    onion_url: "y5n7meyrmackkklfoficwgvirgfromp4gxusryvkmhxtiibcqw5mjoqd.onion",
    priority: Some(170141183460469231731687303715884105733),
    connection: None,
    reachable: Some(Unreachable(1645212456)),
},]
```

Listing 4.3: Debug output of node list taken from Node B after Node A failed

```
node_list: [
Node {
    onion_url: "y5n7meyrmackkklfoficwgvirgfromp4gxusryvkmhxtiibcqw5mjoqd.onion",
    priority: Some(5),
    connection: Some(Ip(127.0.0.1)),
    reachable: Some(Local),
},
Node {
    onion_url: "hl7ecb7runfwsik6oqwbd65fh5tjxthoa2bpywrs6fgtp74zlj5isaqd.onion",
    priority: Some(170141183460469231731687303715884105730),
    connection: None,
    reachable: Some(Unreachable(1645212452)),
},
Node {
    onion_url: "uy3pfz726lj5dckmslhxfwwr5v7o2en7p7rec52xel2drtin5fkcxqyd.onion",
    priority: Some(170141183460469231731687303715884105731),
    connection: None,
    reachable: Some(Unreachable(1645212451)),
},]
```

Table 4.10: Convergence time scenario 4

	convergence time	
$N = 100$	$\overline{x}$	s
step 1	18451	2991
step 2	2865	1171
step 3	17608	3255

This table shows, that in an extremely asymmetric network the convergence times are significantly higher in the network buildup phase. This is because it takes some time to recognize a node as unreachable. In case of a collector node failure the network is able to react quite fast, because one of the nodes is always connected to the collector node, recognizes the fail and sends the information to its connected neighbors. They in turn just have to adapt the new priority values in their node lists. For the same reason already explained in scenario 3 the nodes don't have to change the node they send their sensor data to. Therefore, the reaction is quite fast.

In this scenario 3 errors were encountered. Similar to the first scenario, the error occurred due to a failing node because of a memory corruption in the MQTT-library.

Scenario 5

This scenario is different from the other ones. It simulates a TCP/IP network failure leading to a split network. Node A and B are on the same LAN as well as C and D. The two LANs are connected with a router. This router will stop forwarding packets 30s after all nodes are online, separating the two networks at ISO/OSI layer 3. This should lead to two separate networks with collector node B in the A, B network and collector node D in the C, D network.

After waiting further 90s the router will start forwarding packets again. This should lead to a reunion of the two distinct networks that appeared before. The expected collector node after the reunion is Node D.

Step 1 in this scenario is the building of the network after all four nodes are started at the same time. In the 2nd step, the time it takes to form 2 distinct networks is measured. This means actually 2 different times. On the one hand the time needed to form the network consisting of nodes C and D and on the other hand the time needed until A and B form their own network. The third step measures the time it takes the two networks to reunite.

Table 4.11: Convergence time scenario 5

	convergence time	
$N = 100$	$\overline{x}$	s
step 1	5819	1744
step 2 AB	61401	1589
step 2 CD	1156	306
step 3	2384	752

No errors occurred in this scenario.

This test proves the ability of the network to adequately react to failures in the underlying TCP/IP network. It shows that after a failure 2 different networks are formed and in case of recovery the sensor network is reunited to a single network again.

What stands out in this scenario when looking at the convergence times is that the time it takes to form networks AB and CD in the event of a router failure is very different. This is because Node D is the collector node, therefore the nodes C and D simply continue sending their sensor data in 1s intervals to Node D. Hence

the convergence time of approximately 1s.

The situation for Node A and Node B is totally different. They recognize the failure immediately after the router stopped forwarding packets. After that they try to contact Node D via the Tor network. This does not work either, because the router, which acts as default gateway for all four nodes, stopped forwarding packets. The time it takes to finally accept Node D as unreachable is approximately 30 seconds. This is determined by the Tor connection timeout of 10 seconds and the maximum number of connection attempts before a node is considered unreachable, which is 3 times. This is because if a node in the same network fails other nodes can recognize this immediately, whereas if the router fails the node has to wait for a connection timeout. Node C is the node with the next highest priority, so it takes again 30s to recognize its unreachability, resulting in a total convergence time of approximately 60 seconds.

Summary

All five test scenarios show that the program is able to build functioning networks even in very restricted network environments.

The convergence times measured show that the present network is well suited for collecting environmental sensor data. It is unsuitable for time-critical applications, where it's necessary to have real time "live" measurements.

4.2 Testing Authentication

Section 3.1 already mentioned that the authentication is done at the application layer by the `mosquitto` MQTT broker. Therefore, the tests are performed just on that broker and not the whole application. The broker was set up on a Virtualbox VM running the x86 version of OpenWRT with 128M of RAM. It uses TLS for encrypting the communication.

Wrong or empty password

If the broker is configured properly it is not possible to connect without or with a wrong password. This can be easily tested with the clients `mosquitto_pub`

or `mosquitto_sub` that are part of the `mosquitto` project. The following screenshots show the behavior of the MQTT broker in case of wrong authentication credentials. The first screenshot in figure 4.1 shows a successfully connected client that subscribed the “test” topic.

Figure 4.1: Successfully authenticated MQTT client

```
peda@riemann:~/uni/masterarbeit/code/installer/openwrt-testnode$ mosquitto_sub -h 192.
168.56.100 -u test -P test --cafile owrt-testnode-ca/pki/ca.crt -t "test" --insecure
test 1
```

The second one in figure 4.2 shows clients that try to publish on the “test” topic using different combinations of username and password. The actual username is `test` and the password `test`.

Figure 4.2: Unsuccessful authentication attempts

```
peda@riemann:~/uni/masterarbeit/code/installer/openwrt-testnode$ mosquitto_pub -h 192.
168.56.100 -u "test" -P "test" --cafile owrt-testnode-ca/pki/ca.crt -t "test" -q 0 -m
"test 1" --insecure
peda@riemann:~/uni/masterarbeit/code/installer/openwrt-testnode$ mosquitto_pub -h 192.
168.56.100 -u "test" -P "hallo" --cafile owrt-testnode-ca/pki/ca.crt -t "test" -q 0 -m
 "test 2" --insecure
Connection error: Connection Refused: not authorised.
Error: The connection was refused.
peda@riemann:~/uni/masterarbeit/code/installer/openwrt-testnode$ mosquitto_pub -h 192.
168.56.100 -u "test" -P "" --cafile owrt-testnode-ca/pki/ca.crt -t "test" -q 0 -m "tes
t 3" --insecure
Connection error: Connection Refused: not authorised.
Error: The connection was refused.
peda@riemann:~/uni/masterarbeit/code/installer/openwrt-testnode$ mosquitto_pub -h 192.
168.56.100 -u "" -P "" --cafile owrt-testnode-ca/pki/ca.crt -t "test" -q 0 -m "test 4"
 --insecure
Connection error: Connection Refused: not authorised.
Error: The connection was refused.
peda@riemann:~/uni/masterarbeit/code/installer/openwrt-testnode$ mosquitto_pub -h 192.
168.56.100 -u "" -P "test" --cafile owrt-testnode-ca/pki/ca.crt -t "test" -q 0 -m "tes
t 5" --insecure
Connection error: Connection Refused: not authorised.
Error: The connection was refused.
peda@riemann:~/uni/masterarbeit/code/installer/openwrt-testnode$ mosquitto_pub -h 192.
168.56.100 -u "hallo" -P "test" --cafile owrt-testnode-ca/pki/ca.crt -t "test" -q 0 -m
 "test 6" --insecure
Connection error: Connection Refused: not authorised.
Error: The connection was refused.
peda@riemann:~/uni/masterarbeit/code/installer/openwrt-testnode$
```

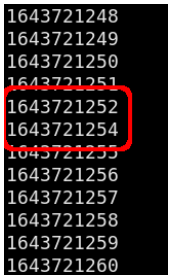
This shows that the authentication mechanism implemented in the mosquitto MQTT broker used in this project works as expected.

DoS with many failing authentication attempts

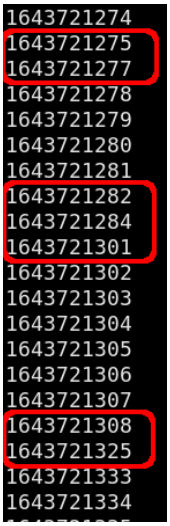
As mentioned in section 3.1, even unsuccessful authentication may interfere with normal operation with the broker. This test shows the vulnerability of the mosquitto MQTT broker to DoS attacks using authentication with wrong credentials. For this test one client subscribes to the topic "test" on the broker and listens for incoming messages. Another client sends a message to the broker on the "test" topic containing the current UNIX timestamp in seconds. The broker then gets contacted by 10 and later by 100 clients trying to authenticate with wrong credentials.

With only 10 clients trying to connect with wrong credentials only one message was lost and not received by the client that subscribed the topic. This is shown in the screenshot in figure 4.3a.

Figure 4.3: Part of screenshot that shows message loss



(a) 10 clients trying to authenticate



(b) 100 clients trying to authenticate

This dramatically changes if 100 clients try to authenticate with wrong credentials. The screenshot in figure 4.3b shows the loss of 33 out of 60 messages.

This shows that this kind of DoS attack can be quite successful.

5. Conclusio and Outlook

In the course of this master thesis, an application was developed that allows a decentralized and distributed sensor network to be set up via the existing Internet network infrastructure. This was done utilizing the MQTT application layer protocol, which was used for transporting sensor data, as well as network management messages. The network operates without a central infrastructure such as nameservers, MQTT brokers or anything else, which is not directly provided by the sensor nodes itself. Every node in the network could be the one collecting the sensor data. The nodes that build the network are also able to operate behind NAT gateways, using the Tor networks onion services for NAT traversal.

Furthermore, the network is able to compensate for failures of the collector node, which collects the sensor data centrally. This is even possible in restricted network settings, where not all nodes are able to communicate directly with each other.

Another central goal was to create a network in which data transmission is always authenticated and encrypted.

Authentication in the present network is done based on username and password as provided by the MQTT protocol [9]. The encryption is implemented using TLS.

TLS and the password-based authentication mechanism serve their purpose in this project. But both are designed to be used in a client server setup with rigidly fixed roles. This does not fit with the peer-to-peer structure of the present sensor network. Especially TLS uses a centralized approach for establishing trust, therefore it is only used for encryption. How this was nevertheless integrated into the network designed here is described in section 3.4.

5.1 Possible improvements

Even though the current implementation of the sensor network fulfills the requirements defined in section 1.1, there is still room for improvements.

The two main areas in which improvements seem possible are:

- The way trust is established. As already mentioned the TLS way of establishing trust does not really fit the peer-to-peer structure of the network.

- The protocol used for network management. In order to minimize the convergence times and the traffic in case the collector node is not reachable directly some improvements on the network management protocol could be made.

Furthermore, some implementation specific improvements are also conceivable.

5.1.1 Improving Trust

The authentication mechanism currently in use lacks a procedure for revoking rights once granted. Furthermore, there are two distinct types of authentication. The broker authenticates itself to the clients using a self-signed TLS certificate. The clients authenticate themselves to the broker using a username and password. It would be desirable to have a unified cryptographic system that would allow both mutual authentication and revocation of privileges.

Establishing trust and authentication in decentralized, peer-to-peer networks is a separate research topic and beyond the scope of this work. A search for research publications in this field revealed different approaches to solving this problem. Two of the papers propose a kind of reputation based system, one based on votings[24] the other based on "performance in the network"[23]. Most of the recently published papers focus on a blockchain based approach to authentication and trust[1].

[1] A search on scholar.google.com on 29/03/2022 for the term "distributed authentication" returned 13 blockchain based authentication schemes of the top 20 search results

As already mentioned it is beyond the scope of this work to implement such a decentralized authentication scheme, but the present system would profit from an authentication system that reflects the peer-to-peer characteristics of the network and enables mutual cryptograhic authentication and the revocation of authentication.

5.1.2 Improving Network Management

Improving convergence time

Especially in scenarios like test scenario number 5 in section 4.1 where the network is split due to a failure on the network layer and lots of nodes with low priority values are concentrated on the same "side" of the resulting networks it takes extremely long for the network containing the nodes with the higher priority values to converge.

To improve this situation a "last_contact" field could be introduced to the node list containing a timestamp with the last successful data transfer to a node. The nodes keep sending some kind of "keep alive pings" at regular intervals to their neighbors. This information gets propagated through the network and helps to indicate which nodes may be unreachable.

Such a mechanism could also help to improve convergence time in case the collector node and several top node candidates go offline at a short time interval.

Minimizing relay traffic and the number of relays needed

The current implementation sends the sensor data to its neighbor if the collector node is not directly reachable. If in turn the neighbor is also unable to reach the collector node directly it forwards the data to its neighbor and so on, potentially leading to a long chain of forwardings. This situation is not optimal concerning the network traffic generated and the time it takes for the sensor data to arrive at its destination.

This could be solved using top node reachability information of every node in the node list. It could be implemented by a field "reaches_collector" in the node list, indicating the way that node is able to reach the collector node. Based on

that, the data can be sent to a node that is reachable directly and has a minimal relay distance to the collector node. This would require that the nodes check the reachability of all other nodes constantly and update their node lists accordingly.

These two measures can be seen like a kind of primitive routing protocol. The downside of these two outlined solutions is increased network management traffic. Depending on the specific application, a decision must be made whether increased network traffic or a higher convergence time is acceptable. This could even be made configurable to adjust the convergence time and network traffic to the application built on top of the network.

Implementation specific improvements

The current implementation relies on an external MQTT broker and the Tor daemon. These two could be implemented as part of the project.

Especially if a suitable trust scheme is to be implemented, it is inevitable to implement a custom MQTT broker. It would also bring further advantages like better error handling, which is now constrained by the capabilities the process control API offers, which basically means starting and stopping the process and checking if the process is still running. Furthermore, a custom MQTT client library would be needed to implement the already mentioned trust scheme. This allows to improve the Tor connections as well. Right now a local proxy relay is used to transmit traffic over the Tor network. This local relay could be avoided, again improving error handling and thus robustness.

To further reduce the dependency on other processes, the Tor onion service could be implemented as part of this program. This would also increase control and robustness due to better error handling. Even though it is possible to use C libraries from Rust, it is not very desirable to use “unsafe” code. Fortunately the Tor project started the “Arti” project in 2021. This is “a project to rewrite Tor in Rust”[16]. They already released some libraries, but this code is far from being production ready and does not offer support for onion services in particular yet.

Bibliography

[1] Architecture - OpenSensorHub. URL: https://docs.opensensorhub.org/dev/architecture/.

[2] Env_logger - Rust. URL: https://docs.rs/env_logger/0.8.3/env_logger/.

[3] ISO/IEC 29182-2:2013(en), Information technology — Sensor networks: Sensor Network Reference Architecture (SNRA) — Part 2: Vocabulary and terminology. URL: https://www.iso.org/obp/ui/#iso:std:iso-iec:29182:-2:ed-1:v1:en.

[4] Memory safety - The Chromium Projects. URL: https://www.chromium.org/Home/chromium-security/memory-safety.

[5] Rfc7468. URL: https://datatracker.ietf.org/doc/html/rfc7468.

[6] Sysinfo::SystemExt - Rust Documentation. URL: https://docs.rs/sysinfo/0.15.9/sysinfo/trait.SystemExt.html#tymethod.get_available_memory.

[7] Tor Project — How do onion services work? URL: https://community.torproject.org/onion-services/overview/.

[8] The Tor Project — Privacy & Freedom Online. URL: https://torproject.org.

[9] MQTT Version 3.1.1. Technical report, December 2015. URL: http://docs.oasis-open.org/mqtt/mqtt/v3.1.1/mqtt-v3.1.1.html.

[10] The Open Group Base Specifications, 2018. URL: https://pubs.opengroup.org/onlinepubs/9699919799/basedefs/V1_chap04.html#tag_04_16.

[11] E Androulaki, S Lambropoulou, I G Economou, and J H Przytycki. Inductive construction of 2-connected graphs for calculating the virial coefficients. *Journal of Physics A: Mathematical and Theoretical*, 43(31):315004, August 2010. URL: https://iopscience.iop.org/article/10.1088/1751-8113/43/31/315004.

[12] Catalin Cimpanu. Microsoft: 70 percent of all security bugs are memory safety issues. URL: https://www.zdnet.com/article/microsoft-70-percent-of-all-security-bugs-are-memory-safety-issues/.

[13] Transforma Insights. Global IoT Forecast Report, 2019-2030. Technical report, November 2020.

[14] Leslie Lamport. The part-time parliament. *ACM Transactions on Computer Systems*, 16(2):133–169, May 1998. URL: https://doi.org/10.1145/279227.279229.

[15] D. MacDonald and B. Lowekamp. NAT Behavior Discovery Using Session Traversal Utilities for NAT (STUN). (RFC 5780), 2010. URL: https://www.rfc-editor.org/info/rfc5780.

[16] Nick Mathewson. Announcing arti, a pure-rust tor implementation. URL: https://blog.torproject.org/announcing-arti/.

[17] Philip Matthews, Alan Johnston, Jonathan Rosenberg, and Tirumaleswar Reddy. K. Traversal Using Relays around NAT (TURN): Relay Extensions to Session Traversal Utilities for NAT (STUN). URL: https://tools.ietf.org/html/rfc8656.

[18] Philip Matthews, Rohan Mahy, Dan Wing, Marc Petit-Huguenin, Jonathan Rosenberg, and Gonzalo Salgueiro. Session Traversal Utilities for NAT (STUN). URL: https://tools.ietf.org/html/rfc8489.

[19] Diego Ongaro and John Ousterhout. In Search of an Understandable Consensus Algorithm. page 18.

[20] Tom Preston-Werner, Pradyun Gedam, and et al. TOML: English v1.0.0. URL: https://toml.io/en/v1.0.0.

[21] Jonathan Rosenberg and Christer Holmberg. Interactive Connectivity Establishment (ICE): A Protocol for Network Address Translator (NAT) Traversal. URL: https://tools.ietf.org/html/rfc8445.

[22] Jens M. Schmidt. A simple test on 2-vertex- and 2-edge-connectivity. *Information Processing Letters*, 113(7):241–244, April 2013. URL: https://linkinghub.elsevier.com/retrieve/pii/S0020019013000288.

[23] Aameek Singh and Ling Liu. Trustme: anonymous management of trust relationships in decentralized p2p systems. In *Proceedings Third International Conference on Peer-to-Peer Computing (P2P2003)*, pages 142–149, 2003.

[24] Andreea Visan, Florin Pop, and Valentin Cristea. Decentralized trust management in peer-to-peer systems. In *2011 10th International Symposium on Parallel and Distributed Computing*, pages 232–239, 2011.

Statutory Declaration

I hereby declare that the thesis submitted is my own unaided work, that I have not used other than the sources indicated, and that all direct and indirect sources are acknowledged as references.

Linz, April 2022

Eidesstattliche Erklärung

Ich erkläre an Eides statt, dass ich die vorliegende Masterarbeit selbstständig und ohne fremde Hilfe verfasst, andere als die angegebenen Quellen und Hilfsmittel nicht benutzt bzw. die wörtlich oder sinngemäß entnommenen Stellen als solche kenntlich gemacht habe.

Linz, April 2022

Peter Wagenhuber