It’s about time - THALES

Group 17; Iris Borgonjen, Ryan Bartelds, Kevin Veldman, Lars van Soest, and Floortje ter Avest

Date: 17 april 2026

Client: Jan Laarhuis

Supervisor: Sabari Anbalagan

Contents

1. Introduction 3
1.1. Thales 4
1.2. Problem description 4
1.3. Terms and definitions 5
2. Background 6
2.1. Theory 6
2.2. Simulator 8
3. Requirements specification 10
3.1. Must have 10
3.2. Should have 11
3.3. Could have 11
3.4. Non-functional 11
4. System design 12
4.1. System overview 13
4.2 Discovery 14
4.3. Clock 15
4.4. Local time difference calculation 16
4.5. Distribution 18
4.6. Global calculation 19
4.7. Error propagation 20
4.8. Message formatting 21
4.9. Logging 22
5. Software design 23
5.1 Software overview 23
5.2. Discovery 24
5.3. Local 24
5.4. sendReceive 25
5.5. Dictionary 26
5.6. Global calculation 27
5.7. Message payloads 30
5.8.1 Global distribution format 31
5.9. Logging 31
5.10. Message 32
6. Miscellaneous development activities 33
6.1. Start and Stop 33
6.2. Simulator Problems 33
6.3. NTP 33
7. Testing 34
7.1. Unit testing 34
7.2. System testing 37
7.3. General setup for testing with the simulator 38
8. Results 43

8.1. Correctness 43
8.2. Performance 43
8.3 Deviation 44
9. Conclusion **44**
10. Future work **45**
11. Reflection **46**
11.1. Improvements 46
11.2. Accomplishments 46
12. Individual Contributions **47**
13. Usage of AI **48**
14. References **49**
15. Appendix **51**
A. System tests 51
B. Influence of sigma graph 54

1. Introduction

1.1. Thales

The client for this project is Thales. Thales is described as a global leader in defence, aerospace, cyber and digital technology, Thales plays a pivotal role in developing solutions that help address the major societal and environmental challenges of our times (About Thales, n.d.). They provide solutions, services and products to help its diverse range of customers to carry out their critical missions, for example military organizations.

1.2. Problem description

When working together in groups, for example with ships at sea or drones in a swarm, information exchange over a wireless network is of great importance. These exchanges consist of different observations done by the nodes in the network. The fusion of exchanged observations is dependent on time alignment to make sure that every node in the network can get a complete picture from the different observations. One of the ways to align nodes in time is through the use of GPS. This aligns all nodes with the 'right' time. However, there is not always the possibility to make use of GPS. In these cases, nodes in a swarm instead need to have the 'same' time and thus have relative time alignment with each other. For this problem time differences with directly connected nodes have to be measured in a wireless mesh network. This network is a mobile ad-hoc network at sea, called MaritimeManet. Then, the information that was measured needs to be shared with other nodes. Lastly, all received information should be fused to obtain the time difference between any pair of nodes. To implement and test these principles, all of this has to run on an existing virtualized simulator.

1.3. Terms and definitions

Within this document we use the following terms and definitions:
Dictionary: A data structure, which uses a key and one or more values.
Eth0 & Bat0: These are interfaces on the mesh nodes. Bat0 is used as the interface to communicate across and eth0 is the interface with which the BATMAN-adv protocol manages the network.
Expiry time: An data characteristic which limits the lifespan of time difference data within the MANET.
Function: A function is a group of actions for a computer to perform. A function can have input values and it can return an output.
MANET: Stands for Mobile Ad-hoc Network.
Neighbour: The neighbours of a specific node are all the nodes in the network that have a direct connection to that node.
Network: The network represents all the nodes that are connected to each other via direct connections or indirect connections.
Node: A node is a ship or drone.
RTT: RTT stands for Round-Trip Time. This is the time it takes for a data frame to travel from the source to the destination and back.
Simulator: The simulator is a reference to the virtual simulator of the maritime MANET on the proxmox machine.
Time difference: The time difference between two nodes is the difference between the clock time of both nodes.
Thread: Within a process multiple threads can be created these threads can run concurrently with some shared system memory. These threads can be used to perform multiple different actions at the same time.
VM: VM stands for Virtual Machine.

2. Background

2.1. Theory

2.1.1. Local measurements

To be able to determine the time differences between a node and all other nodes in the network, it is essential to establish the time difference with neighbouring nodes. This is done with the local measurements. In order to measure this, a timestamp-based approach could be used. With such an approach, a message is sent to a neighbour of a node and that neighbour sends it back. The return-message contains timestamps for both sending and both receiving times. One way of getting these timestamps is by the use of TWAMP. TWAMP stands for Two-Way Active Measurement Protocol. In networking it is widely used to measure performance between two nodes. For example, it helps measure latency, packet loss and jitter. (Vouzis, 2025) TWAMP works with the principle of a sender related to a client, and a reflector related to a server. The protocol involves several steps to complete its measurements. First of all, it starts with a session setup. In this control phase the sender connects to the reflector and negotiates a test session. Once the session is set up the sender sends test packets with timestamps to the reflector in the test session. After that the reflector immediately sends the packets back. (Hedayat et al., 2008) The last step is doing the actual measurement. This means that the sender compares when the packet was sent and when it came back. Based on these comparisons it can then determine delay jitter and packet loss. TWAMP provides accurate results, however there are also a few disadvantages to it. For example, the configuration can be quite complex and it needs specialized hardware. Twampy makes this a lot easier. Twampy is an implementation of TWAMP written in Python. This makes it software-based and a lot easier to experiment with. Twampy runs in reflector mode on one node and in sender mode on another. This way the sender can send timestamped packets and the reflector can echo them back according to the protocol such that the metrics can be calculated. (Wisotzky, n.d.)

The time alignment in this project does not require all these different performance metrics. This implementation merely makes use of the time differences between the clocks of each pair of nodes. Thus implementing TWAMP as a whole is not necessary. Instead, the important timestamps and the eventual calculation of the time differences can be implemented with use of the method described by Van Steen & Tanenbaum (2023, Chapter 5). This method is based on the principles of NTP (Network Time Protocol). However, where a node using NTP contacts a time server that usually is connected to GPS, this implementation only makes use of two nodes. This means that one of the nodes in the network is the sender and the other functions as the time server, causing the nodes to get differences relative to each other.

2.1.2. Global time differences

For the global time differences there are other algorithms to implement the distribution and calculation. One approach to this are tree-based time synchronization algorithms. The core idea of these protocols is that the network is organized in a hierarchical structure. This means that one node acts as the root, which is the leader in the network. Then, other nodes synchronize

along the edges of the tree. This causes that time flows from the root down through the rest of the tree. Relevant examples of such an algorithm can be seen in the Flooding Time Synchronization Protocol (FTSP) as described by Maróti et al. (2004) and Tree Structured Time Synchronization Protocol (TSTP) as described by Rahamatkar et al. (2009).
FTSP is used in wireless sensor networks and works with a few steps. First of all, assuming that all nodes have gotten a unique ID when entering the network, the node with the lowest ID becomes the root node. This node has the reference time to which all nodes will synchronize. This root then periodically broadcasts synchronization messages to all nodes in its broadcast radius containing information on when the message was sent. These messages then propagate through the network. They start at the root and go down the tree, starting with the nodes within the broadcast radius of the root. Those newly synchronized nodes then broadcast synchronization messages as well, and so on. When a node receives such a message, it updates its clock and rebroadcasts it such that it propagates through the tree. As a result every node calculates its time difference relative to the clock of the root node.
TSTP works in two phases. The first phase constructs an ad-hoc tree structure and the second phase synchronizes the local clocks of sensor nodes. For the synchronization phase pair wise synchronization is done along all edges of the tree constructed in phase one. The synchronization process is built on the two-way message exchange between a pair of nodes. In a single message exchange it is assumed that the clock drift between a pair of nodes stays constant in this small period of time. In addition, it is also assumed that the propagation delay is constant in both directions. The root node initiates this synchronization phase. After initiation nodes on level 1 start a message exchange with the root node to adjust their clock. Then, nodes on level 2 do the same until all nodes get synchronized to the root node. Each node waits for a random amount of time to ensure nodes from the level above have completed synchronization. Because this is a random number, collisions on the wireless channel are minimized as well.
To achieve high precision, both aforementioned algorithms use linear regression to compensate for clock drift.
However, there are a few downsides to such approaches. Firstly, the algorithms are sensitive to topology changes, there is a single point of failure and errors accumulate over multiple hops. Besides this, these algorithms assume actual adjustment of the clock as opposed to the calculation of relative time differences that is needed as described in chapter 4.3.

In a mobile ad-hoc network these are not disadvantages that can be ignored. One of the main characteristics of the network that is worked with is that nodes move in and out of the network and thus cause topology changes. For this reason and the fact that the synchronization should not fail completely if a single node fails, the algorithm should be adjusted a bit. In this adjusted algorithm there is not a single leader node. In this algorithm every node acts as a leader. For every node, local measurements are done with direct neighbours. This information will then propagate through the network from neighbour to neighbour. Since this is done from every node, all nodes receive a lot of information. To choose which information it has to save, the node looks at the amount of hops it took from the source of the information to this node. As stated before, errors accumulate for every hop. Thus, the information which has travelled the least should be the most accurate. So, from all the data that comes in, the node should keep the one with the least amount of hops away from the source.

2.2. Simulator

2.2.1. VMs

A simulator for node-swarms was provided and used for testing our system in a realistic situation. The simulator is built from 4 types of VMs.

Simulator VM: This VM handles the simulator and communicates with all other VMs to do so. Within this VM, simulations can be made, started and stopped. This VM is also used for running the time alignment application, as this VM has SSH connections to all other VMs.

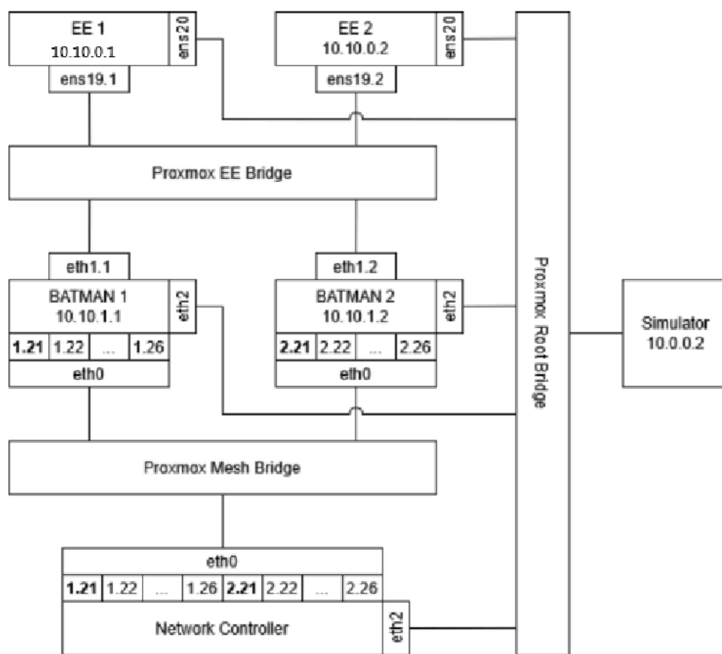


Figure 1: Detailed network topology for two nodes. Edited from original (Hebbink, 2025)

Network Controller VM: This VM handles the BATMAN network of the simulation. By communicating with the simulator, it creates and deletes connections between the BATMAN VMs to make sure that these connections accurately represent the state of the simulation.

BATMAN VM: This VM handles the networking within the mesh of BATMAN VMs in the Proxmox Mesh Bridge. It does networking using the OpenWRT, which is a stripped-down version of linux, BATMAN-adv implementation, which is a Layer-2 network protocol. Each BATMAN VM has a corresponding EE VM, where the BATMAN VM serves as the network entry-point for the EE VM.

EE VM: This VM is not used for any networking/ simulator functionality. However, this VM serves as a machine to run client-side applications on the network of the simulator. It has relatively much computing power compared to a BATMAN VM, to allow these client-side applications to work.

2.2.2. Time Alignment Application on the Simulator

Initially, the time alignment application was to run on the EE VM, as this VM was made for client-side applications. However, this EE VM has no way of accessing its direct neighbours, which would be the starting point of the application.

This forced the application to be split in 2, with neighbour discovery being done on the BATMAN node and the rest of the system being done on the corresponding EE node. This is because the OpenWRT BATMAN-adv implementation provides the 'batctl n' command for accessing a BATMAN VMs direct neighbours, thus solving the neighbour discovery problem. A split was done instead of moving the whole application because of the BATMAN node only supporting compiled programs at the time and it having very low computing power. Compiled programming languages were a thing that most of the team was not very experienced with, thus making the whole application in such a language would be too time-intensive for this project. A C program for getting neighbours and sending that data to the corresponding EE VM was created for this discovery and cross-compiled to run on the BATMAN VM.

This command returns the MAC addresses of neighbouring BATMAN nodes, which were not usable if the System was to be implemented on EE nodes. To solve this, the whole system was moved to the BATMAN nodes, and a python interpreter was installed to allow all team-members to work on code for the BATMAN node. This interpreter turned out to have surprisingly good performance, thus it worked better than expected. The created Python code for the EE node was mostly usable for the BATMAN node, so not much time loss resulted from this. The only time loss for this was the created C programs becoming obsolete.

3. Requirements specification

The requirements were elicited by discussing them with the client. The system requirements cover both the simulator and the algorithm. They are categorized using the MoSCoW method. The labeling for each functional requirement ID is done using the format:
Must have: requirements that are essential for the project functionality.
Should have: requirements that are important, but not critical to the functionality of the project.
Could have: requirements that are considered nice to have, but not essential to the project functionality.

3.1. Must have

ID	Description	Status
M-1	Each node must measure the time difference between itself and its neighbours.	Done
M-2	Each node must distribute the measured time difference across the network.	Done
M-3	Each node must calculate the time difference between itself and all other nodes using the time differences measured and received via the network.	Done
M-4	The system must run on the simulator.	Done
M-5	The system must work for a single and multi-hop network.	Done
M-6	The system must work for changing network topologies, including changing distances between nodes and nodes entering/leaving.	Done

Table 1: Functional Must have requirements

M-1, **M-2** and **M-3** are to be made in a modular fashion to allow future implementation and testing of different solutions to each of these requirements. The implementations of requirements **M-1**, **M-2** and **M-3** will be repeatedly called every 3 seconds to correct clock skew.

3.2. Should have

ID	Description	Status
S-1	The system should detect and discard/weigh down outlier measurements.	Partial*
S-2	Each node should make its current knowledge of time differences in the network available to the user running the simulator.	Done

Table 2: Functional Should have requirements

S-1: When measurements are done and outliers are detected, whether that be on the local or global level, those measurements should be discarded or, depending on the algorithm, hold less weight compared to the measurements that are not considered outliers.
*: this was done by implementing local measurements as bursts instead of single measurements

S-2: The nodes should have some way that a user can access their stored information. This could be done by producing logs, or by allowing the user to request the current information on-demand.

3.3. Could have

ID	Description	Status
C-1	The system could run and compare different implementations in specific networks.	Not Done

Table 3: Functional Could have requirement

C-1: The system allows users to select multiple algorithms. For a given topology in the simulator the performances of both algorithms can then be compared. For example, to see which algorithm works best in moving topologies.

3.4. Non-functional

ID	Description	Status
N-1	The output of the system should be human-readable.	Done

Table 4: Non-functional requirement

4. System design

The designed solution to the time alignment problem for a MANET contains three main parts. The first part is about calculating the time difference between one node and its neighbouring node(s). This will be done by using the principle stated by Van Steen & Tanenbaum (2023, Chapter 5). In the second part, the local information will be broadcasted. To ensure that every node knows their time difference with any other node, this is needed to calculate these differences based on the information that the nodes receive. This is the last part, glocal calculation. When these three parts are combined, it should result in code that tackles the problem of time difference in a MANET. More smaller parts exist within the system, to ensure that these three steps can be performed. All parts are explained in detail in this chapter, beginning with a global system overview, then the separate parts of the system are explained in detail.

4.1. System overview

Before every part of the system is explained in detail, a general overview of the system's components is shown.

4.1.1. Sequence Diagram

The Sequence Diagram in (figure 2) shows the complete system. The system runs in a loop where all the actions are repeated periodically. The initial period for this will be set to 3 seconds. The ref square is a reference to a diagram for the local time synchronisation, this is presented later in this report (see chapter 4.4.1., figure 6). The alt square represents a choice. If there is new data received in the current iteration of the loop, the action on top of the dotted line is executed, else the action below that line is executed.

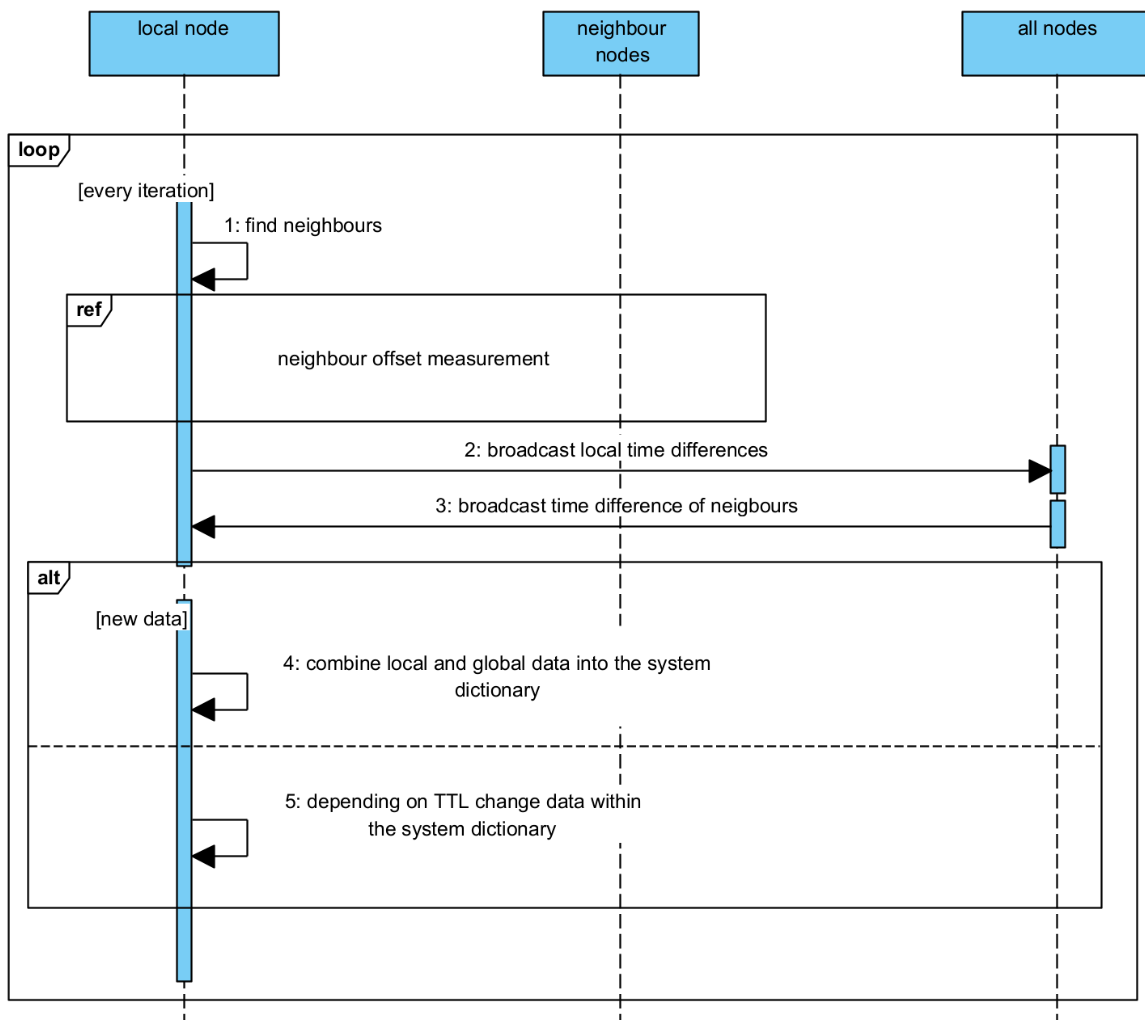


Figure 2: Complete system overview

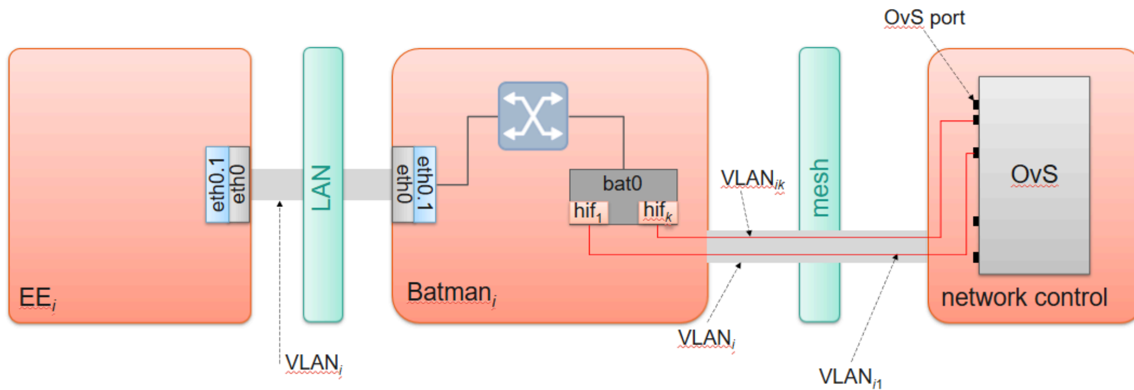


Figure 3: VMs in MaritimeManet simulator (Thales, n.d.)

4.2 Discovery

This part of the system is concerned with finding direct neighbours of a batman node. This is a required step of the protocol, as the Local time difference measurement system needs a way of knowing what neighbours to measure with.
The neighbour discovery is implemented using the ‘batctl n’ command provided by OpenWRT BATMAN-adv. This command gets the MAC addresses of the eth0 interface of neighbouring BATMAN VMs. The output of this command is parsed to retrieve these addresses, after which they can be used in the local measurements. The discovery is performed automatically at the start of each iteration, after which the local measurements are started.
A problem with this ‘batctl n’ command is that it gives the MAC address of the wrong interface for this use-case. BATMAN nodes can communicate with each other via the bat0 interface, but the command gives the MAC address of the eth0 interface. This means that sending messages to neighbours using the ‘batctl n’ MAC addresses, results in no node receiving the message.
To solve this problem, local measurement messages are broadcast and contain the eth0 MAC address of the sender. When a BATMAN node receives these messages, it can quickly check if the received eth0 address is a neighbour of it using the ‘batctl n’ data. If so, it continues the measurement, else it discards it.

4.3. Clock

Within the simulator every node should have a clock with a randomized time difference. This is to simulate the time differences in the clocks of the nodes in the network. To create the clock with a time difference a system clock and a random number are needed. For this to work the system clock of every virtual machine must be the same. In the simulator each node consists of two virtual machines. However, since our program will run only on the virtual machine created for the BATMAN protocol, the solution will only consider one virtual machine for each node. Each of these virtual machines run openWrt, which is a linux based operating system. In linux timekeeping a virtual machine uses two different actions to get an accurate time for the system.

Whenever a linux virtual machine is started it receives the current time of the host. To keep track of the passage of time in virtual machines, linux uses a hypervisor, Hyper-V (linux kernel) in specific. A hypervisor is a system that uses the host machine to provide virtual machines with a synthetic system clock. The virtual machines can then read this system clock to update their own time. However, between receiving the time from the host machine, and the Hyper-V connection starting some time may elapse. This was an issue encountered within the simulator, and to solve this NTP was used as described in chapter 6.3.

Since the whole simulator runs on one machine, all virtual machines in the simulator get their clock from the same hypervisor. This means that all virtual machines follow the same clock speed. Thus, all virtual machines have the same system time within the simulator. This means that to get the clock with a time difference for the simulator each virtual machine can get a random number (Δ) and add this to their own clock time as shown in figure 4. This random number will be such that each clock gets a random offset between -200ms to 200ms, relative to their original times.

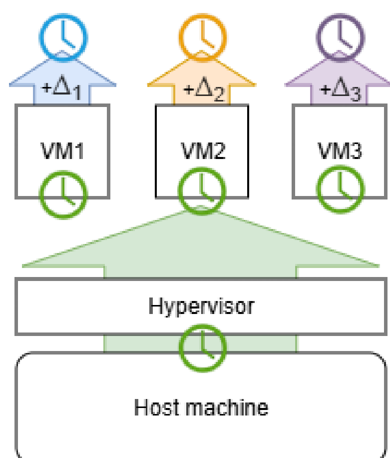


Figure 4: Clock visualisation

In order to simulate inaccurate clocks and thus cause inaccurate measurements, the added offsets need to be more randomized by using a Gaussian distribution. This kind of distribution takes a standard deviation (σ) and a mean (μ). The mean will be the random offset, and the standard deviation can be any value, for our purposes we tested with a value of 0, 5, and 10 (ms). The mean simulates the time differences between clocks and the standard deviation simulates fluctuations in measurements. When a standard deviation of 0 is used, the outcome is

always equal to the random offset, which is used to test the accuracy of the protocol without any noise.

4.4. Local time difference calculation

Every node in the network has some time difference compared to its neighbour. To determine these local time differences between every node and its neighbours, the principle stated by Van Steen & Tanenbaum (2023, Chapter 5) is used. To measure the time difference, this principle makes use of different timestamps between a sender and a receiver.
These timestamps are the following:

T1. Sender send time
T2. Receiver receive time
T3. Receiver send time
T4. Sender receive time

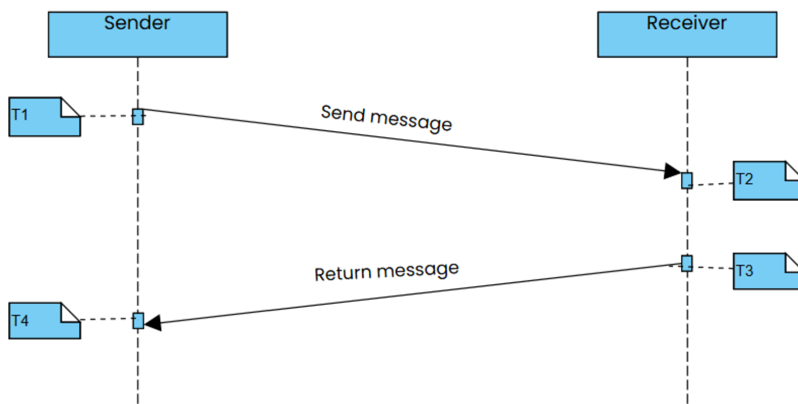


Figure 5: Timestamps in local measurements

A protocol that implements the measurement of these timestamps is the Two-Way Active Measurement Protocol also known as TWAMP. However, this protocol also implements a lot of other things, which are not useful for our case. This means that when using implementations of TWAMP, like twampy, there will be a higher chance of inexplicable bugs and slower measurements due to these extra functionalities needing extra actions from each node. TWAMP needs a setup exchange before any measurements can be done, which will not be needed with an implementation of our own, this way measurements can be performed quicker. Thus, these extra functionalities will be removed and a simplified version of this protocol will be implemented that merely contains the timestamps such that the difference can be estimated. Based on these timestamps, the time difference of the sender relative to the receiver can be estimated with

$$\theta = \frac{(T_2 - T_1) + (T_3 - T_4)}{2} \tag{1}$$

For this to hold, it is assumed that the propagation delay for a message from sender to receiver is roughly equal to the propagation delay for the return message from receiver to sender. Since this is done for every node and its corresponding neighbours, the measurements and estimations are done both ways. For example, this means that one time A is the sender and B is the receiver, whereas another time B is the sender and A is the receiver.

This whole process is run in bursts to achieve high quality measurements. The measurements are done a set amount of times, which is initially set to 5, after which the measurement with the lowest RTT will be selected as the best measurement. This is the best measurement because lowest RTT is the most accurate RTT. RTT can be easily calculated using the already made timestamps, thus only the burst itself needs to be implemented after having a base implementation with no burst. Using the timestamps from formula 1, the RTT will be defined as:

$$RTT = (T_4 - T_1) - (T_3 - T_2) \quad (2)$$

4.4.1. Overview local calculation

A sequence diagram is given below to give a better visual overview of the local calculation. This diagram shows the local time alignment, as described above. The diagram (figure 6) shows that action 2: send a message is an asynchronous operation. This is because the implementation sends 5 requests in a burst, after which the rest of each measurement is pipelined.

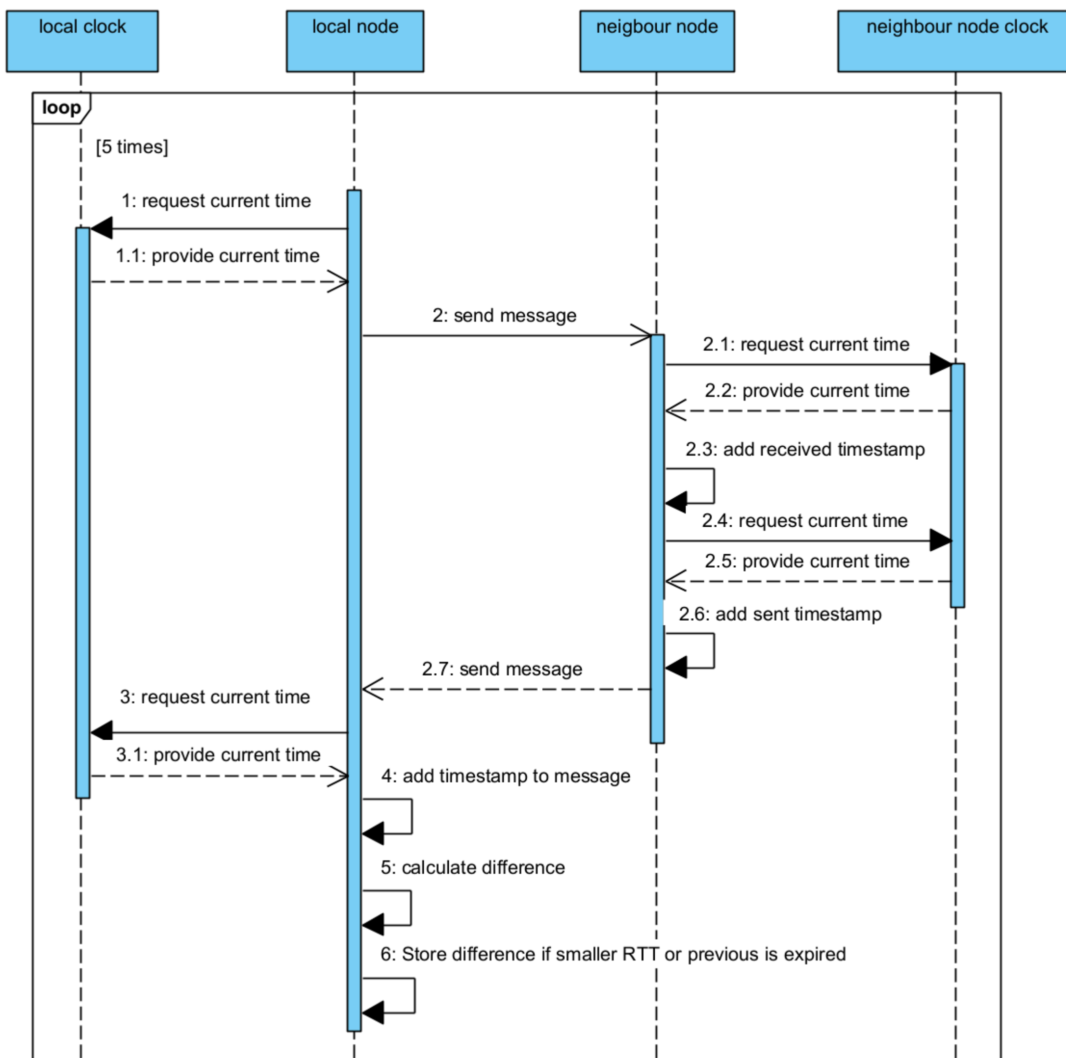


Figure 6: Local measurement system

4.5. Distribution

4.5.1. Expiry time

An expiry time protocol is required to maintain accurate routes while also ensuring the least hops method can be applied. The expiry time is simply a timestamp for when the calculation expires. This expiry time will be 4 times the global time interval. The value was chosen as a balance between correctness, and stability within the system. Changes in the network could make some routes invalid, so there must be an expiry time to ensure that only the most up to date information is used when calculating a route..

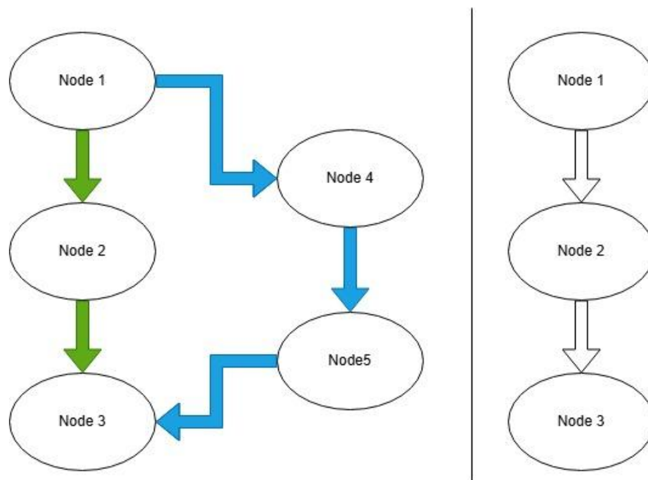


Figure 7: Scenarios to indicate need for expiry time

Consider the scenarios from figure 7, in the first scenario (left) two routes for calculating the time difference between node 1-3 are shown. In this scenario the green route is the one initially used. If node 2 were to exit the network, a different route to node 3 would still exist, but node 1 would not choose the new route since it uses more hops. This issue can be solved if the first route expires and a new route for calculating time difference can take its place.
For the second scenario (right) only one route exists to node 3, and if node 2 were to leave, no alternative route could be made. So eventually the calculated time difference would expire and node 1 would need to resynchronize the time difference with node 3 if it enters the network again. Thus, expiry time is necessary to ensure shorter stale calculation routes can be replaced with current longer calculation routes.

To implement this fully into the system, the system will consider the expiry time only when it receives a higher hop route. Then the system will check the expiry time of the current route, and if expired, replace the current route for the new, higher hop route. If the route it receives has less or equal hops, the system will always replace it, so the most up to date data is always stored.

4.5.2. Message types

The system will contain three message types, each being handled differently by the system. Firstly, a global type which is used to indicate a global distribution message that contains time differences for neighbouring nodes of the sending node. Then there is a 'measure request', this is used to quickly indicate the start of a measurement between neighbours. This also means that this measurement is still on-going. It will have a timestamp added to indicate receival time, and it will be sent with a timestamp to indicate sending time. When another node has received this message, then its type will be changed to 'measure receive'. Upon receival of this type of message, the calculation will be done as indicated in the local time difference calculation.

4.6. Global calculation

The method for calculating global time differences applied is based upon tree-based time synchronization methods, like FTSP (Maróti et al., 2004) and TSTP (Rahamatkar et al., 2009), mentioned in the theoretical background. In our situation the method will be adjusted to fit the given constraints, namely the constraint of not adjusting the actual system clock, and the network being an ad-hoc network. Due to these constraints a traditional tree-based time synchronization with a master node, and only single hop communication would not be entirely appropriate for the problem at hand.

Instead the algorithm is slightly adjusted to a fully distributed system, where each node maintains a list of time difference calculations, based on a least-hop method. This method is used so the error propagation is identical to tree-based algorithms. Instead of selecting a single master node, each node functions like a master node, with the same levels a master node would have in a tree-based algorithm (see figure 8). To communicate time differences, each node only broadcasts the information it has of its neighbours, as each level further adds a step of error propagation, where each receiving node updates its information on time differences according to a system explained in chapter 5.6. In short, the adjusted algorithm functions similar to tree based algorithm as it still uses the least-hop method, but is adapted to be fully distributed due to the given problem specifications.

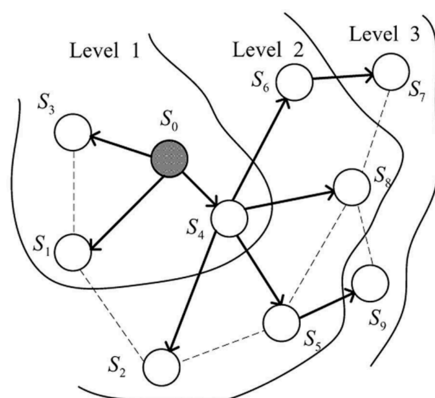


Figure 8: Tree-based network
Taken from "Distributed Clock Synchronization for Wireless Sensor Networks Using Belief Propagation." by Leng, M., & Wu, Y. (2011), p. 2.

4.7. Error propagation

To elaborate on why an adaptation of a tree-based algorithm was chosen, the error propagation will be explained here.

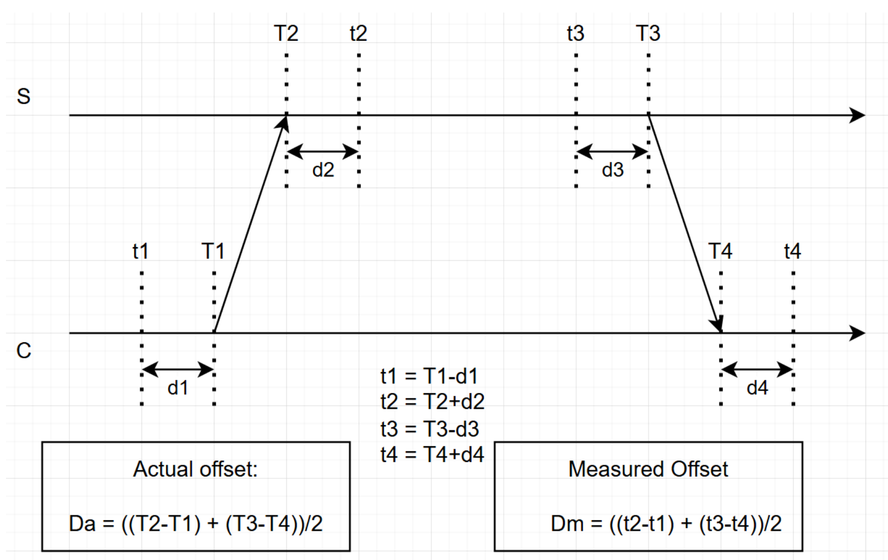


Figure 9: All possible variables in a single measurement

Variables in figure 9:
tX: the timestamp made.
TX: real timestamp.
dX: difference between made timestamp and real timestamp.
Dm: the offset measured from the made timestamps, i.e. the measured offset.
Da: the real offset that would be calculated from the real timestamps, i.e. the actual offset.

The assumption is that each machine is equal to another, as all are made from the same template. As both d1 and d3 are results from delays in sending, and d2 and d4 are results from delay in receiving, we can conclude that d1 and d3 are distributed (roughly) equally and the same goes for d2 and d4. Because of this, the mean of the (not absolute) difference between d1 and d3 is 0 and the same goes for d2 and d4.
Using this, we can get to the following:

$$Dm = \frac{(t_2 - t_1) + (t_3 - t_4)}{2} = \frac{((T_2 + d_2) - (T_1 - d_1) + (T_3 - d_3) - (T_4 + d_4))}{2} = \frac{((T_2 - T_1) + (T_3 - T_4) + d_2 + d_1 - d_4 - d_3)}{2}$$

As was concluded, the mean of the (not absolute) differences of d1 and d3, and that of d4 and d2 are 0. This means that Dm is an unbiased estimator of Da (as the average of d2+d1-d4-d3 is 0). This is true for all nodes, as the assumption was made that each machine is equal. Because of this assumption, all d-values for sending and all for receiving will have the same characteristics. Thus, on long paths in global calculations, the local measurement errors will not constructively interfere, but rather stay centered around 0.

On the other hand however, the variance of this error does increase at each hop, meaning increased chances of large errors. This is minimized with the system's algorithm, as it uses the shortest path between nodes for its calculations. This results in a mean staying centered around 0, while making sure that the variance of these errors is minimized.

The average of the measurement error introduced by RTT is also 0, as increases in RTT can add or subtract to the perceived time difference, based on whether this increase is in the first or second frame sent. This means that this error stays centered around 0 for longer paths, but again, shorter paths ensure lower variance of this error.

4.8. Message formatting

The format of a message mostly follows the ethernet frame format, with one small caveat. In the table below, a typical ethernet frame is shown. In this System, the first 6 bytes of the frame-payload will always be the Eth0 MAC field, thus it is depicted explicitly in the table below. Because this is in the payload, this message is still compatible with the regular ethernet protocol. The eth0 MAC field is used to indicate if a message that is broadcast is meant for the receiving node. When a node sends a measurement frame, it adds its own eth0 MAC address into that field, where a receiving node can confirm whether the frame came from a neighbour or not, using the neighbour list received from the discovery.
The Destination MAC is the bat0 MAC address of the receiver, and source MAC is the bat0 MAC address of the sender. The ether type consists of two bytes, which function like a filter. This protocol uses the **0x88b5** ether type, defined by IEEE 802 as a local experimental ether type (*IEEE 802 Numbers*, n.d.). Frame Check Sequence (FCS) is used to verify the correctness of the frames (i.e corruption), but not handled within this program.

Dest MAC	Source MAC	Ether type	Eth0 MAC (part of payload)	Payload	FCS
6 bytes	6 bytes	2 bytes	6 bytes	Max 1500-6 bytes	4 bytes

Table 5: Message format

4.9. Logging

To be able to see what the system is doing internally logging is required. To have easy access to these logs all logs should be collected in one location. For this the simulator virtual machine is used. This is also the location where simulations are started and stopped so it makes sense to also store the results of these simulations there. All information in a simulator is generated and calculated within separate nodes so these should send their logs to the simulator.

While the simulation is running the received log data is displayed above each node on the web page. This allows users to see what the system is doing in real time. When the simulation is stopped the timeSyncLogs folder is populated. This folder will contain one file for each separate node and one general overview file. In the file for separate nodes the information is stored for debugging. This shows the log messages of every node and what information they are sending to the simulator. The general overview file is meant to see the final results of a simulation. Here the results are explained with text. In this file the accuracy in milliseconds is also calculated. This makes it easy to see how close a node's calculation was to the actual offset of the target node.

5. Software design

To go more into detail as to how the system design will be realised, this chapter describes the design of the software within the system. The software is written in a class based system. Per class, the goal of this class and the main functions are explained. In some cases the functions displayed here are split up into helper functions within the code. This is to keep the code readable.

5.1 Software overview

Before going into detail for each class and function, two diagrams are presented that show the dependencies they have with each other.
Firstly, in figure 10 a class diagram is shown. Classes for the Global and Local calculations/measurements exist. These classes create Message objects as a part of their calculations/measurements, these Message objects are used for Global and Local, depending on their MessageType. These classes also contain a SendReceiver object. This object handles almost all networking actions for the system, except for the logging, which is done by the logSender. This logSender is part of a Global, as only Global calculations are logged. Because of its network-focus, the SendReceiver encodes and decodes Message objects. The Global, Local and SendReceiver all contain a Clock object. This is used for getting the current time with an offset and deviation for that offset added to it.

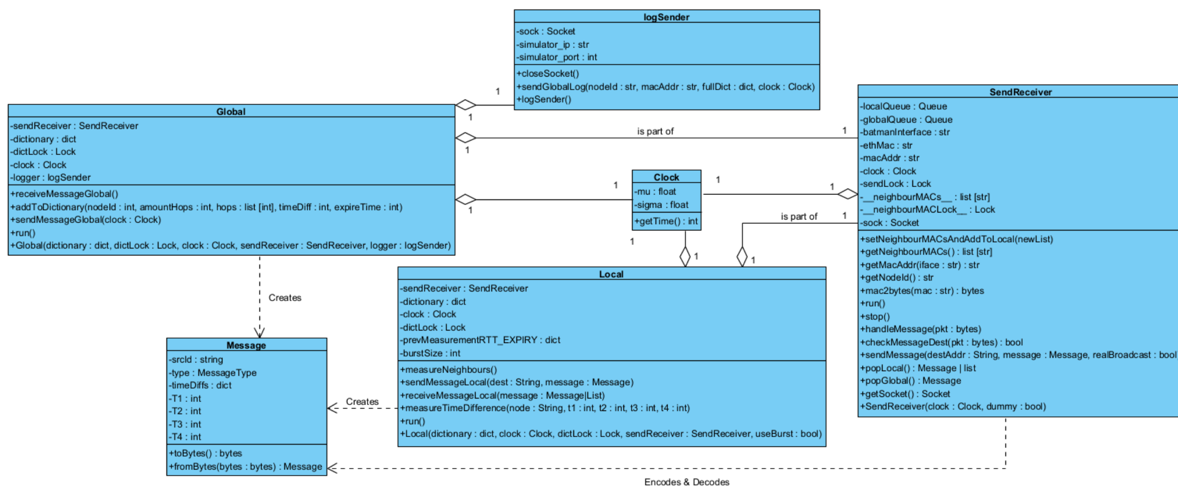


Figure 10: Class diagram for the implementation

Secondly, in figure 10 a diagram showing the relations of files of the program are shown. There is a main file that handles the start and stop of the system. This file also handles the neighbour discovery, using the code from discovery, which it sends to a sendReceiver. The main program ensures that one global, one local, one sendReceiver, one logSender and one Clock object are made. This means that these objects all share the same clock, sendReceiver, etc. The Global class is located in globalCalc, Local in local, SendReceiver in SendReceive, logSender in logSender, Clock in clock and Message in message. Most of these files also reference constants, such as filepaths, interface names and clock offsets, that are defined in the constants file.

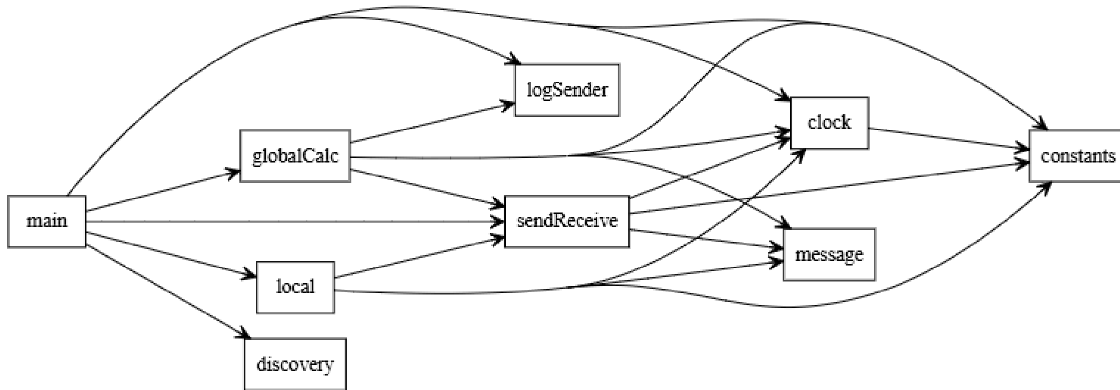


Figure 11: Overview of program files and their dependencies

5.2. Discovery

This part of the system is used for getting the neighbours of a node. This is vital for local measurements between neighbours.

5.2.1. getNeighbourMAC()

This function uses the 'batctl n' command on the BATMAN VMs and parses its result to get the eth0 MAC addresses of the VMs neighbours. This command is part of the openWRT BATMAN-adv system and returns a table of mac addresses, last-seen times and communication interfaces for each neighbour. The getNeighbour MAC function parses this data and thus retrieves the unique MAC addresses of all neighbours. Uniqueness is needed here, as some neighbours may be in this list twice just after simulator initialization.

5.3. Local

For the implementation of this functionality, the following functions will be used

5.3.1. measureNeighbours()

The function that starts the local measurement. This function sends the first local message as a broadcast to the neighbours to measure differences, it is broadcast since the protocol is not aware of the MAC addresses of its neighbours bat0 interfaces. The function allows for burst measurements with a set size *n*. This means that *n* first messages will be sent. The functions below can then be used to get all the timestamps needed to measure the time difference. It uses the sendMessageLocal(Destination, Message) function for the actual sending of the message.

5.3.2. sendMessageLocal(Destination, Message)

When a node wants to send a message, it should make sure to include the correct timestamps and type indicating a measure request or measure receive message. The message is then sent to

the given destination with the general sendMessage function from the sendReceive class as described in chapter 5.4.2.

5.3.3. receiveMessageLocal(Message)

This function is called in a constant loop in a dedicated thread/process that is started when the System is started. In this loop, firstly a message is taken from the queue with received local messages. With this received message from the queue, this function should either do the calculations needed (using the formula (1)) or send the message back based on the type of message. If the message has all four timestamps, this means that it is a message with type 'measure receive' and it will calculate the time difference. If not, the type is 'measure request' and it will have to send the message back to the sender with type 'measure receive' to finish the measurement. When sending a message back, this function should call sendMessageLocal(Destination, message) with the source node of the received message as destination. When the calculations need to be done, the function measureTimeDifference(Node, T1, T2, T3, T4) is called with the source node of the message that was received and T1-T4 being the corresponding time measurements (see figure 5).
After a neighbour discovery is done, the list of neighbours is put into the same queue as the local messages. This receiveMessageLocal function then sends measurement requests when it detects that a list of neighbours is taken from the queue in the current loop iteration. This way, only 1 thread/process is needed for all of the local measurement functionality.

5.3.4. measureTimeDifference(Node, T1, T2, T3, T4)

When this function receives the timestamps, it will calculate the time difference according to formula (1). After it has calculated the time difference with the node, it will determine the expiry time and put the calculated time difference in the dictionary. When using bursts, this function will check the round trip time of the measurements as well, since the time difference with the lowest round trip time should be most accurate and will thus be used.

5.4. sendReceive

This class has a run function which constantly checks if messages are incoming. The following functions are implemented for use cases regarding receiving and sending messages.

5.4.1. handleMessage(Message)

This function handles incoming messages, first determining whether they belong to this protocol or not, as the socket receives all ethernet frames incoming on the bound interface (i.e. any BATMAN message is received).
To filter if the received message belongs to this protocol, the ether type in the message is checked (defined in chapter 4.8), and the Eth0 MAC address is checked. For an incoming message to be intended for the current node, the eth0 MAC address field of this message is either the broadcast MAC address or an element in the neighbouring MAC address list (see chapter 4.2).
If the message is in fact intended for the current node and thus successfully comes through the filter, then it is parsed into a message object. If it is a global type it is added to the queue for global messages.

Once we determine a frame is intended for the current machine, a timestamp of retrieval is made, and the payload is parsed into a message object. The timestamp is then added and the message object is added to the appropriate queue, either local or global.

5.4.2. sendMessage(Destination, Message)

This function encodes a message to bytes and sends it to the destination MAC address. It adds the destination MAC address, source MAC address and ether type to the message all as byte. In addition to the aforementioned fields for an ethernet frame, it adds the eth0 MAC address used, as part of the filter for our protocol, to the first part of the payload.
Right before sending, a timestamp is added to the message, after which it is encoded into bytes and sent over the network.

5.5. Dictionary

The required information we have is an expiry time for each time difference calculation, a time difference, the amount of hops it uses, and each hop used in the calculation. This expiry time will be needed to keep track of the validity of these calculations, based on its age. Each node will have an object with the following structure: a dictionary using another node as key of the dictionary, and it has a tuple as the value. The tuple will contain the hops, expiry time, time difference, and nodes used in the calculation. This data type is the most logical, as the data is best interpreted as key-value pairs.

Node ID/Address	Amount of hops (BATMAN hops)	List of used nodes (hops being the Node ID's/ Addresses)	Time difference	Expiry Time
A	0	[] (an empty list, to represent a neighbour)	15ms	17-04-2026 T16:34:05
B	1	[A]	-30ms	17-04-2026 T16:34:15

Table 6: Example of dictionary structure

Table 6 shows an example dictionary, the Node ID or address is the key of the dictionary, used to look-up the other values linked to it. While it is displayed more like a database, a dictionary only holds a single value for a key, so this value will be a python tuple, in which all four values will be stored.
The values per entry are:

- The amount of hops used in the calculation, where, in order to make checking for neighbours in the software more readable, we start at 0 for neighbours.
- A list of nodes used, each node in the list was used to calculate the time difference.
- The time difference with that node and this node.
- The Expiry time, which indicates the maximum use date and time.

5.6. Global calculation

In the global calculation class, there are two important functions. Firstly the receiveMessageGlobal(Message) and secondly the SendMessageGlobal(Clock). The receiving of messages is done on a different thread than the sending of messages, to allow the sending to happen per-iteration, while the receiving can be event-driven (where the 'event' is receiving a global message). The receiving of messages is dealt with using a message queue (see chapter 5.4.1). The receiving messages thread is constantly checking if there are any messages in the global queue, if this is the case, then it calls receiveMessageGlobal(Message). The sending messages thread calls sendMessageGlobal(Clock) and should stay in this method until the simulation is stopped.

5.6.1. receiveMessageGlobal(Message)

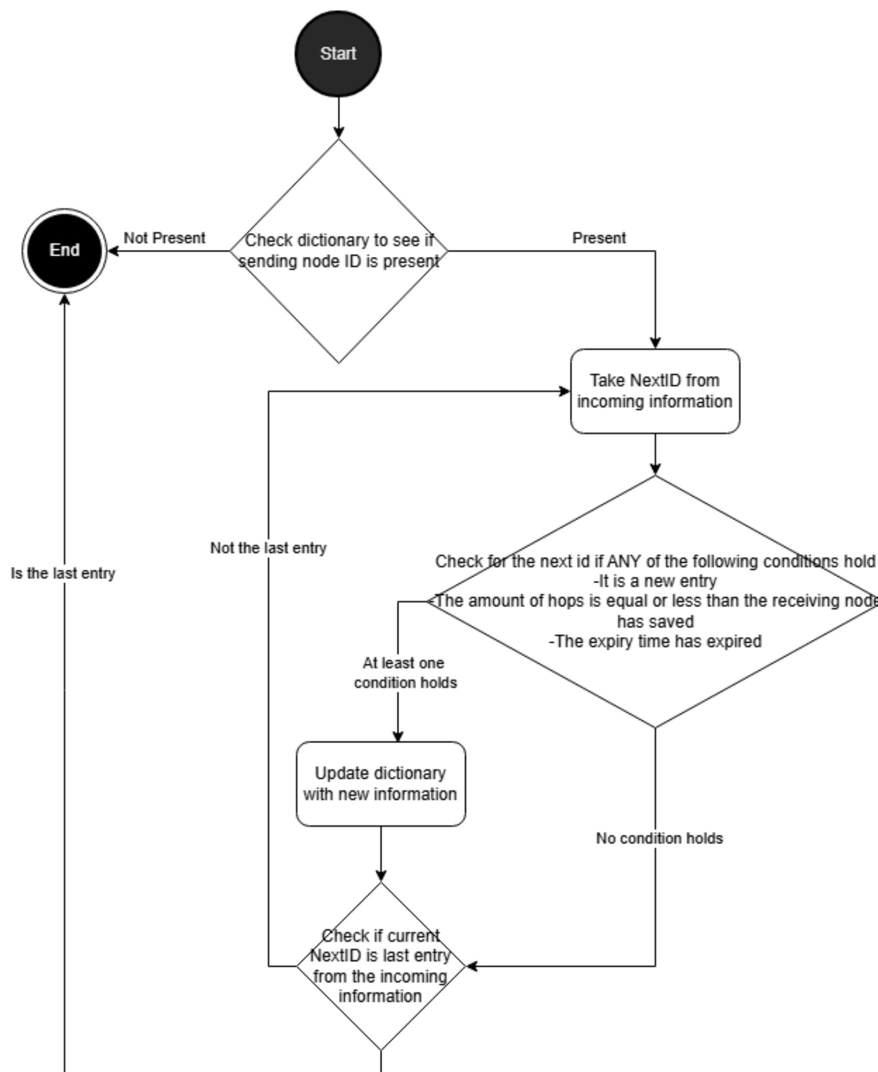


Figure 12: Activity diagram for incoming dictionaries

This function will save the incoming information from a node into its own dictionary. The incoming message will contain the local measurements that the sender node has of its

neighbours. The method of how this is being done is described in figure 12 and will be demonstrated in an example as well.

For example node 1 receives the following information from node 2 :

Node	Time difference
3	106
4	210

Table 7: Example of received information

In table 7 above, the message information is shown. When node 1 receives this information from node 2, it will compare this information to the information it already has. This information in saved in a dictionary and could be the following:

Node ID/Address	Amount of hops (BATMAN hops)	List of used nodes (hops being the Node ID's/ Addresses)	Time difference	Expiry Time
2	0	[]	110	17-04-2026 T16:34:05
3	3	[6,5,4]	309	17-04-2026 T16:34:05

Table 8: Dictionary of node 1 before information was added

After the message is received, the following things are checked:

1. Is the incoming information new information?
2. Does the incoming information use a route with less or equal amount of hops?
3. Has the expiry time of my information on this node passed?

When any of these are true, the current information will be replaced with the information that is received. For example, in this case node 1 receives information from node 2 about node 3 and node 4. The information about node 4 will directly be added because there is no information about this node yet in node 1 its dictionary. If 1 was not in the dictionary, 4 would not be added, because then unusable data would be stored.
For the information about node 3, firstly the amount of hops will be checked. Every node only sends information about their neighbours, therefore the amount of hops can be calculated by taking the amount of hops needed to go to node 2 which is 0 and add 1 to this. In node 1's current dictionary, it shows that it only knows a route which uses 3 hops, which means node 1 will replace this information with the new incoming information.
If there was any information in node 1's current dictionary from which the expiry time was passed (meaning that the current time is past the expiry time), then this information will be

replaced with new information even if the amount of hops is more. This is to favour current data with more hops over stale data with less.

Afterwards this information will be as follows:

Node ID/Address	Amount of hops (BATMAN hops)	List of used nodes (hops being the Node ID's/ Addresses)	Time difference	Expiry Time
2	0	[]	110	17-04-2026 T16:34:05
3	1	[2]	206 (which is 110 + 106)	Current time + 3 * the time interval at which information is sent
4	1	[2]	320 (which is 110 +210)	Current time + 3 * the time interval at which information is sent

Table 9: Dictionary of node 1 after adding new information

5.6.2. SendMessageGlobal(Clock)

This function will ensure that the local measurements that are still up to date (thus those of which the expiry time has not expired yet) are broadcasted to all other nodes in the network. This will be done periodically in every iteration of the System, at the moment every three seconds (this is also true for the local measurements). This function will run on a separate thread/process from the other parts of the system, because it has to wait for these iterations to start and end, which can most easily be implemented as such.

5.7. Message payloads

Within the system all messages conform to one of two different formats, one for local and one for global messages.

5.7.1 Local measurement format

```
payload = {
    "srcId" = "12:23:34:45:56:67",
    "type" = MEASURE_RECEIVE,
    "T1" = 100,
    "T2" = 200,
    "T3" = 250
}
```

Figure 13: Payload local message format

This packet-format is like the TWAMP format of a TWAMP packet, however there is a lot of extra information that is not within our format.

Entry descriptions:
srcId: this is the source MAC Address. It is used for checking if a local measurement is coming from a neighbour
Type: This contains the type of the message ('measure request' or 'measure receive') for local measurements)
TX: these are the timestamps for the measurement, and will be received using python's time.time() function. This function has at least millisecond accuracy on OpenWRT(*Chapter 8. KVM Guest Timing Management | Red Hat Documentation*, z.d.),, and thus will be usable for this purpose. First send will only contain T1 and last send will contain T1-T3. T4 is not needed in the packet, as this is never sent.

5.8.1 Global distribution format

```
payload = {
		"srcId" = “12:23:34:45:56:67”,
		"type" = GLOBAL,
		"1234512345" = 10,
		"2345623456" = -1
}
```

Figure 14: Payload global message format

This is an example payload for 2 time differences that are distributed. The message contains a sourceID and Type, but instead of Timestamps, it contains MAC addresses. These MAC addresses are accompanied with the time-offset measured in the local measurements.

Entry descriptions:
srcId: this is the source MAC Address. It should always contain a broadcast MAC Address (ff:ff:ff:ff:ff:ff) for global distributions.
Type: This contains the type of the message (GLOBAL for global distributions)
MAC_ADDR (1234512345 and 2345623456 in this example): these are the bat0 addresses of the neighbours for which the local measurements are shared by the sender. The value belonging to a MAC_ADDR entry in the payload contains this offset. This offset is the amount of time to add to the sender’s clock to get the time of the neighbour’s clock.

5.9. Logging

For the logging two separate systems are needed, one on every BATMAN VM and one on the simulator. The system on the node can simply be called by running threads when needed. For the system on the simulator a new thread should be started, this thread can then constantly listen for logging messages.

5.9.1. logSender

The logSender is a class located in the BATMAN VM . This class contains two simple functions. The first function simply closes the socket through which the messages are sent. This is needed to ensure no sockets are left open on exit of the program. The second function can send messages to the simulator. This function takes the node id, MAC address, dictionary (containing the calculated data) and the clock class. When called, this function sends a message containing the input data with json format.

5.9.1. Receiving logs

In the simulator the receiveLogs class is used. When a simulation is started in the server.py, a separate thread is started. In this thread, the log_socket function loops as long as the simulation is being run. This thread uses a socket with the corresponding port and ip to receive the messages of all nodes in the simulation. Whenever a message is received the information is written to the corresponding file for that node. When the simulation is stopped the newest data is used to create a general overview message. In this message the accuracy is also calculated. This is done simply subtracting the calculated offset of a node with its actual offset in the clock.

5.9.2 Webpage logs display

While the simulation is running, the receiveLogs class stores the most recent information. This information is then passed to the javascript through an app route. In the main.js (the javascript code) this data is gotten through a fetch request. When the processUpdate function is called another function, UpdateShipLogTexts is called with the log data. This data is then displayed for each node in the network. To allow users to toggle the log data, a button is added in the html. When this button is pressed the visibility of the log data is set to none.

5.10. Message

This class is used to pass received messages within the system, and to decode and encode received messages to and from bytes.

5.10.1 messageType Enum

An enum to indicate the different message types as described in the system design (4.5.2).

5.10.2 toBytes

This function is used to encode a Message object to bytes. This is done via a JSON object, where the relevant data, which differs depending on the message type, is entered into a JSON object, which is then encoded to bytes.

5.10.3 fromBytes

This function is used to decode a byte array into a Message object. First, the array is decoded back to the JSON object. Then depending on the message type, each applicable field is added to a new Message object.

6. Miscellaneous development activities

While implementing the system, some things that are not within the system design needed to be made. These are mostly solutions to problems that appeared while making and testing the system, but some other implementations were also needed.

6.1. Start and Stop

Since the simulation can be started and stopped multiple times it is not possible to simply add a startup script on the batman vm. The code on every batman vm should be started when the user presses the start simulation button, and similarly the code should stop when the stop simulation button is pressed. To achieve this, a ssh connection with every batman vm node is made. Two lists are created, runOnStart and runOnEnd. In these lists commands can be added. Every command in this list will be executed on every batman vm when the start or stop simulation button is pressed.

6.2. Simulator Problems

While working on the simulator, some issues with the existing code appeared. Two problems were caused by the fact that the code on the physical machine was originally made on a different machine. The api key and the template id were incorrect, to solve this a new api key had to be made and the values in the constants.py were updated accordingly.
The other issue was that the neighbour list in the batman vm did not correctly update. This caused our system to not see new neighbours when they appeared. To fix this, a student that worked on this system before was contacted, after which this bug was removed from the simulator code.

6.3. NTP

An issue that was encountered with the simulator is that upon startup, all machines had large time offsets. These showed up in the measurements and made determining accuracy difficult. To solve this issue an NTP protocol was applied. Initially the built-in system on OpenWRT was used, NTPD, and while this made sure the mesh nodes synchronized their time to the simulator machine, it took several minutes before fully synchronized. As the machines are started immediately before use, this is less than ideal.
So the final solution was to use 'chrony', a package on linux implementing NTP, but one that allows for more customization. A startup script is executed that synchronizes the system via a burst of measurements with the simulator, quickly synchronizing the node clock with the simulator clock. According to chrony, local time alignment accuracy is within 2 micro seconds (*Chrony – Configuration Examples And Accuracy*, n.d.). As our protocol works within milliseconds this is sufficient for our purposes.

7. Testing

To reduce the chance of errors during integration testing methods are used. For this project, mainly unit tests are used, some system tests are also used for the parts of the system that can not be verified with unit tests. The python unit test library is used for the unit tests. This makes it easy to test check all unit tests with a single command whenever any code is changed.

7.1. Unit testing

To ensure that our code works as we expect, we have written unit tests for as many classes as possible. Some functions/classes proved to be difficult to test with unit tests. This is the reason some classes have been tested in other ways, for example by doing system tests. The unit tests will be explained per test class.

7.1.1. testClock

This class has two tests. The goal of these tests is to test if the clock works like a clock should. The first one creates a new Clock with mu equal to 10 and sigma equal to 0. After this, it will check if the real time is the same as the time given by the Clock that was made. The second test also creates a new Clock with mu equal to 10 and sigma equal to 0, then gets the time and finally checks that the time is still the same after a few milliseconds have passed. The sigma is not tested as sigma is not deterministic, therefore making it hard to test. Additionally the function that uses the sigma is a built-in function, we assume it functions as expected.

7.1.2. testDiscovery

This class tests the parsing of the 'batctl n'-command output that is done by getNeighbourMAC. It has 2 cases, one for valid output with a duplicate neighbour, and one for an error message. The test showed that erroneous output is correctly handled by the parser. It also showed that valid output is handled correctly and duplicates are removed as needed.

7.1.3. testGlobal

This class tests the receiving of messages. The class should put the information out of the message into the dictionary when needed (see figure 12). In every class, the test is written from the perspective of node 1. All nodes within the tests (except the first test) already have the information of their neighbours (from the localCalc), because otherwise the information will not be saved. The global calc will only be called after receiving information from a node within the network that knows its time difference with its neighbours. The testGlobal class consists of four tests with different topologies.

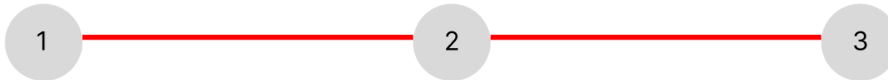


Figure 15: Test scenario three in a row

The first test checks the case for three nodes in a row (figure 15). This is the only test where the dictionary of node 1 is empty. This is because this test should show that any new information will not be saved when it comes from a node that is not known to the receiving node (thus not in its dictionary).

The second test checks the case above as well (figure 15). This will test if the new global information is saved in the dictionary. Node 1 already has the information (time difference) from node 2 . Node 2 has the information from node 3. In this case, node 2 sends its information (its time difference with node 3) to node 1. Node 1 should update its dictionary by adding node 3. It also adds the information that belongs with node 3. There is 1 hop between node 1 and node 3, the hoplist should contain node 2, and the time difference should be equal to the total time difference between node 1 and node 3.

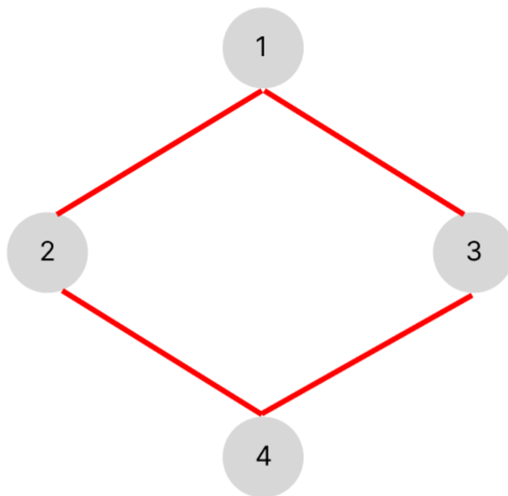


Figure 16: Test scenario diamond

The third test (figure 16) is made to test whether node 1 overwrites new information when it gets information from a new node with the same amount of hops. Node 1 should overwrite the time differences with nodes that it already has if the hops are equal or less than the hops corresponding to the information in the dictionary. In the test, node 1 first receives a message from node 2 which contains the time difference between node 2 and 4. After receiving this, the dictionary of node 1 should contain the information for node 4: 1 hop, hoplist containing node 2 and the time difference between 1 and 4 using node 2 as a hop. Then, node 3 sends the time difference between 3 and 4. With this information, node 1 should update their information about node 4 to: 1 hop, hoplist containing node 3 and the time difference between node 1 and node 4 using node 3 as a hop.

Figure 17: Test scenario four in a row

The last test (figure 17) tests whether calculating time difference works with multiple hops. In this case node 2 first sends their time difference with node 3 to node 1. After this, it is tested that this information is correctly saved in the dictionary. The dictionary should have added node 3 as a key with 1 hop, a hoplist containing node 2, and the time difference between node 1 and node 3 using node 2 as a hop. After this, node 1 receives information from node 3 containing information about node 4 and node 2. The information about node 2 should not replace what node 1 has already saved (this is tested), because the incoming route to node 2 has more hops. The information about node 4 however, should be saved by node 1 in their dictionary. It should include: 2 hops, a hoplist containing node 2 and 3, and the corresponding time difference between node 1 and node 4.

7.1.4. testLocal

It is difficult to test in a unit test whether or not messages have been received on different nodes. This is why it is not tested within the unit tests. Something that however is being tested is the receiving of the 'measure receive' message (the message that contains all of the timestamps) on one node. The message is handed to the appropriate method, receiveMessageLocal. After this, it checks whether the information in the dictionary corresponds to the correct time difference according to formula (1).

7.1.5. testMessage

This class tests whether the translation from bytes to a message and the other way around works as it should. The toBytes function and the fromBytes function are tested in the following way.
There is a function to check if two message objects are equal, which is used to check after using fromBytes and toBytes to generate an additional message object if the messages are still equal.
Firstly, a test for both local message types, where the local message is made just how it would be made in a real simulation. After making a copy using fromBytes and toBytes we compare both objects, confirming that they are still equal. Then we change a single value in one message, and confirm they are no longer equal. The latter is done to ensure it fails if not equal.
To test global messages, a similar process is followed, we create a message object as it would be created in a real scenario, then copy it using toBytes and fromBytes, and we ensure equality.

7.1.6. testSendReceive

This class tests whether actually sending the messages over sockets still results in the same message being received on the receiving end. To make it possible to test on a single machine, two sockets are created where one socket sends the message to the other socket, upon receiving we check if the incoming message is identical to the sent message. This can only be tested on the BATMAN machines, as the sockets have to be built on linux and bind to specific interfaces only available on a BATMAN machine.

7.2. System testing

Some parts of the system can not be tested with unit tests. These parts are mainly the connection between multiple nodes, the connection between the simulator and the nodes in the simulation and the accuracy of the running system itself. For these last system tests the tester must run the complete system to see how it performs.

7.2.1. simConnectionTest

In this file two separate capabilities are tested. The first one is the opening and closing off a program through the simulator. The second one is sending log messages to the simulator.

For opening and closing off a program the system test should be performed as follows:

- In the DSM/main.py file in the simulator, the run on start and stop lists (at the top of the file) should contain the following commands:

```
RunOnStart = ['cd /root/TimeSync/timesynchmaritimemanet && /usr/bin/python3
simConnectionTest.py & echo $! > /tmp/simConnectionTest.pid']
RunOnEnd = ['kill -TERM $(cat /tmp/simConnectionTest.pid)']
```

Figure 17: Start up commands

- The simulator should be started as done normally, then at least one boat should be added to the simulator. Afterwards, the simulation can be started.
- After waiting for ~30 seconds the simulation can be stopped.
- Now the mesh node should be opened, then in the terminal run the following command:

"cat TimeSync/timesynchmaritimemanet/testfile.txt"

- This should show the contents of a file. If the system worked the file contains both "program started" and "program exited".

To test the logging all the steps above should be run in the same way (it is possible to test both at the same time). After running the simulation and stopping it, the folder timeSyncLogs should contain a file, in this file information about node 1 should be visible.
With these two tests the logging system is verified to work.

7.3. General setup for testing with the simulator

To run the following system test, the same starting setup must be performed as for the test above. For this reason it is not mentioned in great detail again but it is mentioned in short.
To start the system, firstly the readme in the simulator code should be followed to start the system. After this a configuration in the web page can be loaded. A configuration for every test below is stored in the following file directory: downloads/timeSyncTests. After loading the configuration the simulation can be started. While the simulation is running the known offsets of every node are visible on the top of the nodes. After waiting for the system to properly initialize and propagate, the simulation can be stopped. After stopping the simulation a file will appear with important data. A file called all_nodes_log.txt will appear in the following location: /home/maritimemanet/digital_twin-main/timeSyncLogs. In this file the accuracy of every node can be found. A corresponding image of the complete system for every system test will be shown in the appendix.

7.3.1. Local nodes

The first system test is to see if the local communication with nodes works. A simple system with two neighbouring nodes is enough for this. After loading the system, running it and stopping after the system was completely set up. The following results are expected within the *all nodes* log:

```
for every node the accuracy of their calculated offset:
node id: 2 has calculated:
for node node id: 1: calculated offset: 317, actual offset: 319 thus difference = 2
node id: 1 has calculated:
for node node id: 2: calculated offset: -320, actual offset: -319 thus difference = 1
```

Figure 18: Expected results for the local nodes test

Here it is shown that the calculated offsets are correct with some margin for error.

7.3.2. Shortest route

The goal of this system test is to see if a node uses the shortest route that is available. For this, a triangle of nodes is used. In this example, node 1 could reach node 3 simply as a neighbour or via node 2 and then to node 3 (see appendix A.2) . To check if the test has the expected results, in the same directory as the *all nodes* log a log file exists named node_1.txt. The last line of this file is relevant:

```
node addr: node id: 1 actual time offset: 149
dict: {'node id: 2': (0, [], -292, 1775810568756), 'node id: 3': (0, [], -150, 1775810562746)}
```

Figure 19: Expected results for the shortest route test

For both node 2 and node 3 it is visible that the array (the [] next to the node id) is empty. This means that node one made a direct connection and thus took the shortest route.

7.3.3. Nodes in a line

The goal of this system test is to see if the system can handle multi hop calculation. The reason why seven nodes are used is to also check the accuracy of the network when there are a lot of hops. This accuracy will be discussed more in the results (see chapter 8). The expected results for this test should look similar to the following:

```
node id: 1 has calculated:
for node node id: 2: calculated offset: -107, actual offset: -104 thus difference = 3
for node node id: 3: calculated offset: -113, actual offset: -117 thus difference = -4
for node node id: 4: calculated offset: 55, actual offset: 50 thus difference = -5
for node node id: 5: calculated offset: -131, actual offset: -136 thus difference = -5
for node node id: 6: calculated offset: 111, actual offset: 109 thus difference = -2
for node node id: 7: calculated offset: 155, actual offset: 152 thus difference = -3
```

Figure 20: Expected results for the nodes in a line test

The important part for this test is that every node can reach every node in the system. This shows that the system also works with non neighbouring nodes.

7.3.4. Node entering the network

The goal of this system test is to check if, when a new node arrives in the network, this node is correctly added to the network and all data is correctly communicated. After loading the configuration, node 3 should move from its starting position to the rest of the network on the right (see appendix A.5). After node 3 has arrived within the network. All other nodes should get a calculated offset for node 3, and node 3 should get the offset of every other node.
To easily check the expected result the log information on top of every node can be used (this can be toggled with the show logs button). If this is not possible for some reason, it is still possible to look at the logs in a similar manner as with the shortest route test.
Expected outcome after node enters the network:

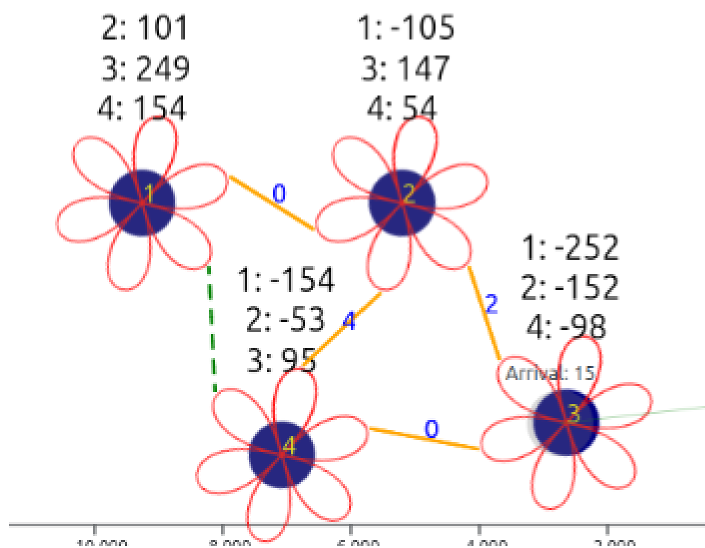


Figure 21: Expected results for the node entering the network test

Here it is visible that every node in the network knows the relevant information of the node that entered the system later on.

7.3.5. Node leaving the network

The goal of this system test is to check if, when a node leaves, the network forgets this node. This test is similar to the previous test. The difference here is that node 3 starts within the network, and after some time leaves said network. In this test it is expected that when the node has left the network, all other nodes forget the offset of node 3 and node 3 forgets all offsets of other nodes.

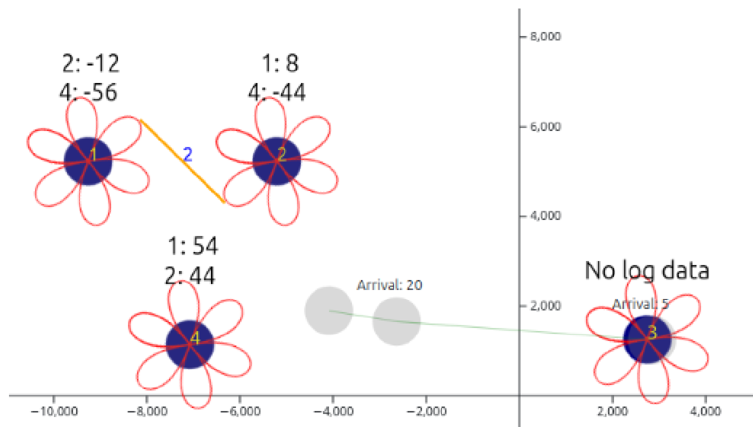


Figure 22: Expected results for node leaving the network test

Here it is shown that all nodes in the network forgot the offset of node 3 and that node 3 itself also forgot all of the data of other nodes.

7.3.6. New shortest route appears

The system can correctly handle nodes entering and leaving the network. However it has not been tested yet if, when the network changes and a new shorter route appears, this shorter route is also used. For this test a new node will enter the system in a location to create a shorter possible route (see appendix A.7). To check the results, similar to the shortest route test the log results of one singular node are relevant.
To see the expected results the file node_1.txt should be opened:

```
, 'node id: 3': (3, ['node id: 4', 'node id: 5', 'node id: 6'], 12, 1775923016284)}

, 'node id: 3': (1, ['node id: 2'], 14, 1775923019274), 'node id: 2': (0, [], -90, 1775923019252)}
```

Figure 23: Expected results for new shortest route appears test

Here it is clear to see that the route to node 3 is initially three hops long, but when node two moves into the network this route changes to only have node 2 in the route. Depending on how long the simulation was running this file might be quite large. To easily check if this test performed correctly the last entry in the file is relevant. Here the route to node 3 should only contain node 2 in it.

7.3.7. Shortest route disappears

When the shortest route disappears, the system should replace this with a new route, even if this route is worse than the route that disappeared. To test this a setup similar to the previous test is used, only here node 2 will leave the network (see appendix A.8). To see the expected result the file node_1.txt should be opened:

```
dict: {'node id: 3': (3, ['node id: 4', 'node id: 5', 'node id: 6'],
```

Figure 24: Expected results for shortest route disappears test

On the last line of this file, the route to node 3 should contain the route in the image above, this shows that node 1 changed its route to not use the node that left the system even though that route had less hops.

8. Results

8.1. Correctness

From the System and Unit tests, it can be concluded that the algorithm is correct, in the sense that it achieves functionally correct results in static, dynamic, single-hop and multi-hop networks. This proves that features of the algorithm, such as the local measurements, usage of expiry times and shortest-path based global-calculations, were suitable for the system.

8.2. Performance

Considering the part of this report on Error Propagation, the algorithm works as expected. The tests show that offset-errors are centered around 0, regardless of the size of the test-network. The tests also show, as expected, that the variance of these errors increases with network size, with larger networks showing larger errors than smaller networks. Still, these errors stay centered around 0.
It is hard to make exact conclusions about the quality of the algorithm in relation to others, as no other algorithm was tested. Comparing the algorithm to the FTSP algorithm (Maróti et al., 2004), as was done in this report before, the FSTP algorithm performs with a measurement error in the order of microseconds. In the largest system test of this project, an average measurement error on the order of milliseconds can be seen in a 6-hop network. However this difference does not necessarily imply that the algorithm of this project is of very low quality compared to the FSTP algorithm.

This difference could be caused by inaccuracy of the clock on the simulator. According to red hat (*Chapter 8. KVM Guest Timing Management | Red Hat Documentation*, z.d.) the accuracy of the clock on a linux VM can decrease when there are multiple VM's in the system, or if the VM is under heavy load. In the simulator tests, there are multiple VMs and every VM is under 100% load, thus this can be a good reason for why the clocks in the simulator are not accurate to microseconds. At the moment, the clock readings are rounded to milliseconds. The system was tested without this rounding, however this did not change the accuracy of the results.

Currently, the measurement accuracy is within the milliseconds range, however it should be tested if this is the limit of this algorithm or if this is caused by inaccuracies in the simulator environment. For this reason, the simulator should be changed/made more efficient to ensure that the VMs are not overloaded while the system is running.

8.3 Deviation

In the clock function, there is an option to change the sigma. This value is used for the gaussian distribution of the clock offset. The sigma determines how much deviation there is in time readings. All the above tests were performed with the sigma at zero. To see how the sigma affects the accuracy of the algorithm multiple tests were performed. These tests were similar to the nodes in a line system test, but with the sigma value changed to a higher value. In the graph (see appendix B) the average accuracy of all the tested values of sigma are shown. For this test the sigma values zero, five and ten.

9. Conclusion

In this project, a protocol for time alignment in a GNSS-denied MANET was designed, implemented and tested. Firstly, existing protocols and useful tools were researched. Then an iterative approach to this design was used, with client-feedback rounds. After this, implementation and testing followed quickly.
Most requirements were completed in the project, with only the requirement on testing multiple protocols being not completed.
Considering the correctness of the algorithm, the project was a success, as the algorithm works the way it is supposed to. This conclusion follows from it passing the many system tests it was tested on.
Considering the quality of the algorithm, results were not necessarily better than existing protocols such as FTSP (Maróti et al., 2004). A possible cause for this was identified, that being clock inaccuracy. This cause is an interesting opportunity for testing and/or solving in the future, but for now inhibits making good conclusions for comparisons with other algorithms.
Additionally, the influence of noise on clock-reading was tested. These tests resulted in the algorithm performing worse in cases of more noise, which was as expected.

10. Future work

There are still some improvements that could be made to the current solution of the time alignment within a MANET. These improvements are outside of the scope of this project within this module. This is why these possible future improvements are denoted as follows.

Firstly, within this project, only one protocol is produced and tested. There is a possibility that another protocol works better which is why this should be researched. This way, another protocol might be found that proves to be more efficient and accurate, or the implemented protocol can be improved.

Secondly, the simulator on which the implementation was tested, did not allow for many nodes (e.g. 30 nodes would either not work or take a really long time to simulate) and it did not work fastly which is why testing took quite some time. This was not ideal. This is why it might be worth investing in a simulator that can work with larger quantities of nodes, or making the current simulator more efficient. There is a chance that the current code/implementation which runs the simulator can be improved to require less resources. This can for example be done by translating (parts of) the code to a faster programming language (for example compiled languages such as C/C++, Rust, Go, etc.), or combining the BATMAN VM and EE VM into 1 client VM running on OpenWRT. This last change is possible, as the EE VM is not needed for client-side applications, if the BATMAN VM was given more computing power. If such changes work, tests can be a lot faster and they can be made for more realistic scenarios and scenarios where there are a large number of nodes.

Lastly, the system could implement more measurements for detecting and discarding/weighing down outlier measurements. This is already partially done by measuring the local time difference in bursts and choosing the time difference with the smallest round trip time. But this could be improved by using outlier detection, by for example using a normal distribution and ignoring/deleting the measurements at the ends of the curve.

11. Reflection

11.1. Improvements

After literature research and finishing the project proposal, we spent a lot of time on the design of our system. This time was well spent at first but after a week, it might have been better for us to start coding. After being done with the coding, we had to change a decent amount of the original design report. Because of this, we lost some time on perfecting things in the original design that we replaced in the end. It would have been more useful to make the design report, then directly implement what we had designed (with some feedback in mind) and afterwards improve the design report and implementation iteratively. We should have spoken/discussed more with our client on where our priorities were.

11.2. Accomplishments

During the project, we got the code working in a short period of time. This was accomplished because of the teamwork and team division. Every person in the team had their own task and made great progress, while they kept communicating about any uncertainties. This way, everyone was updated on what was happening and how parts work together in every part of the process.

12. Individual Contributions

Within the team we made the decision to give everyone their own task and then combine these while also keeping each other updated when things change or go wrong. The following is the division of the tasks per person:

Task	**Person/ people**
Design Report for University	Everyone
Design Report for Client	Everyone
Project Proposal	Everyone
Peer review & Final presentations	Floortje & Ryan (respectively)
Discovery (including obsolete C implementation)	Ryan
Local	Iris
Global	Floortje
Networking	Kevin
Start & Stop of the program, Logging & Visualization	Lars
Simulator Bug fixing	Ryan & Lars
NTP	Kevin
Contact Person	Ryan

Table 10: Division of tasks

13. Usage of AI

During the preparation of this work the author(s) used generative AI in order to assist in finding sources for academic papers. After using this tool/service, the author(s) reviewed and edited the content as needed and take(s) full responsibility for the content of the work

14. References

About Thales. (n.d.). Thales. https://www.thalesgroup.com/en/about-thales

Chapter 8. KVM Guest Timing Management | Virtualization Deployment and Administration Guide | Red Hat Enterprise Linux | 7 | Red Hat Documentation. (z.d.). *https://docs.redhat.com/en/documentation/red_hat_enterprise_linux/7/html/virtualization_deployment_and_administration_guide/chap-kvm_guest_timing_management*

chrony – Configuration examples and accuracy. (n.d.). https://chrony-project.org/examples.html

Clocks and Timers — The Linux Kernel documentation. (n.d.). https://docs.kernel.org/virt/hyperv/clocks.html

Hebbink, J. (2025). *Optimizing Multicast in MaritimeManet: Diagnosing Communication Failures and Evaluating BATMAN-adv Optimizations*.

Hedayat, K., Krzanowski, R., Morton, A., Yum, K., & Babiarz, J. (2008). *A two-way active measurement protocol (TWAMP)*. https://doi.org/10.17487/rfc5357

IEEE 802 Numbers. (n.d.). https://www.iana.org/assignments/ieee-802-numbers/ieee-802-numbers.xhtml

Leng, M., & Wu, Y. (2011). Distributed Clock Synchronization for Wireless Sensor Networks Using Belief Propagation. *IEEE Transactions On Signal Processing*, *59*(11), 5404–5414. https://doi.org/10.1109/tsp.2011.2162832

Maróti, M., Kusy, B., Simon, G., & Lédeczi, Á. (2004). The Flooding Time Synchronization Protocol. *SenSys'04 - Proceedings of the Second International Conference on Embedded Networked Sensor Systems*, 39–49. https://doi.org/10.1145/1031495.1031501

Rahamatkar, S., Agarwal, A., Sharma, V., & Gupta, P. (2009). *Tree structured time synchronization protocol in wireless sensor network*.

Thales. (n.d.). *MaritimeManet: Broadband mobile ad-hoc network for Naval collaborative combat* [Slide show].

Van Steen, M., & Tanenbaum, A. S. (2023). *Distributed Systems* (4th ed.).

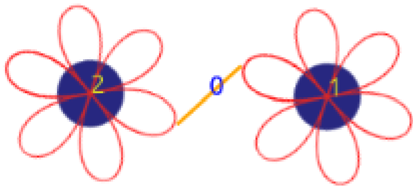
Vouzis, P. (2025, May 14). *A Deep Dive into Two-Way Active Measurement Protocol (TWAMP)*. Netbeez.net. https://netbeez.net/blog/twamp/

Wisotzky, S. (n.d.). *Twampy documentation*. Github. Retrieved 14 April 2026, from https://nokia.github.io/twampy/

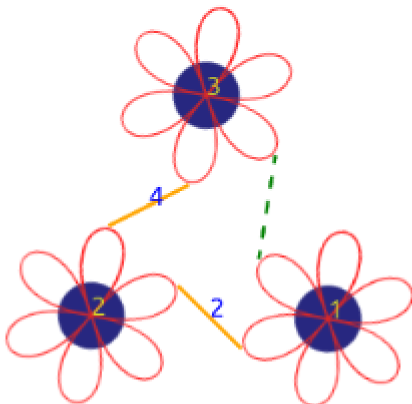
15. Appendix

A. System tests

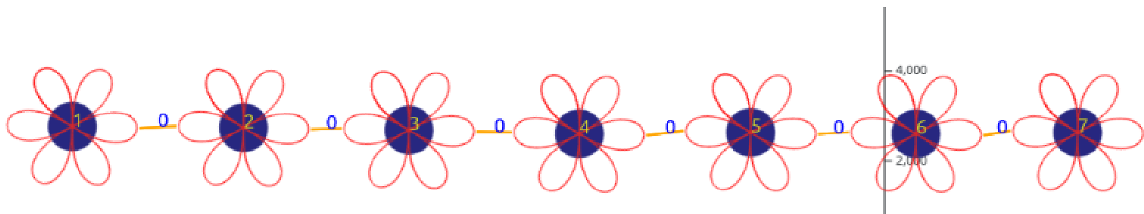
1. Local nodes



2. Shortest path



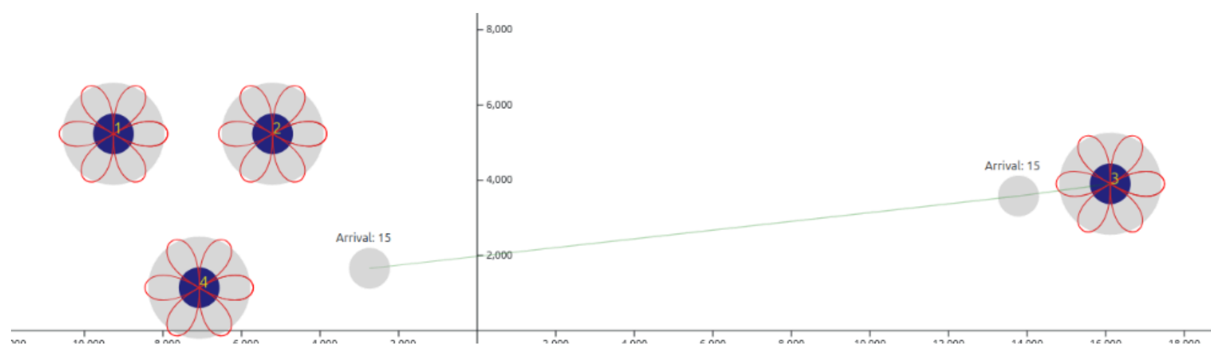
3. Nodes in a line



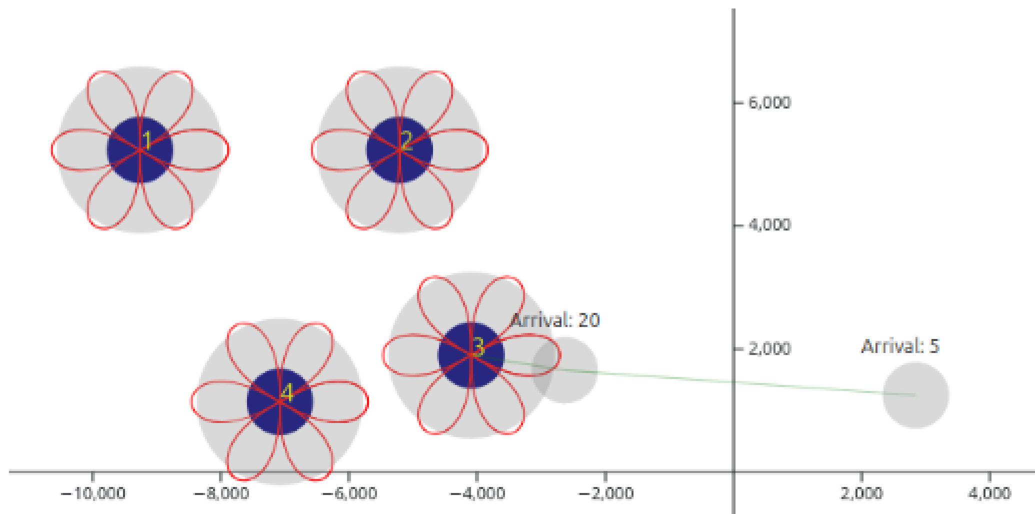
4. Accuracy result for nodes in a line

```
for every node the accuracy of their calculated offset:
node id: 6 has calculated:
for node node id: 5: calculated offset: -244, actual offset: -245 thus difference = -1
for node node id: 4: calculated offset: -60, actual offset: -59 thus difference = 1
for node node id: 7: calculated offset: 44, actual offset: 43 thus difference = -1
for node node id: 3: calculated offset: -229, actual offset: -226 thus difference = 3
for node node id: 2: calculated offset: -218, actual offset: -213 thus difference = 5
for node node id: 1: calculated offset: -112, actual offset: -109 thus difference = 3
node id: 5 has calculated:
for node node id: 4: calculated offset: 184, actual offset: 186 thus difference = 2
for node node id: 6: calculated offset: 242, actual offset: 245 thus difference = 3
for node node id: 3: calculated offset: 15, actual offset: 19 thus difference = 4
for node node id: 2: calculated offset: 26, actual offset: 32 thus difference = 6
for node node id: 1: calculated offset: 132, actual offset: 136 thus difference = 4
for node node id: 7: calculated offset: 286, actual offset: 288 thus difference = 2
node id: 7 has calculated:
for node node id: 6: calculated offset: -46, actual offset: -43 thus difference = 3
for node node id: 5: calculated offset: -290, actual offset: -288 thus difference = 2
for node node id: 4: calculated offset: -106, actual offset: -102 thus difference = 4
for node node id: 3: calculated offset: -275, actual offset: -269 thus difference = 6
for node node id: 2: calculated offset: -264, actual offset: -256 thus difference = 8
for node node id: 1: calculated offset: -158, actual offset: -152 thus difference = 6
node id: 1 has calculated:
for node node id: 2: calculated offset: -107, actual offset: -104 thus difference = 3
for node node id: 3: calculated offset: -113, actual offset: -117 thus difference = -4
for node node id: 4: calculated offset: 55, actual offset: 50 thus difference = -5
for node node id: 5: calculated offset: -131, actual offset: -136 thus difference = -5
for node node id: 6: calculated offset: 111, actual offset: 109 thus difference = -2
for node node id: 7: calculated offset: 155, actual offset: 152 thus difference = -3
node id: 4 has calculated:
for node node id: 5: calculated offset: -186, actual offset: -186 thus difference = 0
for node node id: 3: calculated offset: -169, actual offset: -167 thus difference = 2
for node node id: 6: calculated offset: 56, actual offset: 59 thus difference = 3
for node node id: 2: calculated offset: -158, actual offset: -154 thus difference = 4
for node node id: 1: calculated offset: -52, actual offset: -50 thus difference = 2
for node node id: 7: calculated offset: 100, actual offset: 102 thus difference = 2
node id: 2 has calculated:
for node node id: 3: calculated offset: -24, actual offset: -13 thus difference = 11
for node node id: 1: calculated offset: 106, actual offset: 104 thus difference = -2
for node node id: 4: calculated offset: 158, actual offset: 154 thus difference = -4
for node node id: 5: calculated offset: -28, actual offset: -32 thus difference = -4
for node node id: 6: calculated offset: 214, actual offset: 213 thus difference = -1
for node node id: 7: calculated offset: 258, actual offset: 256 thus difference = -2
node id: 3 has calculated:
for node node id: 4: calculated offset: 168, actual offset: 167 thus difference = -1
for node node id: 2: calculated offset: 11, actual offset: 13 thus difference = 2
for node node id: 1: calculated offset: 117, actual offset: 117 thus difference = 0
for node node id: 5: calculated offset: -18, actual offset: -19 thus difference = -1
for node node id: 6: calculated offset: 224, actual offset: 226 thus difference = 2
for node node id: 7: calculated offset: 268, actual offset: 269 thus difference = 1
```

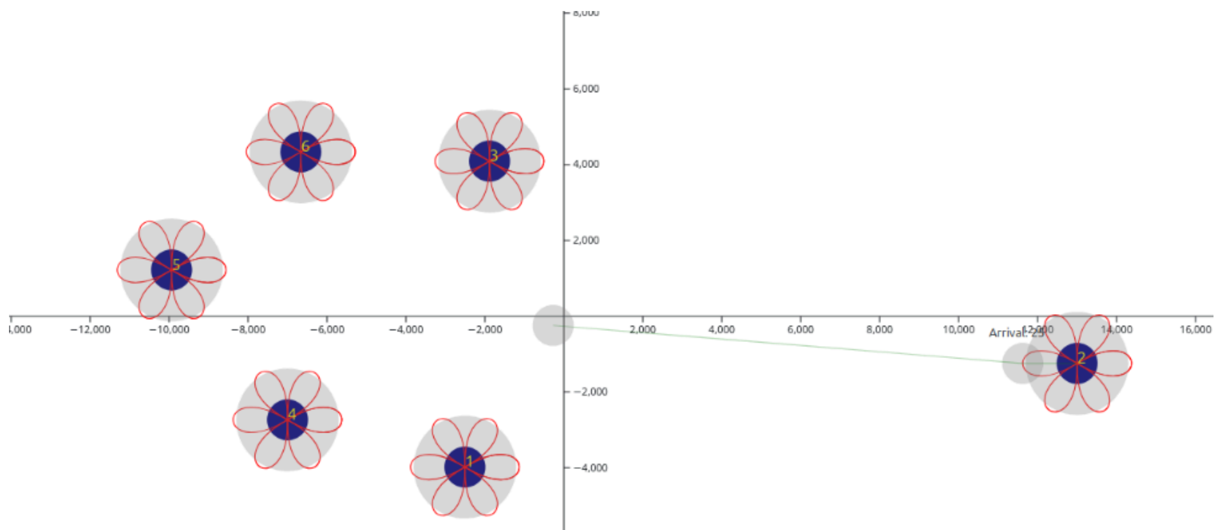
5. Node entering the network



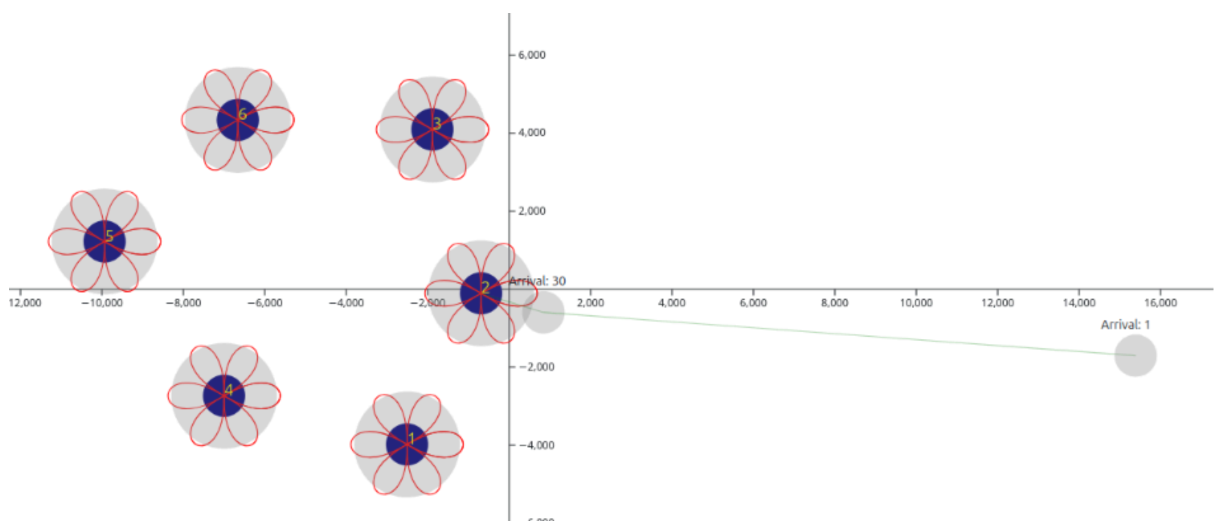
6. Node leaving the network



7. New shortest route appears



8. Shortest route disappears



B. Influence of sigma graph