AALBORG UNIVERSITY
STUDENT REPORT

Department of Computer Science
Selma Lagerløfs Vej 300
DK-9220 Aalborg Ø

Title:
Designing a Simulation Tool for a Synchronization Algorithm in a Sensor Network

Theme:
Distributed Systems

Project Period:
February 2021 - June 2021

Project Group:
ds105f21

Participant(s):
Søren Bundgaard

Supervisor(s):
Ulrik Nyman

Copies: 1

Number of Pages: 30

Date of Completion:
June 17, 2021

Abstract:

This project is about building a simulation tool that builds upon [1]. One of the main improvements is the routing algorithm the simulation tool uses. It is a managed flooding algorithm similar to the one found in Bluetooth mesh. The tool able more accurately simulate how data flows through the sensor network. The simulation tool also takes device location into account as the range of each device is limited. It is implemented in Scala and the performance is good compared to a similar model in Uppaal SMC. It allows the user to specify parameters from a real sensor network and then simulate the synchronization algorithm. The metrics that are produced can then be used to compare the synchronization algorithm against a more naive algorithm. The goal is that the user would know how many gateways per sensor are needed and how the gateways should be configured after running the simulations.

Contents

1 Introduction

This project works upon the findings from [1] which was done the previous semester. [1] was about designing and modeling a synchronization algorithm for a sensor network for the company Hexastate [3]. Hexastate is a company that sells software that can do predictive maintenance, and a problem with their sensor setups was that the connectivity from a factory to their cloud infrastructure could be limited. A solution to this was to move the predictions to the gateways placed within the factory. This limits the amount of data sent to the cloud, which has benefits in terms of bandwidth savings and security as less data leaves the factory floor. A problem with doing this is that the prediction algorithm bases its predictions on previous waveforms, which means that when a gateway receives a waveform from a sensor, all subsequent waveforms should be sent to the same gateway to achieve optimal predictions. An example setup was modeled in Uppaal SMC and the algorithm was tested. The algorithm seemed to work based on the results, but the processing times for each waveform were too high. The devices communicate through Bluetooth mesh, which has some specific properties which were not modeled. Uppaal SMC was also limited in how arrays and time worked which led to a more simplified model. This project will explore how a sensor network can be modeled and simulated more accurately. While including more realistic parameters such as device location and a better networking model. This should allow the user of the system to find good setups for their sensor networks.

1.1 Communication

In [1] the communication was modelled in a point-to-point manner. This meant that every device in the system was connected to each other through a central medium. This has the potential to work, but more tuning is required with relation to how additional devices affect the transfer speed. In the real world, a Bluetooth mesh network is used and a property of Bluetooth mesh is that when more devices are added there are more routes between the sender and receiver. Meaning that adding more devices to relay the data can have a positive effect on the transfer speeds as different parts of the same message are sent to a single recipient [10]. However, the locations of the added devices do have an impact on the change in transfer speed e.g. sensor 10 from Figure 1 has a lower impact on the transfer speed between gateway A and B compared to sensor 4, as data is more likely to flood through sensor 4 compared to sensor 10. A problem in [1] was that a lower amount of devices resulted in a much lower processing time, as a lower amount of devices did not affect the bandwidth of the system.

1.2 Bluetooth Mesh Networking

As mentioned in section 1.1 the sensor network uses Bluetooth mesh for communication. Bluetooth mesh runs on top of Bluetooth low energy which has some implications on the range and transfer speed. The range for Bluetooth low energy is less than 100m and the application throughput ranges from up to 305 kb/s - up to 1360 kb/s depending on which version of Bluetooth low energy is used [11].

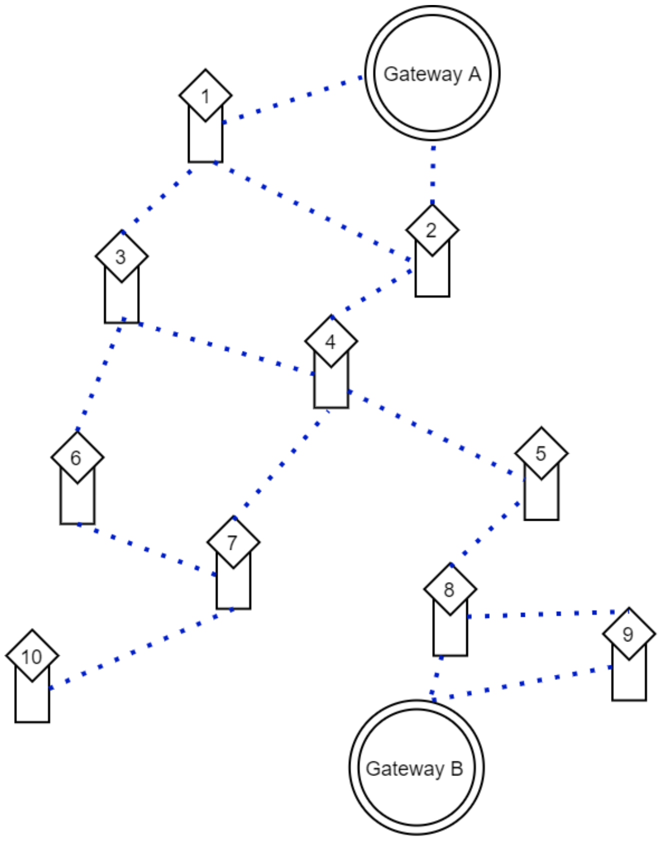

Figure 1: An example setup of 10 sensors and 2 gateways, which forms a Bluetooth mesh network.

As seen in Figure 1, not every device is connected to every other device, this would be infeasible due to Bluetooth low energy's range constraint. When a sensor wants to send data to a gateway the data has to go through intermediary sensors, e.g. if sensor 10 were to send data to gateway A the intermediary sensors could be sensors 7, 4, and 2. The routing algorithm used in Bluetooth mesh is a flooding algorithm where data is sent to every connected node [9]. By using flooding messages can take multiple paths and thereby achieve a higher transfer speed. Caching and a limit on hops are used to limit the reach of messages sent within the network. The message cache stores the identifiers of previously received messages, such that if a message is received and it is present in the cache it is not relayed any further. The cache logic is an implementation detail that is up to the specific manufacturer to implement.

A time to live (ttl) counter is present on every message, it is decremented each time the message is relayed and when the counter reaches zero it cannot be relayed any further. This puts a limit on how far in the network a message can travel, the theoretical limit is 127 hops. However, in a small network such as the one depicted in Figure 1 a ttl value of 127 would not be useful, because when the value is that large any message can reach every device, which is wasteful. An ideal ttl value for a message from sensor 10 to gateway A would be either 4 or 5 such that it is able to reach its destination but not much further. If the message is a broadcast message the ttl value is less important as the message should reach every device. Setting a low ttl value is more important when there are few receivers. Heartbeat messages are used to estimate a good ttl value, each device will emit heartbeat messages periodically. Heartbeat messages help with determining which devices are online and estimating the topology of the network. Heartbeats are broadcast to every device and when they are received the amount of hops is recorded. Each device keeps track of the received heartbeat messages, the sender, and a minimum and maximum amount of hops are saved and those values are used to estimate ttl value for the given component. The actual estimation based on the minimum and maximum hops is an implementation detail.

1.3 Comparing Simulation Tools

The sensor network has to be simulated with more realistic parameters, where location and network topology is taken into account.

Parts of the simulation in [1] were simplified due to time constraints and the tools used for simulation. Uppaal SMC lacked some features and required additional tooling for setting up the simulation, running in parallel, and processing the results. This made the process of running the simulation more complicated.

Requirements For Tools

To alleviate some of the issues with the model in [1] it makes sense to explore different tools. The requirements for the simulation tool are as follows:

- Expressive language for modeling
- Free for commercial use
- Low memory footprint
- Parallel simulation
- Integration with development environment
- Progress tracking

The simulation tool needs to be expressive to model the environment. This was an issue with Uppaal SMC as it was limited with how arrays were used, allocated, and initialized. Time was also an issue as it could only be used for logic but timestamps themselves could not be recorded within the model itself, which limited the model. Bluetooth mesh networking should be able to be implemented in the given tool as it is important for this specific simulation.

The tool being free for commercial use would be a benefit as it would make it cheaper for Hexastate as Uppaal SMC is only free for academic uses.

Parallel simulation is important for speeding up the simulation time. Uppaal SMC did not support this out of the box and in [1] a runner system had to be used to run simulations. This made the user experience more tedious as the runner system also had to be configured to work with Uppaal SMC and some functionalities from Uppaal SMC were no longer usable because of the runner system. The feature that estimated the number of runs needed was not usable as it would give the value after it was done with all the simulations.

Memory footprint is another requirement, it is not the most important requirement. However, having a low footprint makes it easier to test on a local system. Uppaal SMC for Windows was only available in 32-bit which limited the memory usage to about 4GB which was not enough for some models. However, Uppaal SMC is available as a 64-bit application on Linux and macOS.

Progress tracking can be very helpful for catching errors early, as a slowdown in the simulation can indicate that something is wrong. In [1] some errors could have been caught early if there was some kind of progress tracking, by being able to analyze the collected data before the simulation is finished. The progress being slow could also indicate something being wrong within the model.

Integration with a development environment makes the user experience nicer when developing, running, and processing data from the model, since everything would work together. Uppaal SMC does have some Java libraries and CLI for interfacing with other tools. The CLI for Uppaal SMC helped with creating the runner tool but the Java libraries were not used in [1].

Simulation Tool Candidates

There exist many simulation tools the main characteristics I looked for were how they are used, which libraries they offered to make modeling, especially the Bluetooth part, easier, and their license. The candidates I have chosen are Uppaal SMC, ns-3, Matlab, and writing my own custom tool.

Custom Tool

A custom tool gives a lot of flexibility in terms of development, features, and license. Since it is made from the ground up features from section 1.3 can be implemented. However, more work is required as it has to be designed from scratch. The way the tool would work would be to define some components which would react to time passing, much like Uppaal SMC but simpler. The language the tool would be written in would be Scala as that is the main language Hexastate uses, which also makes it easier for them to maintain in the future. Another benefit of writing the tool in Scala is that some of the logic can be run on the real hardware, such as the logic on the gateways. And it also benefits from having access to the Scala ecosystem and the flexibility that comes with the language. However, the Bluetooth mesh networking logic has to be implemented.

Uppaal SMC

Uppaal SMC [2] could be used again for this project, but it is not flexible in the modeling language, the main problem is how arrays are defined and allocated. However, this could be generated by an external tool by editing the XML file Uppaal SMC uses for representing a model. The biggest issue with Uppaal SMC is the memory footprint and how it is integrated into the development environment. To implement asynchronous communication between components an array has to be used. This array had to be sized after the worst-case scenario [1]. Debugging and testing Uppaal SMC models can be a bit difficult as there is no debugger and when the model is large the program becomes unstable. However, there are libraries for interfacing with Uppaal SMC, but like the custom tool, the adapter has to be implemented. It is also not possible to get progress data or estimates on when the simulation is done, but this can be estimated with a runner tool. It is proprietary software meaning a license is required to use it commercially and the software itself cannot be modified.

NS-3

NS-3 is a network simulation platform, it is very low level in terms of abstractions and has a lot of features when it comes to internet communication. It is designed as a collection of libraries that are used in C++ applications, it is also free and open-source [7]. Since it does not use a proprietary language it is possible to integrate it into a development environment and progress tracking can also be implemented. However, it does not support Bluetooth mesh networking out of the box, but there have been developed modules for Bluetooth mesh networking by third parties [8]. Ns-3 is very focused on the network aspect, such as how the specific protocols work, how the routing is done, and the development of these. This might be out of scope for this project since an approximation of the Bluetooth mesh network might be good enough.

Matlab

Matlab is another tool that can be used for simulating this system. It offers some nice functionality for displaying data and running simulations in a multi-core environment [5]. Unlike ns-3 it has some modules for Bluetooth mesh, but it does not have much documentation for the specific modules [4]. Like ns-3 the Bluetooth modules might be too focused on the protocols and how they can be developed and optimized. The biggest downside to Matlab is its license, it is free for academic use only.

Simulation Tool

It makes sense for this project to build a custom simulation tool. The reasoning behind the decision is that it gives a lot of flexibility in terms of what and how different parts of the network are implemented. The Bluetooth mesh networking part will also only implement the logic as described in section 1.2 since those parts are important for this project. Matlab and ns-3 would require the developer to use the entire Bluetooth stack and thereby make the simulation tool needlessly complicated. The language used for implementing the simulation tool will be Scala as it is the main language used at the Hexastate. Implementing the tool in a language they already know will allow them to make modifications easily. Scala offers a nice selection of features and libraries for modeling the problem in a typesafe manner which will make the tool easier to debug.

2 Design

2.1 Simulation Tool Design

The simulation tool will work with a discrete-time system to simplify the model. Meaning that each step the simulation takes is a specific point in time, the time between two steps, therefore, cannot be represented. Time can then be represented as an integer that can correspond to either a second, minute, or an hour, depending on the granularity that is required. A more precise time representation will require the simulation to go through more steps. The system itself consists of a top-level system, which can be seen as a container that contains a global state, and a list of subsystems.
The subsystems have their internal state and have access to the global state but they do not have subsystems of their own. For each simulation step, each subsystem takes turns in changing the global state and its state. However, the order that the subsystems appear in has to be shuffled for every step to introduce some form of randomness as all traces otherwise would result in the same end state, it also simulates concurrency.
A simple example of a system would then be a top-level system containing 1 gateway and 2 sensors. The sensors would send data to the gateway and it would in turn record when data has been received in the global state. Each of the components would communicate through the global state.

A state St consists of two parts. (V, Co)

- V - a set of Scala variables, representing the global state
- Co - Sequence of Components, $C \in Co$

A component C consists of four parts. $(V_{in}, V_{out}, V_c, F)$

- V_{in} - $\subseteq V$ Input variables
- V_{out} - $\in V$ Output variables
- V_c - a set of Scala variable, the internal state of a component
- F - a Scala function defined over: $V \times C \to V \times C$

A simulation trace is generated by taking the *tick* transition n number of times. A trace is a sequence of ticks $St_0 \xrightarrow{tick} St_1 \ldots \xrightarrow{tick} St_n$ the length of the trace corresponds to the amount of tick transitions. Both components of the state change.

$$(V, Co) \xrightarrow{tick} (V', Co')$$

The values of the variables in V change and the order of components in Co are shuffled. To describe the tick transition and how the values change additional functions have to be defined.

The function F which is defined on every component is the logic of the specific component and will behave differently based on the input, the component state, and the global state. $f \in c, c \in Co, f(v, c) = (v', c')$ where both the global and local state is changed. Each components' F within a state St has to be called for each *tick*. Each call will modify the global state $v \in V$, which is supplied for the next component's call of the function F. However, the new internal state of the components have to be collected, this is done with the function $tick_{help} : V \times Co \times C \to V \times Co$. The result of the function will result in a changed state and the state change of the component is prepended to the input list. $tick_{help}(v, co, c) = (v', co')$ where $c.f(v, c) = (v', c')$ and $co' = c' :: co$ This function in combination with a combination function will result in a new state and is the logic of the tick transition. The idea is the co will start out empty and at the end of the *tick* transition it will contain every updated component. The *tick* transition can be described as a call to a fold[1] method. $fold(tick_{help}, (v, \langle\rangle), co)$ The result of the *fold* call will result in a new state as the return type can be combined into a new instance of St. However, using the above definition to produce a trace will result in every trace being identical. Randomness have to be introduced as not every sub-component will change the global state in the same order for every tick. This is done with a shuffle function.

[1]The *fold* method, also sometimes called reduce, is a part of the Scala standard library and combines every element into one, given a function. Url: `https://www.scala-lang.org/api/current/scala/collection/IterableOnceOps.html`

$\mathit{shuffle} : Co \to Co$ will change the order of the components in Co. The semantics would be:
$\mathit{shuffle}(co) = co'$ where $co = \langle c_1, c_2 \ldots c_n \rangle$ and $co' = \langle c'_1, c'_2 \ldots c'_m \rangle$ such that $\forall c_x \in co \implies c_x \in co' \wedge \forall c_y \in co' \implies c_y \in co \wedge n = m$

With these definitions we can define four functions:
$step : St \to St$
$steps : St \to \mathcal{P}(St)$
$trace : St \times \mathbb{Z}^+ \to Sts$ where Sts is a sequence of St
$traces : St \times \mathbb{Z}^+ \to \mathcal{P}(Sts)$
The *step* function is a call to $\mathit{fold}(tick_{help}, (v, \langle\rangle), \mathit{shuffle}(co))$ where the result is the state of the system in the next time step.
The *steps* function like *step* also takes a state St as an input. However the return is all possible states for the next time step.
The *trace* function given a positive integer n and a starting state st will call $step$ n times:
$trace(st, n) = \langle st_1, st_2 \ldots st_n \rangle$ where $st_1 = st, st_2 = step(st_1) \ldots st_n = step(st_{n-1})$
the *traces* function given a starting state st and a positive integer n will return all possible traces of length n.

The result of the function *traces* will describe all of the possible behaviors of the model. However, the result of *traces* is infeasible to calculate for most models. The Monte Carlo method [6] is used to approximate the result of the functions *traces*. This is done by generating random traces with the *trace* function. The result should then come close to the result *traces* after a sufficient amount of random traces have been generated.

2.2 Component Design

Components that make up the sensor network have to be designed. The components include gateways, sensors, and mediums. These all have to fit into the description from section 2.1. Like in [1] there were additional helper components that would turn off gateways to simulate crashes, these would have to be designed as well.

Communication between components

To accurately capture the behavior of the managed flooding routing algorithm found in Bluetooth mesh, messages have to include *Message ID* and a *Time to Live* counter. To make the implementation easier two types of messages are defined and can be seen in Figure 2. The size of the message is given by the data field. The messages are placed within message queues and the queues for every device can be seen in Figure 3.

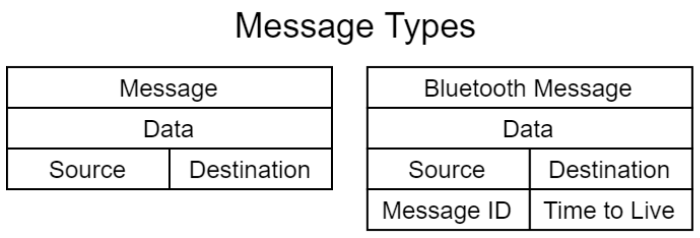

Figure 2: Fields included in messages.

Messages are stored within the *Outgoing* and *Incoming* queues and the *Bluetooth Message Queue* contains Bluetooth messages. To simulate wireless data transfers messages are transferred from the queues from one device to another.

Messages from the *Outgoing* queue are messages generated from the device. Messages in the *Bluetooth Message Queue* are messages received from other devices, which can be relayed and consumed. The reason for multiple queues is to create a separation between messages. The separation makes it easier for the consumer of the interface as it does not have to calculate the time to live field. Figure 3 also includes the high-level communication API which inserts and retrieves messages from the message queues.

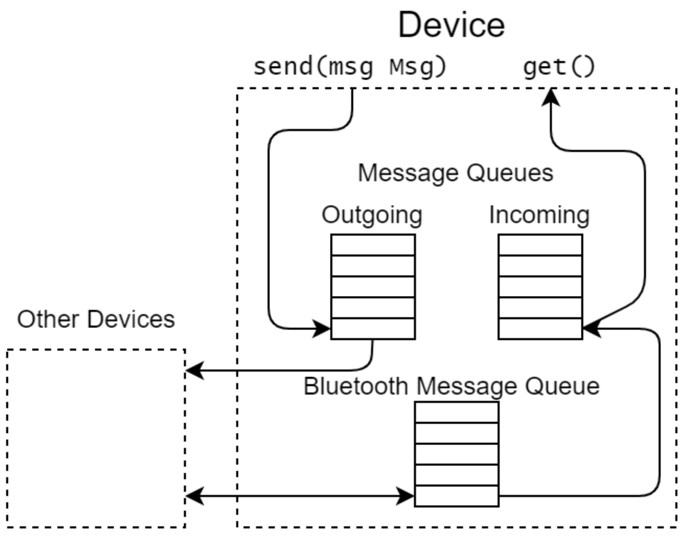

Figure 3: Message queues used for communication.

The algorithm for processing messages can be seen on listing 1, which handled the sending logic for messages already in the queues. The *consumeMessage* function will put the message in the *Incoming* queue. The *relay* function handles the actual sending, it will lookup the nearby devices and send the message to them, the bandwidth used is dependent on the distance between the devices and the size of the message. The bandwidth increases based on the inverse square law, which means that if the distance doubled the amount of data needed to be transferred also doubles. This makes the location of the devices in relation to each other important, as the devices can only transfer a set amount of bandwidth units per tick.

```
processMessage() {
  val msg = getMsg()
  cache.add(msg)
  if (msg is broadcast or for this device) {
    consumeMessage(msg)
  }
  if (msg is not for this device or is broadcast) {
    relay(msg)
  }
}
```

Listing 1: pseudo code for processing messages

Medium

The medium uses the queues from Figure 3 to facilitate communication. Mediums, sensors, and gateways are all run concurrently, but every medium will belong to either a sensor or a gateway. The medium is the device that will handle the Bluetooth logic.

To simulate the fact that data transfers are not instantaneous each medium has a bandwidth limit, this limit is reset every tick.

The medium also has a cache containing the message ids of received messages, when a message is sent to a medium the message id is checked against the receiving mediums cache. The message is added to the cache when it is processed as stated in Listing 1.

The state diagram of a medium can be seen in Figure 4, it will either process messages or broadcast heartbeats. Broadcasting heartbeats is an action that occurs at a regular interval e.g. every 15 minutes. The time to live value is estimated as an average between the minimum and maximum amount of hops. When a message is taken from the *Outgoing* queue the time to live value will be calculated.

Heartbeat messages can be seen as regular messages that have to be relayed to every device within this network.

To keep message processing fair the medium will alternate between getting messages from the *Outgoing Queue* and the *Bluetooth Message Queue* in the case that they both have messages waiting to be processed. Devices that are offline will not process or be sent any messages.

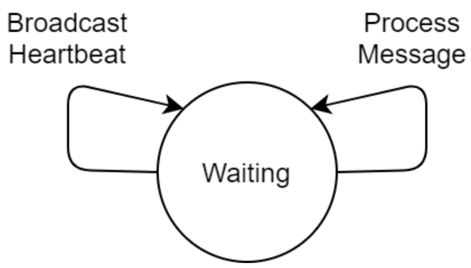

Figure 4: State diagram of a medium.

Sensor

A sensor has a schedule e.g. four times each day where it samples waveforms at 8:00, 10:00, 14:00, and 18:00, with a window of 30 mins, meaning the first measurement of the day would be sometime between 8:00 and 8:30.

The sensor has two states, a *Waiting* state and a *Ready to sample* state, which is shown in Figure 5. Using the previous example the state would be *Ready to sample* between 8:00 and 8:30 and between 8:30 and 10:00 it will be in the *Waiting* state. This schedule is in the real world sent from the gateways when the system is set up, but to simplify the system this project assumes that sensors have already gotten the schedule information.

The Sensors assume no knowledge of the topology of the network, meaning that a waveform is broadcast to all reachable gateways. This means that a waveform eventually will end up at every gateway, the closest online gateways would in most cases receive the waveforms first. The waveforms are split up into multiple pieces and may not all end up at the same gateway, in that case, the sensor should create a new waveform if it is requested within the time window. It should be possible for one gateway to receive a complete waveform within 30 min as they only take about 20 seconds to send.

The sensor uses the API from Figure 3 and then the medium handles the actual data transfer.

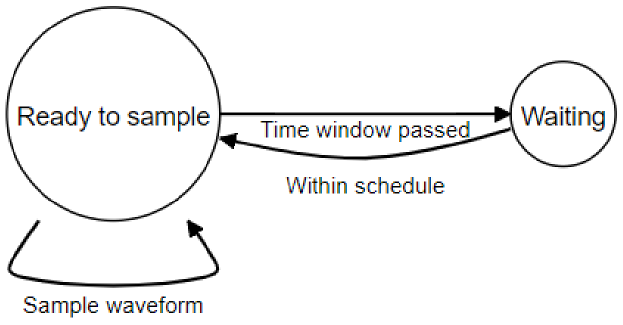

Figure 5: State diagram of a sensor.

Waveforms

Waveforms are the data that is sampled by the sensors. They are sampled in a given time window e.g. 30 mins, this means that two waveforms sampled within the same window are equivalent in terms of building a statefile. Waveforms are sent in parts due to their size, splitting, and reassembly is not handled by the Bluetooth layer and there is no error correction, meaning all parts have to be present to assemble the waveform. The waveform information can be seen on listing 2.

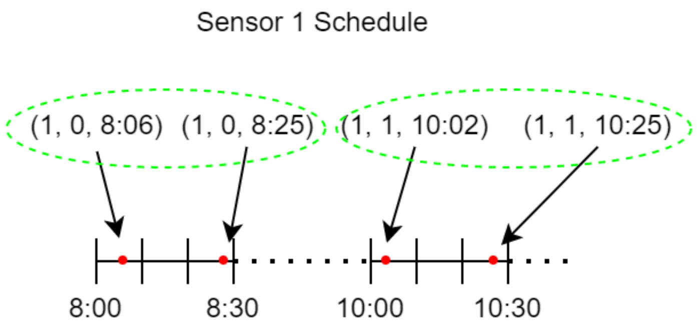

Figure 6: Schedule for sensor 1, red dots denoted times where waveforms have been sampled, the triples is a waveform representation (id, number, sample time).

A waveform is distinguished by which sensor sampled it and the number, two waveforms sampled within the same time window will have the same id, this is shown in Figure 6. However, parts from different waveforms cannot be mixed and the newer waveform is preferred, they are only ignored if an older waveform with the same id already has been sent to the cloud. The waveform number is incremented when a time window has passed.

```
class Waveform(
  sensorId: Int,
  number: Int,
  shouldBeSampledBefore: Int,
  sampledAt: Int
)

class WaveformId(
  sensorId: Int,
  number: Int
)
```

Listing 2: Code describing waveform types

Gateway

The Gateway is the most complex component of this system as it needs to run the waveform data synchronization algorithm.

Gateways sometimes crash or lose connection this happens throughout the day. The gateway crashing mechanism is the same as the one in [1].

It makes sense to split the gateway logic into two parts, one that handles the assembly and requests for waveform parts, and the other that handles the synchronization. This adds another message layer that only exists within the gateway. These messages are waveform messages that consist of a fully assembled waveform and waveform updates.

```
class WaveformPart(
  waveform: Waveform,
  partNumber: Int,
  maxNumber: Int
)

class WaveformUpdate(
  sensorId: Int,
  length: Int,
  nextNeeded: Int,
  trackedBy: Int,
  timestamp: Int,
)

class WaveformPartsRequest(
  waveformId: SimpleWaveformId,
  requester: Int
)

// receiver is given by the message wrapper
class WaveformSampleRequest
```

Listing 3: Code describing gateway messages

Waveform Handler

The waveform handler is the part of the gateway that assembles waveforms, if that is not possible it will either request waveform parts from other gateways or send a sample request to the sensor to which parts are missing. The data contained within the waveform handler can be seen in Figure 7. It is similar to the Bluetooth message queues from figure 3, but the waveform handler treats some messages differently. All messages going to the gateway have to go through the waveform handler. This is to achieve separation of concerns since waveform data arrives in parts and they have to be assembled before they can be used. The main tasks are to assemble and disassemble waveforms such that they can be processed and sent to other gateways.

Gateways can process four types of messages, WaveformPart, WaveformUpdate, WaveformPartsRequest, and WaveformSampleRequest, they can be seen in listing 3.

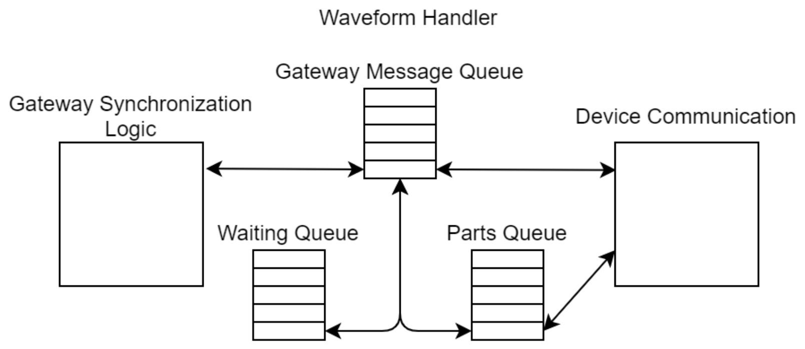

Figure 7: Data contained within the waveform handler.

Waveform parts belonging to the same waveform are placed within a container, which keeps track of which parts are needed. The container is then placed within the parts queue. It is not a message queue, but rather a priority queue where the placement denotes the urgency of getting the parts for the given waveform. The parts queue is checked every tick, both for completed waveforms and waveforms that will soon reach the deadline. If pieces are missing a parts request can be broadcasted to the other gateways such that the missing parts can be retrieved, this is done 10 minutes before the deadline. If that fails a sample request can be sent to the sensor that sampled the waveform such that a new one can be sampled. Sample requests are sent 5 minutes before the deadline and the new sample will invalidate old parts. Each of these requests is only sent once per waveform and towards the end of the deadline as to not spam the network. The times when the requests are sent are configurable. When a waveform is completed it is forwarded to the gateway via the *Gateway Message Queue*, and the waveform handler will not respond to any parts request for that waveform. If a gateway has to send a waveform to another gateway, it has to be disassembled, which is also done within the waveform handler. If the receiving gateway is offline when the waveform is sent the waveform is put into the waiting queue, it will try to send it about 10 times and will wait about 3 minutes between each attempt, these times are also configurable. The waiting queue is also a priority queue where the top element is the waveform that should be resent next.

Synchronization Algorithm

The synchronization algorithm is almost the same as it was in [1]. However, due to the mesh network, it has to be modified to handle that there are multiple copies of the same waveforms within the network. The main change needed is to add a timestamp to the waveform updates. The waveform updates will then include information about the waveform length, the next needed waveform, which gateway is tracking it, and a timestamp for its creation. A statefile is built based on the received waveforms and whenever a waveform is committed to the statefile a waveform update is made. This update is then sent to all gateways. Alongside the statefile is information about which sensors are tracked by which gateways, this is updated through waveform updates. When a waveform update is received the next needed value is compared and the value of the incoming waveform update has to be higher than the one that is present. If they are equal it is the oldest update that takes precedence. The idea is that the gateway that creates the oldest update for a given sensor is the gateway that is closest and should therefore be the gateway that tracks that sensor. When a waveform has been committed the waveform conclusion is sent to the cloud. The conclusion has information about the waveform, the gateway, and when the conclusion was created. This makes it easier to calculate the processing time, which is:

$$\textit{conclusion timestamp} - \textit{sampling timestamp}$$

To test the synchronization algorithm a baseline algorithm also has to be developed. Like in [1] the baseline system is simple, when a waveform is received it is sent to the cloud regardless of which other gateways were tracking it.

3 Implementation

The simulation system was implemented according to the design described in section 2.1. The Global component represents the state, this component will then contain the gateways and sensors. Subsystems will change the global state through interfaces such that only relevant variables are exposed to the subsystems.

3.1 Component Abstractions

Having abstractions makes it easy to change parts of the model, while also making it easier to test. Testing each component individually makes it possible to verify the specific component logic works as intended. Without abstractions, a complete model has to be set up for each test, which will make the tests more complicated.
Having the interfaces allows for simpler implementations in unit tests, where only the logic of the component being tested is important. An example of such an interface can be seen in Listing 4. In the production model, there is one big communication class, which includes queues for every component, but in the unit test, there is an incoming queue and a queue denoting the sent messages.
The idea is then to setup messages, which the component is about to receive, then to wait some amount of time and then make assertions to the component state and the sent messages.

```
trait HighLevelCommunication {
  def send(msg: HighLevelMessage): Unit
  def sendSelf(msg: HighLevelMessage): Unit
  def get(id: Int): Option[HighLevelMessage]
}
```

Listing 4: HighLevelCommunication interface, used by sensors and gateways.

There are abstractions for logic related to devices that are in range, down detection, time handling, and communication with the cloud.

Tests and Optimization

During development tests were used to validate that the changes made to the code base did not break any component logic. There ware two types of tests, unit test which tested each component individually and integration tests, which was a test setup with a specific seed and settings.

The result of the integration test resulted in a value for the processing time of the waveforms. Since the seed between runs is the same, the order in which the components were interacting with each other are also identical thus resulting in the same result. Memory usage and run time were also recorded such that the performance also can be compared.

The development cycle after having the model and tests developed, included making changes to the code base and use the unit test to identify areas in the code base were errors could arise. After the unit tests all passed the integration tests could be run and the results would be compared.

The integration tests were mostly used for performance related changes, as that should not have an impact of how the components act with relation to each other, thus resulting in the same results. However, the performance changes should result in lower ram usage and run time.

3.2 Profiling

Having the tests was not enough as it only gives a total value of how long it took to run or how much memory was used in total. A profiler can give a more granular picture of which methods used resources. The specific profiler used was VisualVM[2], which hooks onto a running JVM program and shows metrics. This is useful as it will show potentially inefficient functions.

Cache Optimization

An inefficient part of the program was message caching, it used a lot of memory as it kept growing.

[2]https://visualvm.github.io/index.html

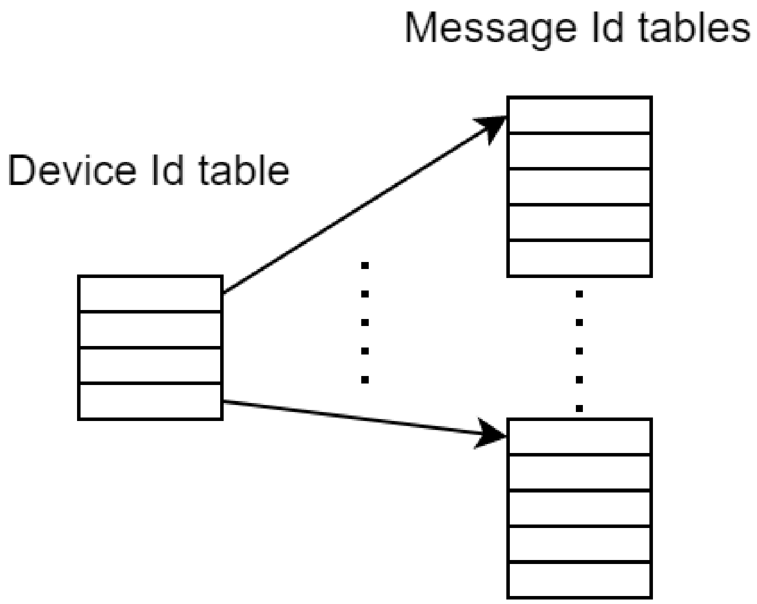

Figure 8: Memory layout of the message caching.

Figure 8 shows the memory layout of the cache, a global object has a table of each device, which points to a set of messages ids received by a specific device. The easiest way to solve the memory issue is to make the cache smaller. However, there needs to be a system that can figure out which messages are not looked up. It should be possible to make the cache smaller as messages only are within the network temporarily. When the message is no longer in the network it can be removed from the cache without it impacting the model behavior.

To get the set of messages that are still actively transferred, the Bluetooth message queues for every device can be used. The set of all message ids in all Bluetooth message queues is the set of active messages. However, iterating through every message of every message queue every tick is inefficient and scales very poorly when adding more devices.

Reference counting can be used to keep track of active messages in a time and space-efficient way. Whenever a message is sent from any device the message is added to a lookup table and the count for that message id is incremented by 1. Whenever a message is taken out of a queue the value for the message id is decremented by 1. Then every message with a non-zero value is an active message.

This active message lookup table can be compared against the message cache and every element in the message cache that has a value of zero can be removed. However, to remove one message from the message cache with the memory representation of Figure 8 the time complexity is $O(Cache\_Size * Devices)$. This can be improved by inverting the cache which can be seen in Figure 9.

The inverted cache works by looking up a message id which then gives a set of devices that contain that message. The space complexity is identical, but the time complexity to remove a message is only $O(Cache\_Size)$ which is an improvement.

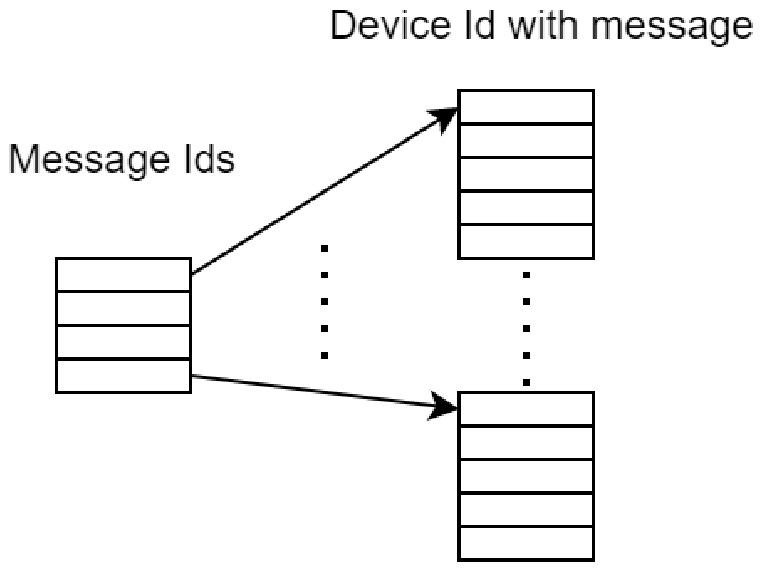

Figure 9: Memory layout of the inverted message caching.

Intuitively it makes sense to remove messages from the message cache as they are removed from the set of active messages. However, by running benchmarks, it was found that removing messages in batches was faster.

4 Simulation Experiments

The simulation system produces metrics for processing time for waveforms, score, and statefile rebuilds, all of these are similar to the ones from [1]. The score metric is the least intuitive of the three, it is calculated based on the following formula:

$$Score = \frac{\sum_{i=1}^{w} len(i)}{\sum_{i=1}^{w} i}$$

The score value is per sensor, and the function $len(i)$ gives the current length of the statefile when waveform i was committed. The value of i corresponds to the optimal length of the waveform. The idea is that a score of a given statefile is penalized when a rebuild happens, as $len(i)$ becomes zero after a rebuild. The longer a statefile is the better predictions it can produce, this means that in the beginning, a statefile rebuild is not as severe as one that happens later. In the best case, we have that $\forall i, len(i) = i$ and then the score would be 1.

Each system can be configured such that it reflects a real system. This is done by specifying parameters such as where sensors and gateways are placed and how they are configured. To simplify the system it is assumed that some settings are the same for all sensors and gateways, i.e that all devices have the same bandwidth. However, each sensor can have a unique sending schedule.

4.1 Result Data

The output of a single experiment is a list of waveform conclusions, which can be seen in Listing 5. Conclusions for a single experiment can be aggregated into the metrics.

```
class Conclusion(
  sensor: Int,
  sampleTime: Int,
  receivedTime: Int,
  deadline: Int,
  number: Int,
  length: Int,
  gateway: Int
)
```

Listing 5: Datapoints for results

All metrics are calculated on a per sensor basis, this makes it possible to combine the data from experiments with different seeds.

4.2 Parameters

To run an experiment parameters have to be defined. There are parameters for running the simulation and for setting up the experiment. To run the simulation the following information is needed:

- How many ticks to run
- Seed for random number generator
- The amount of trials
- How many threads to use
- Experiment setup file

Before any experiment can run parameters have to be defined.

The experiment setup file includes information about locations of sensors and gateways, it also defines global settings such as how long a day is in ticks. The length of a day is used to generate the sending schedule for each sensor. The other global parameters can be grouped into three groups, gateway settings described in Table 1, message size in Table 2, and Bluetooth settings in Table 3. The tables include default values. The default for how many ticks in a day is 86400 which corresponds to one tick per second. By default all time-related settings assume that one tick is one second. however, if a different day length is needed such as 1440, which corresponds to one tick per minute, then all time-related settings have to be changed accordingly, to ensure similar behavior. To simplify the system every device in the system will have the same global settings.

Gateway Settings		
Setting Name	Type	Default Values
Time before new sample	Int	300
Time before requesting parts	Int	600
Resend attempts	Int	10
Wait time	Int	180
Time between crashes	Int	60
Crash time	Int	30
Baseline	Boolean	False

Table 1: Global settings for gateways. Time units are in ticks.

For the global gateway settings, the parts and sample requests can be tweaked to achieve better results. If the values are high the requests are sent earlier, leading to more messages in the system, this may be necessary if waveforms get lost due to gateways that crash. In [1] different values for resend attempts were tested, but in this system, the wait time value specifies the time between the attempts. In [1] it was found that the higher amount of attempts the better the score became, but at a cost of processing time. In this system both values have to be tuned as the total wait time will always be:

$$Total\ wait\ time = Resend\ attempts * wait\ time$$

Message Sizes		
Setting Name	Type	Default Values
Heartbeat message size	Int	1
Waveform part size	Int	10
Parts	Int	1000
Sample request size	Int	3
Parts request size	Int	3
Waveform update size	Int	5

Table 2: Global settings for messages. Time units are in ticks.

Message sizes can also be tuned. The most interesting messages are waveform parts as they are the most sent messages and therefore will have a big impact on the transmission speed. Different customers might also have different requirements for how much data is sampled and how many parts the waveform consists of.

Bluetooth Settings		
Setting Name	Type	Default Values
Max time to live	Int	127
Bandwidth	Int	20000
Max Multiplier	Int	10
Device range	Double	4.5
Time between heartbeats	Int	900

Table 3: Global settings for Bluetooth. Time units are in ticks.

Bluetooth settings specify how data is transferred. Bandwidth specifies how much data can be sent for each tick. The distance between two devices also impacts the data needed to be sent. Two devices have to be within the specified range of each other and the further away the devices are the more data needs to be sent. However, due to the way the inverse square law works the amount of data needed to be sent grows towards infinity if 2 devices have the maximum distance between them. To avoid having very large messages the max multiplier is can be specified, a max multiplier of 10 will increase message size up to about 90% of the distance. This will still penalize devices that are far away without using all the bandwidth. In a real world setup the devices would not be able to send anything to a device that is approaching the maximum distance. In the test setups all devices are within 90% of the maximum distance.

Only sensors have unique settings which are the sending schedules. The sending schedule is a list of integers which is an offset for when on a given day the waveform should be sampled. A schedule has a deadline which is the same for every waveform sample time. A new waveform cannot be sampled before the deadline for the previous waveform has passed.

4.3 Experiments results

Experiments have been conducted to test the synchronization algorithm. Six experiments have been conducted each of them uses the default parameters, but the number of gateways varies. Half of the experiments are baseline experiments. The locations of the devices have been randomly generated, but with the constraint that every device is reachable from every other device and that every device is within 90% of the total device range. There is a setup with 3, 5, and 7 gateways all with 50 sensors, and the sensors have the same locations across all experiments and all sensors would sample waveforms 4 times each day.

The physical locations of the devices can be seen in Figure 10, 11, and 12. These experiments should be able to show the impact of a different amount of gateways. The experiments were run for 30 simulated days and each of these were repeated 40 times.

There are three metrics for the results: Processing time in Table 4, statefile rebuilds in Table 5, and score in Table 6.

When looking at the processing times the general pattern is that the baseline runs tend to have a lower worst-case processing time, which is consistent with the results from [1]. However, adding or removing gateways does not seem to have a big effect on processing times as the results are similar and the worst-case processing time for the 7 gateways could be attributed to bad luck.

When looking at statefile rebuilds the baseline has more than the non-baseline experiments which is also consistent with [1]. However, In these experiments waveforms were broadcast to every gateway which could explain the low number of statefile rebuilds on the median case, whereas in [1] sensors would randomly send the waveform to another gateway. With more gateways in the system more devices have to share information, this could explain why the number of rebuilds is higher when more gateways are added. However, the rebuilds that do happen seem to be at the beginning as those have less impact on the score.

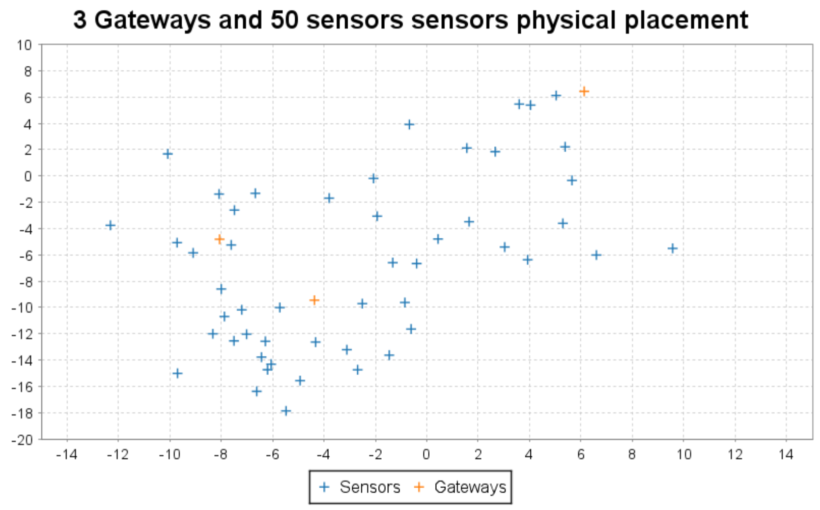

Figure 10:

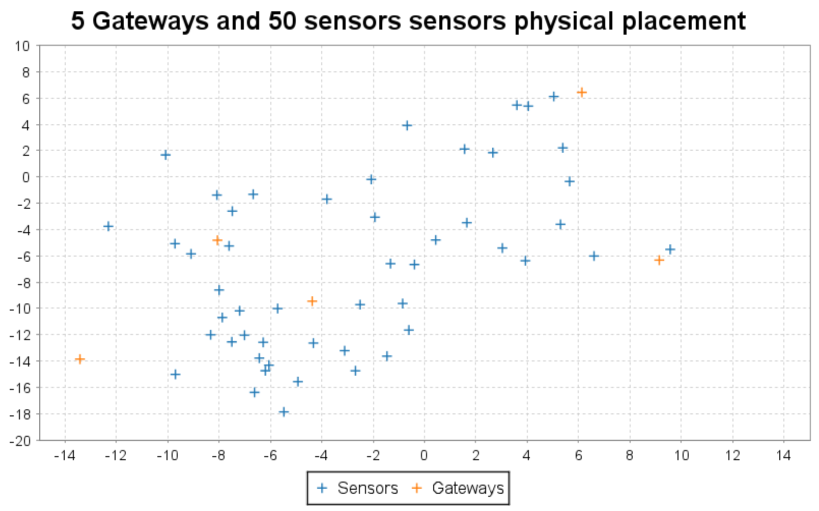

Figure 11:

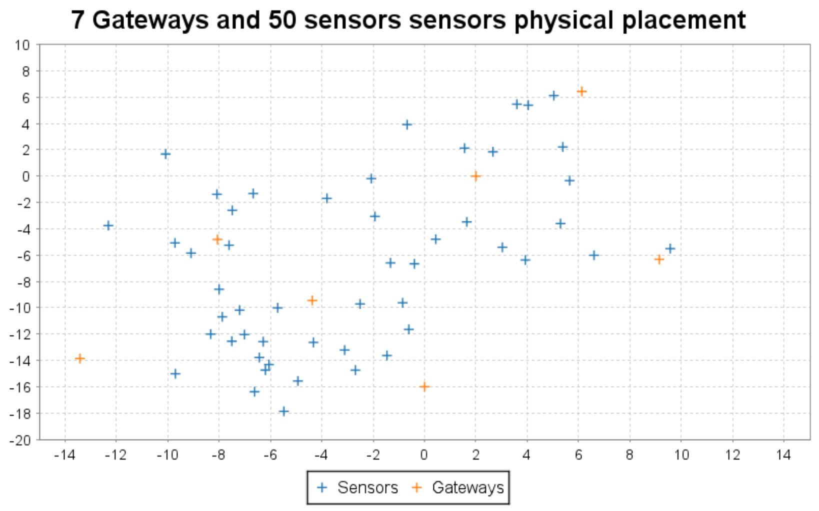

Figure 12:

The results of the experiments suggest that there is an advantage to use the synchronization algorithm as the scores for the experiments are higher. However, the difference is smaller, the max score for the experiment with 5 gateways and 50 sensors in [1], was around 9% whereas the max for this experiment was 100%, this could be because of the waveforms being broadcast to every gateway. In general, the scores are high for the baseline experiments compared to [1]. However, having more gateways per sensor suggests that the statefile is being rebuilt more often, this is not reflected in the score, which indicated that they happen in the beginning. A reason for this could be that more information has to be shared between the gateways and this scales when adding more gateways. The risk of any gateway being crashed increases as more are added which then could lead to information being lost and then statefile rebuilds.

These experiments show that the synchronization algorithm works compared to the baseline for the test setups. Having longer crash times or having gateways crash more often would probably make the experiments with different amounts of gateways have more different results. A user of this tool should be able to figure out a good setup e.g how many gateways per sensor and then get metrics for how this system performs.

Processing Time					
Test Setup	Min	Avg	Max	Median	5% worst
3 gateways 50 sensors	44.00 s	1595.70 s	13500.00 s	1148.00 s	>5537.00 s
3 gateways 50 sensors baseline	44.00 s	1574.42 s	9471.00 s	1094.00 s	>5779.00 s
5 gateways 50 sensors	10.00 s	1619.71 s	17783.00 s	1057.00 s	>5668.00 s
5 gateways 50 sensors baseline	11.00 s	1171.09 s	9411.00 s	842.00 s	>3569.00 s
7 gateways 50 sensors	11.00 s	1631.45 s	30747.00 s	1100.00 s	>4779.00 s
7 gateways 50 sensors baseline	11.00 s	1054.18 s	9168.00 s	737.00 s	>3237.00 s

Table 4: Processing time for the experiments. Values are per sensor.

Statefile Rebuilds					
Test Setup	Min	Avg	Max	Median	5% worst
3 gateways 50 sensors	0.00	0.02	1.00	0.00	>0.00
3 gateways 50 sensors baseline	0.00	4.10	119.00	0.00	>48.00
5 gateways 50 sensors	0.00	0.04	1.00	0.00	>0.00
5 gateways 50 sensors baseline	0.00	8.27	119.00	0.00	>60.00
7 gateways 50 sensors	0.00	0.05	1.00	0.00	>0.00
7 gateways 50 sensors baseline	0.00	13.37	119.00	0.00	>66.00

Table 5: Statefile rebuilds for the experiments. Values are per sensor.

Scores					
Test Setup	Min	Avg	Max	Median	5 % worst
3 gateways 50 sensors	98%	100%	100%	100%	<100%
3 gateways 50 sensors baseline	0%	94%	100%	100%	<3%
5 gateways 50 sensors	98%	100%	100%	100%	<100%
5 gateways 50 sensors baseline	0%	85%	100%	100%	<1%
7 gateways 50 sensors	98%	100%	100%	100%	<100%
7 gateways 50 sensors baseline	0%	74%	100%	100%	<1%

Table 6: Scores for the experiments. Values are per sensor.

5 Evaluation

The simulation tool seems to work and it produces good results based on the test parameters. Based on the requirements from section 1.3 the simulation tool seemed to satisfy most of them.
The **expressive language for modeling** is a more general requirement for how the tool is built, not necessarily how the logic of the specific components behaves. However, if more features were to be added having unit tests and interfaces to abstract away concrete implementations makes the process easier. The interfaces also provide good guidelines for how new features could be implemented.

The **free for commercial use** requirement is fulfilled since both Scala and the libraries used to develop the system are free for commercial use.

The **low memory footprint** requirement has not been solved when compared to the model from [1] as their memory usage were similar. Even with high memory usage, the tool was still more usable on a local machine compared to the model from [1]. However, it can be argued that this model is more advanced since there are more components and more message queues. But the messages themselves could probably be represented in a more space-efficient way, which could lead to a lower memory footprint.

The **parallel simulation** requirement was solved as each experiment only runs on one thread and multiple experiments can be run on separate threads and this is all contained within the system.

The **integration with development environment** was fulfilled as the tool is written in Scala and that it integrates well with IntelliJ. The development environment also made it easier to make and run tests for the tool. This makes it easier to implement new features as they can be tested with little to no effort. In [1] every component was tightly coupled and it was difficult to test.

The **progress tracking** requirement was about having information about how far the simulation has run. For this tool progress was the number of completed ticks, displayed as a percentage of the total amount of ticks. This is possible as it is known beforehand when the experiment stops e.g. an experiment can run for 172,800 ticks. The progress tracker was mostly used to give an indication of if the model was stuck, which was useful for development. Uppaal does not show the progress, it only showed how many states per second, which does not indicate how far it was.

5.1 Tool performance

The performance of the tool was okay, for the tests that were run. The memory usage was about 10 - 11 GB and would grow even more if more sensors were added, making it difficult to test larger models. The tests that were ran took about 1 hour and 30 minutes to simulate 30 days, where one tick corresponded to one second. Compared to [1] which took about the same time to simulate 30 days where one tick corresponded to one minute. This is a big improvement, as this tool does more for each tick. Ram is a bottleneck for this tool, the time it took to run a simulation was only fast if there was enough ram. On bigger models, the tool would end up spending more time garbage collecting and that resulted in running simulations on less thread being faster.

6 Improvements

6.1 Memory Improvements

A possible improvement that could improve ram usage would be to change how messages are stored. As it is now each message is copied to each device, and a message is an allocated object with some fields. Instead of copying a message a message could be sent to a message lookup table and then only the message id would be in the message queues. By doing that only the message id which is an integer would have to be copied, this is similar to how the cache was handled as it also works as a central lookup table. Then all messages would never be copied and just accessed based on their ids. However, this approach has to be analyzed by a profiler to determine how messages should be removed from the message database. Removing messages in batches for the cache was faster than removing them at the earliest possible time as the message became inactive.

6.2 Model Improvements

This tool has a lot of settings. However, some of these settings could be made dynamic such that the model would be able to adapt to current conditions. Some of the settings from Table 1 could be dynamically set which could potentially improve the system. These would also be implemented as unique parameters for each gateway. Examples of parameters that would adapt dynamically would be the resend attempts and waiting times.

7 Conclusion

To conclude this project a simulation tool was built, which used the synchronization algorithm. Based on the tests the algorithm seemed to work with different amounts of gateways and based on the results a lower amount of gateways per sensor might be preferable. It is possible to define test setups and define the locations of the devices and the schedules for when waveforms are sampled. This can all be used to model a real system and thereby provide metrics of how it would perform. The performance of the tool is good compared to the model from [1] as it could process 60 times more ticks while doing more each tick. However, it used a lot of memory and it grows fast when adding more devices. Memory is the bottleneck for this tool and to simulate larger sensor networks more memory optimizations are needed.

References

[1] Søren Ebbesen Bundgaard. Modelling an edge computing datasynchronization system. *Aalborg Universitet*, Jan 2021.

[2] Alexandre David, Kim G. Larsen, Axel Legay, Marius Mikučionis, and Danny Bøgsted Poulsen. Uppaal smc tutorial. *International Journal on Software Tools for Technology Transfer*, Jan 2015.

[3] Hexastate. Hexastate. https://hexastate.com/. Accessed: 13-06-2021.

[4] Mathworks. Bluetooth mesh flooding in wireless sensor networks. https://se.mathworks.com/help/comm/ug/bluetooth-mesh-flooding-in-wireless-sensor-networks.html. Accessed: 15-02-2021.

[5] Mathworks. Math. graphics. programming. https://se.mathworks.com/products/matlab.html?s_tid=hp_products_matlab. Accessed: 15-02-2021.

[6] Nicholas Metropolis and S. Ulam. The monte carlo method. *Journal of the American Statistical Association*, 44(247):335–341, 1949. PMID: 18139350.

[7] nsnam. ns-3 a discrete-event network simulator for internet systems. https://www.nsnam.org/. Accessed: 20-02-2021.

[8] Kartik Patel. Bluetooth low energy. http://kartikpatel.in/ns-3-dev-git/group__ble.html. Accessed: 20-02-2021.

[9] Bluetooth Special Interest Group (SIG). Bluetooth mesh profile. https://www.bluetooth.com/specifications/specs/mesh-profile-1-0-1/. Accessed: 21-02-2021.

[10] Bluetooth Special Interest Group (SIG). An intro to bluetooth mesh part 2. https://www.bluetooth.com/blog/an-intro-to-bluetooth-mesh-part2/. Accessed: 01-03-2021.

[11] Jon Gunnar Sponås. Things you should know about bluetooth range. https://blog.nordicsemi.com/getconnected/things-you-should-know-about-bluetooth-range. Accessed: 21-02-2021.