Debugger and Visualizer for a Shared Sense of Time on Batteryless Sensor Networks

Design Document

Team 15 (May 2021)

Client/Advisor

Dr. Henry Duwe

Team Members

Adam Ford - Report Manager

Allan Juarez - Scribe

Maksym Nakonechnyy - Design Lead

Anthony Rosenhamer - Facilitator

Quentin Urbanowicz - Test Engineer

Riley Thoma - Project Manager

Email: sdmay21-15@iastate.edu

Website: http://sdmay21-15.sd.ece.iastate.edu

Revised: November 15, 2020 (Final)

Executive Summary

The proposed solution is to create a set of applications: a desktop application that will simulate a batteryless sensor network and a web-application that will allow the user to visualize and debug the shared sense of time across this network. The system will also provide functionality to simulate a sensor network, and visualize and debug these simulated networks.

Development Standards & Practices Used

- The backend application will be developed following the layered architectural style to allow for code maintainability and ease of implementation and consistency. The presentation layer will provide RESTful APIs that will be used to provide access points for the frontend application. It will also implement communication with the simulator.
- This communication will use HTTP requests. It will follow the conventions defined in the appropriate standards for HTTP [1].
- We will use terminology that is standardized by IEEE to ensure that we effectively communicate the meaning of our work [2].
- We agreed to use standard coding conventions for the programming languages to make the code more readable and documentable. Where possible, we will use tools to automate documentation.
- We decided to use GitLab for both source control and task tracking.
- We decided to follow agile methodology for our project aiming to deliver functionality incrementally. We will communicate with the client after each iteration to make sure that the product satisfies the requirements and client's needs.

Summary of Requirements

- Display the live status of time on the sensors
- Store past data
- Simulate the data in the same format as real data
- Accept a seed value for pseudo-random simulation
- "Replay" past data
- Display up to 15 sensor nodes on screen

Applicable Courses from Iowa State University Curriculum

- COM S 227: Object-Oriented Programming
- COM S 228: Introduction to Data Structures
- COM S 309: Software Development Practices
- COM S 327: Advanced Programming Techniques
- COM S 363: Introduction to Database Management Systems
- CPR E 288: Embedded Systems I
- CPR E 458: Real-Time Systems
- EE 201: Electric Circuits
- ENGL 314: Technical Communication
- SE 319: Construction of User Interfaces
- SE 329: Software Project Management
- SE 339: Software Architecture

New Skills/Knowledge acquired that was not taught in courses

- SimPy
- pytest
- UNIX-domain sockets
- ExpressJS
- MongoDB database
- React JS
- Jest
- Selenium
- MirageJS

Table of Contents

1 Introduction **6**
1.1 Acknowledgement 6
1.2 Problem and Project Statement 6
1.3 Operational Environment 7
1.4 Requirements 8
1.5 Intended Users and Uses 8
1.6 Assumptions and Limitations 9
1.7 Expected End Product and Deliverables 9

2 Project Plan **10**
2.1 Task Decomposition 10
2.2 Risks and Risk Management/Mitigation 12
2.3 Project Proposed Milestones, Metrics, and Evaluation Criteria 12
2.4 Project Timeline/Schedule 13
2.5 Project Tracking Procedures 15
2.6 Personnel Effort Requirements 15
2.7 Other Resource Requirements 15
2.8 Financial Requirements 15

3 Design **16**
3.1 Previous Work and Literature 16
3.2 Design Thinking 16
3.3 Proposed Design 16
3.4 Technology Considerations 21
3.5 Design Analysis 22
3.6 Development Process 23
3.7 Design Plan 23

4 Testing **24**
4.1 Unit Testing 24
4.2 Interface Testing 25
4.3 Integration Testing 25
4.4 Acceptance Testing 26
4.5 Results 26

5 Implementation **27**

6 Closing Material **29**
6.1 Conclusion 29
6.2 References 29
6.3 Appendix 30
6.3.1 Appendix A: Lifecycle of a Node 30

List of figures

Figure 1.1	High-Level System Diagram
Figure 2.1	Task Graphs
Figure 2.2	Design Gantt Chart
Figure 2.3	Simulator Gantt Chart
Figure 2.4	Backend Gantt Chart
Figure 2.5	Frontend Gantt Chart
Figure 3.1	System Block Diagram
Figure 3.2	Simulator Class Diagram
Figure 3.3	Backend Block Diagram
Figure 3.4	Frontend Block Diagram
Figure 5.1	Dashboard Prototype. Node States
Figure 5.2	Dashboard Prototype. Communication

List of tables

Table 2.1	Task Decomposition

List of definitions

Backend	Application for storing and processing time data to be exported or sent to the frontend, receives data from the simulator
Frontend	Web application for viewing the sensor network's time data
GUI	Graphical User Interface
Node	Sensor node in the network
Sniffer	Component that reads messages being sent or broadcasted between sensor nodes
Simulator	Application that produces time data and simulates the time data that would be coming from a sensor network, sends data to the backend and a trace file

1 Introduction

1.1 Acknowledgement

We would like to acknowledge our faculty advisor and client, Dr. Henry Duwe, for his guidance and expertise throughout the course of this project. We would also like to thank Vishal Deep for his research on distributed batteryless timekeeping systems, which forms the need for our project and without which our work would not be possible. In addition, we would like to extend our gratitude to Iowa State University for providing access to software and hardware resources for testing and development, and for providing our team this opportunity to contribute to the ongoing development of batteryless intermittent computing technologies.

1.2 Problem and Project Statement

Problem Statement

In distributed embedded computing systems, reliable timekeeping is essential. Typical methods of keeping track of time require continuous power to function, usually from a battery. However, the use of batteries in certain applications, like distributed sensing, may pose significant challenges, particularly in circumstances where replacing batteries is infeasible or impossible.

For such use cases, batteryless devices, which draw the energy required for their operation solely from ambient sources, present numerous advantages; the lack of a consistent power source, however, necessitates a new method of keeping track of time and ensuring that sense of time is shared between all nodes in a system.

A graduate research group at Iowa State University, led by Vishal Deep, has demonstrated designs for real-time embedded clocks which are capable of keeping time without the need for a continuous source of power. However, due to variations in manufacturing, susceptibility to noise, and the added complexity of keeping a sense of time synchronized across a distributed system, visualizing and debugging these clock systems is exceedingly difficult.

To optimize designs and detect unknown bugs, a simulation tool capable of modeling the interactions within a distributed network of clocks and determining each node's resultant sense of time is required. A debugging system capable of probing and visualizing the state of the network and examining the dependencies between each node's sense of time is also needed.

Proposed Solution

Our team aims to develop a pair of software tools for debugging the shared sense of time across a network of distributed clocks: a simulator, which models and records time estimates across a network over time, and a visualization dashboard, which utilizes data from simulations, or a yet-to-be-developed hardware sniffer, to visualize these changes and analyze the interconnections between individual nodes. Our simulator will utilize optimizations where possible to reduce computation time and enable scalability across larger numbers of nodes. The dashboard will utilize a locally-hosted web application to ensure compatibility across all operating systems and enable use by multiple users simultaneously. With these deliverables, we hope to create a set of utilities which

are powerful enough to achieve the levels of accuracy and precision required for academic research, yet flexible enough to adapt to design changes and enable collaboration with minimal effort. An overview of the system is shown in Figure 1.1 below.

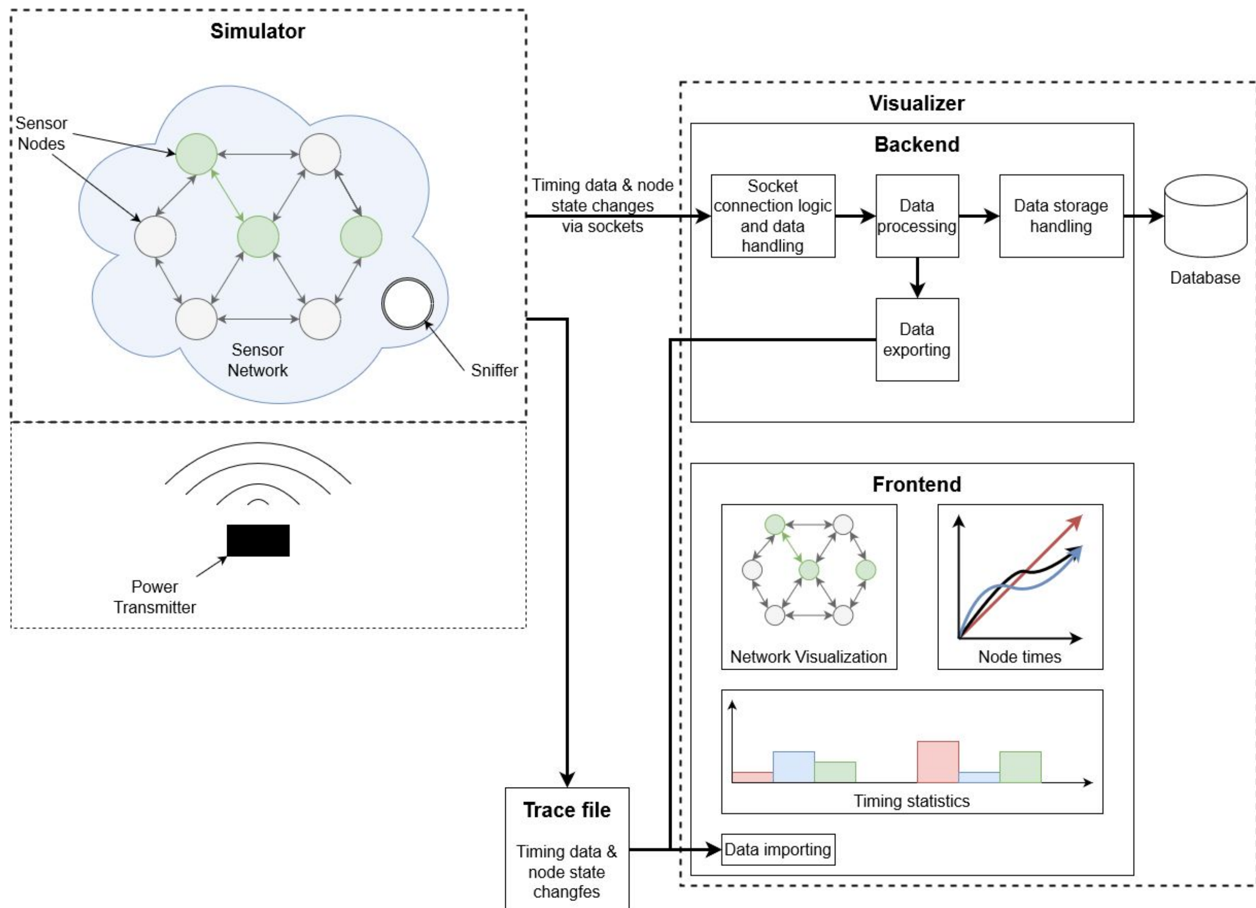


Figure 1.1. High-Level System Diagram

1.3 Operational Environment

Due to the necessity of generating, storing, and analyzing large sets of data, our team will be developing our software tools with high-performance computing hardware in mind. The simulator will be tailored to run natively in a Linux environment and will be operated primarily through a command-line interface. The visualization dashboard will run as a locally-hosted web application and thus will require a web browser with support for modern web standards such as HTML5, CSS 4, and JavaScript. Since the dashboard web app will be locally hosted, and since interaction with sensitive information is not required, there are no specific concerns for data security or privacy compliance.

1.4 Requirements

Functional requirements:

- The system shall store past data.
- The system shall monitor which nodes are currently communicating.
- The system shall monitor which nodes have scheduled communication at next on-time.
- The system shall record any successful/unsuccessful communications between nodes.
- The simulator shall generate the data in the same format as real data.
- The simulator shall accept a seed value for pseudo-random simulation.
- The visualizer shall display the live status of time on the sensors.
- The visualizer shall display up to 15 sensor nodes on the screen.
- The visualizer shall display previously saved sample data.
- The visualizer shall visualize the statistics of system communication.
- The visualizer shall display the data (e.g. on-time) transferred during communications.
- The visualizer shall display events of two types: “communication” and “node-time-query”.

Non-functional requirements:

- The system shall be modular to allow for maintainability.
- The system shall not lose any sensor readings.
- The simulator shall run natively in a Linux environment.
- The simulator shall maintain sub-second accuracy of timing.
- The simulator shall produce on-time/off-time data from a user-provided function.
- The visualizer shall update node status every second.
- The visualizer shall be implemented as a web-application.
- The visualizer shall be accessible from any device or OS.

Economic requirements:

- The system shall utilize hardware provided by the client.
- The system shall be built using open source software to minimize costs.

Environmental requirements:

- The system shall run in a controlled environment, so there aren’t any environmental requirements.

1.5 Intended Users and Uses

The system shall support research group members. Their team requires this system for their research into timekeeping for batteryless sensor networks. The system shall provide the following functionality to users:

- View the state of sensors in real time.
- Add a sensor to be tracked.
- Remove a sensor.
- Simulate a network.
- “Replay” past data for the network as a whole.
- “Replay” past data for an individual node.
- See the statistics of system communication.

- Provide a seed value for pseudo-random simulation.
- Customize the dashboard.
- See the propagation of error from one node to another.
- Log the errors.
- Load previously saved logs to be displayed.
- Export past and live data.
- Import past data.

1.6 Assumptions and Limitations

Assumptions

- We will be provided data from the sniffer to emulate it.
- The minimum number of nodes to be simulated is three.
- The project will only be used by Dr. Duwe and his graduate students.
- The visualizer will be used on a screen that is of size at least 24".

Limitations

- With no budget right now, we will have to find some ways to host our project and find a database for the project without spending anything.
- With the pandemic going on, we are unable to go into the lab and test our project, or get fresh data.
- There is no actual sniffer, so it will be hard to do an integration test of the visualizer with the sniffer.

1.7 Expected End Product and Deliverables

Design Deliverables

By the end of the first semester we are expected to complete and present our design document. The design document will have all our procedures and plans to complete our full system for the following semester. We are also expected to turn in our bi-weekly status reports and lightning talks.

Development Deliverables

The expected end product for this project is to have a simulator and a dashboard. The simulator shall provide data similar to an actual sniffer node from the hardware project. These goals will be met at the end of the spring semester. The dashboard will be expected to display three nodes at a minimum, but up to 15 being the end goal. The dashboard will be accessible from any computer that Dr. Duwe and his Graduate Students have. In the dashboard we will add panels that give information about all the nodes, including the error each node currently has and what time it is displaying.

This development process will also result in a deliverable of the code documentation. This will detail how the code works and how it should be used in specific. Additionally, our project will have test cases and other deliverables related to testing the product.

2 Project Plan

2.1 Task Decomposition

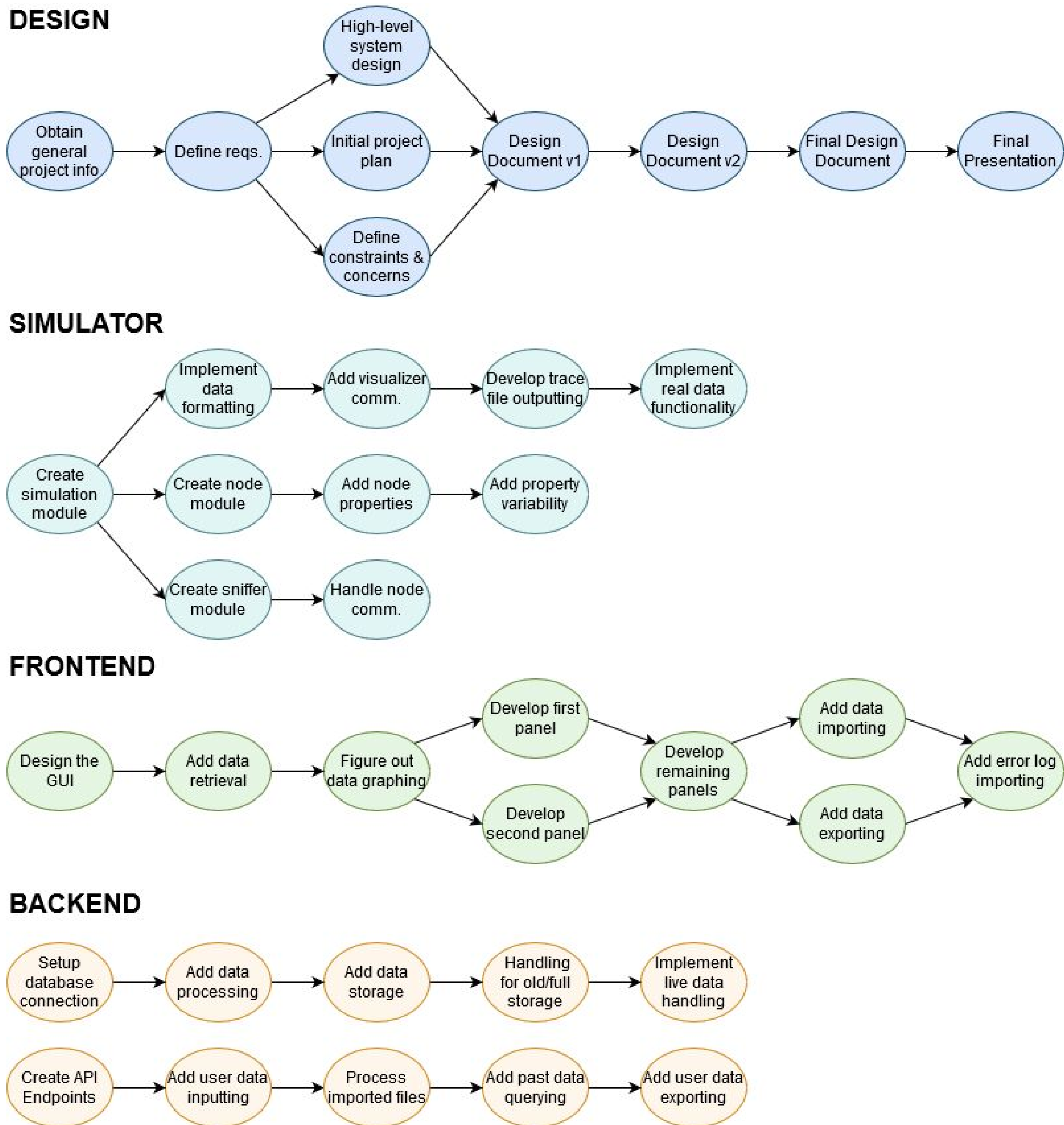


Figure 2.1. Task Graphs

The task graphs in Figure 2.1 show the dependencies between each of the tasks for the four sections of the project (Design, Simulator, Frontend, Backend). The Design and Frontend tasks have stages that allow for concurrent work on tasks that are co-dependencies for the following task.

Component	Task	Allocated Time (person-weeks)	Due By	Risk
Design		**Fall Semester**		0
	Design Doc v1	2	Oct. 4	0
	High-level system design	2	Oct. 4	0
	Initial project plan	2	Oct. 4	0
	General project information	2	Oct. 4	0
	Define requirements	2	Oct. 4	0
	Define constraints and concerns	2	Oct. 4	0
	Design Doc v2	3	Oct. 25	0
	Design Doc Final	3	Nov. 15	0
Simulator		**20**		
	Create simulation module	2		0.5
	Create node module	2		0.25
	Add node properties	2		0.25
	Add variability to node properties	3		0.5
	Create sniffer module	2		0.25
	Handle communication between nodes	1		0.25
	Implement outputting data in the correct format	2		0.25
	Add communication with visualizer via sockets	2		0.5
	Develop trace file outputting	1		0.25
	Implement real data functionality	3		0.5
Frontend		**14**		
	Design a GUI	2		0
	Retrieve and Process data from backend	1		0.5
	Learn how to graph and visualize the data	1		0.5
	Develop the First Panel	2		0.25
	Develop the Second Panel	1		0.25
	Develop the rest of the necessary panels	4		0.25
	Import data to the database	1		0.25
	Export Data from the database	1		0.25
	Import a log of the error in nodes	1		0.25
Backend		**16**		
	Make and setup database connection	1		0.5
	Process real or simulated data into storable format	1		0.25
	Store data in database from backend	1		0.25
	Drop data when database is full, or it is too old	2		0.25
	Provide live current data	1		0.75
	Create API Endpoints	2		0.5
	Accept input data from user	1		0.25
	Process imported data files	2		0.25
	Query and return past data to “replay”	2		0.75
	Query and return past data to user as export	3		0.25

Table 2.1. Task Decomposition

2.2 Risks and Risk Management/Mitigation

The risk level for each task is given in Table 2.1 of the Task Decomposition section. The higher risk items (> 0.5) are analyzed below.

Provide live current data - risk: 0.75

The inherent risk in this task comes from the need for live data to be generated, processed, and displayed. This is a crucial feature for our project as a whole and will be risky in that it involves all three main software components to work together in real time; this complexity could potentially cause issues in meeting the sub-one second latency requirement for data transfer and processing from simulator to backend to frontend. If the latency requirements are not being met, other tasks may be indirectly affected, as code in other software components may need to be cleaned or changed to be more efficient. To mitigate this risk, our team will be conscious of efficiency when coding other parts of the projects.

Query and return past data to "replay" - risk: 0.75

This task is risky because of the difficulty in trying to retrieve old data to then replay it again. We will mitigate this risk by prototyping the replay feature and making sure our design will support it once we reach this task. We will future proof our application to make sure data is saved and easily retrieved for replaying.

2.3 Project Proposed Milestones, Metrics, and Evaluation Criteria

Major Milestones:

- Design Document Final Draft Complete
- Minimum Viable Product (MVP) built
- Simulator Complete
- Frontend Complete
- Backend Complete
- Integration Complete (Final Release)

Minor Milestones:

- Simulator algorithm working and producing data
- Frontend panels are created and working (milestone for each)
- GUI is fully designed
- Importing and Exporting functionality is complete
- Database can store simulator data and provide it to the frontend
- Database can store past data for replayability
- Database set up and working

2.4 Project Timeline/Schedule

The following Gantt charts show the timeline for our project. The tasks are divided into four sections: Design, Simulator, Frontend, and Backend.

All of the Design tasks will be completed in the first semester as shown in Figure 2.2. The work on consecutive design documents will likely begin while the previous document is being completed.

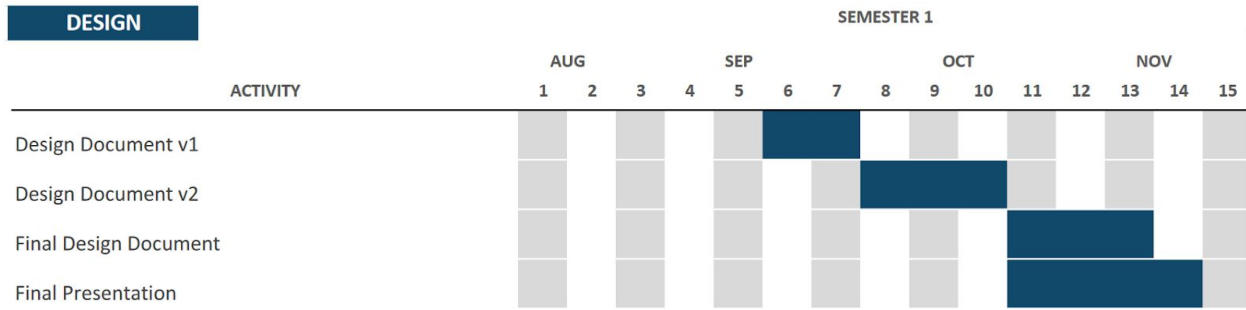


Figure 2.2. Design Gantt Chart

The Simulator tasks (shown in Figure 2.3) include implementing the simulation algorithm, outputting real data, and developing the output data format, and these should be completed around the middle of the semester.

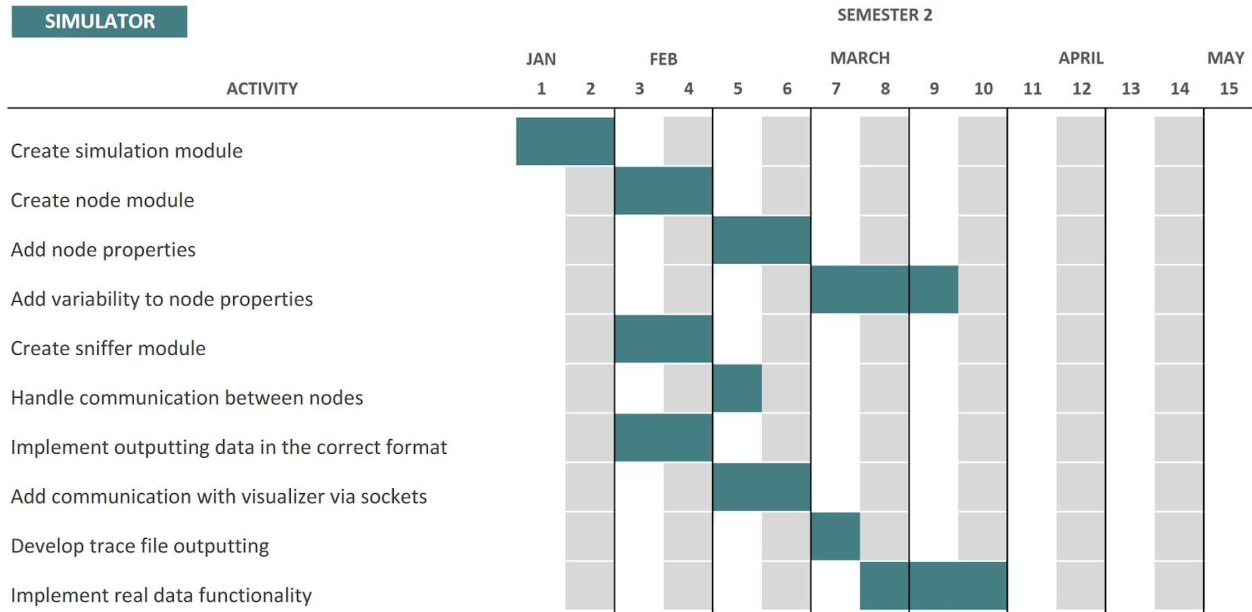


Figure 2.3. Simulator Gantt Chart

The Gantt chart in Figure 2.4 shows the Frontend tasks, which has a few tasks that allow for concurrent work. The Frontend tasks are broken into GUI design, data retrieval, data visualization, GUI panel development, data importing and exporting, and error log importing.

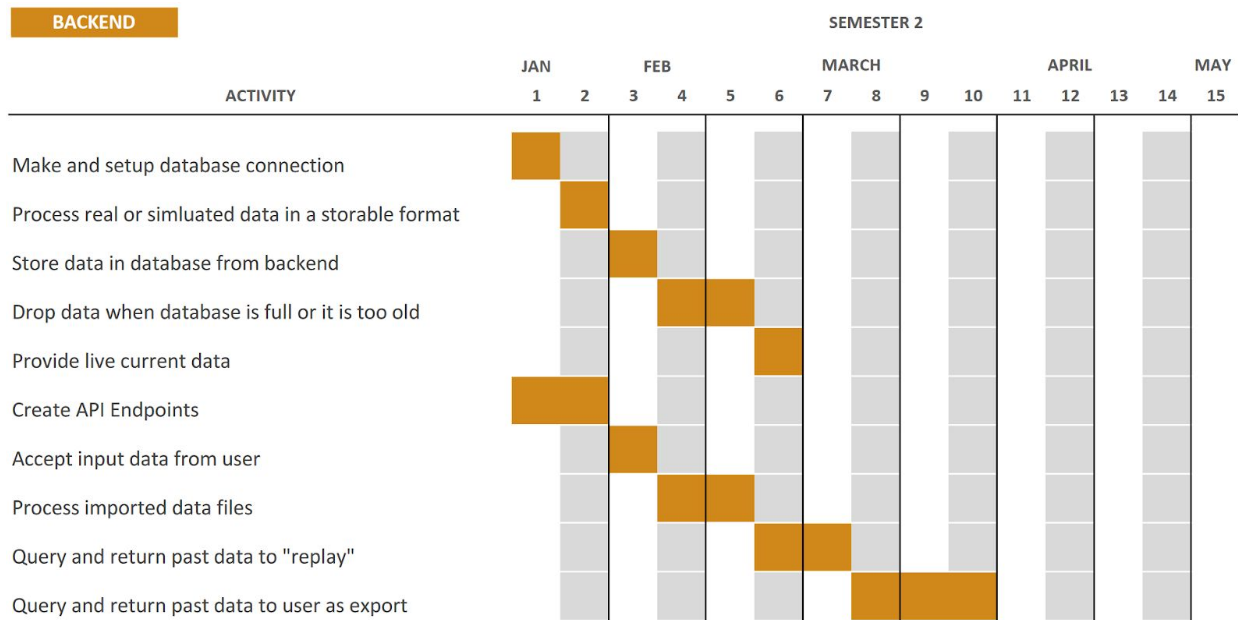


Figure 2.4. Backend Gantt Chart

The final section shows the Backend tasks (Figure 2.5); it is broken into tasks for setting up the database, processing and storing data, handling old or excess data, providing live data, query past data, accepting user data inputs, and providing data exporting. The divisions in the second semester represent the two-week sprints that we will use since we are practicing the Agile approach to software development.

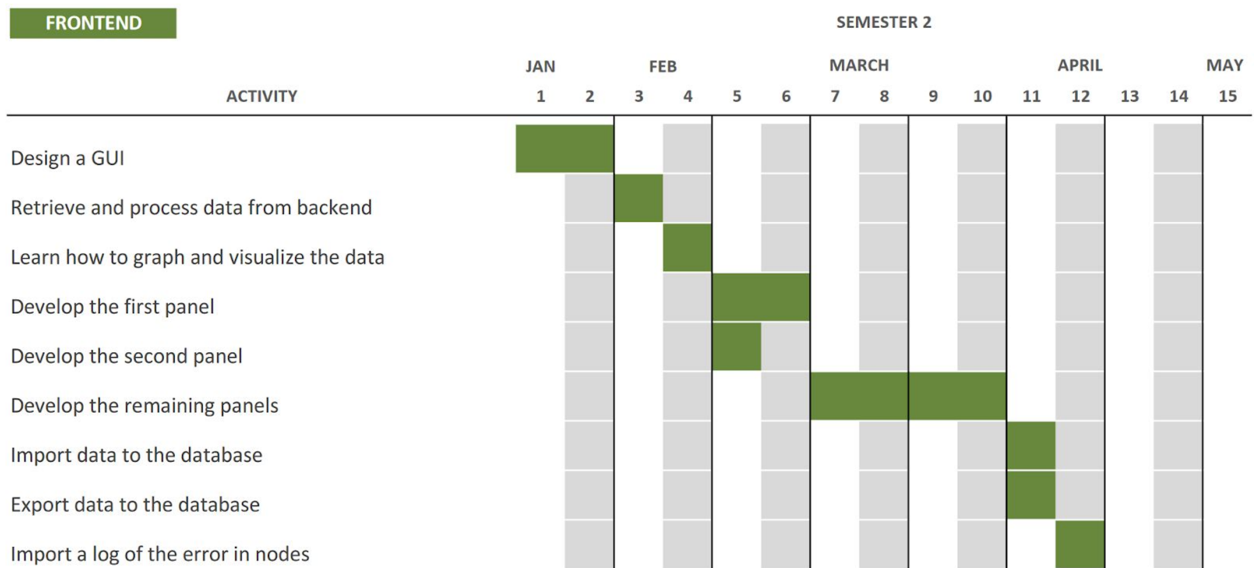


Figure 2.5. Frontend Gantt Chart

2.5 Project Tracking Procedures

Our group will use a combination of online tools to track progress on our project. We have a Slack workspace that we are using for quick communication and schedule coordination. We also have a shared Google Drive folder that we use to store collaborative files, such as Status Reports, Design Documents, Lightning Talks, etc. We will use GitLab for managing code, tracking issues, and monitoring overall progress. We will use GitLab's built-in project management tools, such as the Kanban board, to track our Agile development.

2.6 Personnel Effort Requirements

The effort estimates are shown in Table 2.1. However, each team member's focuses are shown below. These will guide which tasks each team member primarily works on, however adjustments can be made when the project pivots.

Simulator (20 person weeks) - Anthony, Quentin
Frontend (14 person weeks) - Riley, Maksym
Backend (16 person weeks) - Allan, Adam

2.7 Other Resource Requirements

Linux Machines (VMs) - Our simulator is planned to run on a Linux Machine for production, so we will need the ability to develop and test on similar machines.

Software Packages - We will attempt to utilize open source software in as many places as possible, however it is possible that we may need a license to a package if the project specifics require it. The most likely case of this is our live graphing needs, more research will need to be done to determine packages that can serve these and other needs.

2.8 Financial Requirements

There is currently no financial need with the project. Some unknowns that may lead to financial need are hosting, Linux Virtual Machines and package licenses. The assumption is that the university can provide hosting, virtual machines and licenses for our uses. However, if they cannot, we will address those budget concerns when they occur.

3 Design

3.1 Previous Work and Literature

Vishal Deep and Dr. Duwe's research group has shared with us a paper they had published on the topic of maintaining a shared sense of time in a network of batterless nodes [3]. It has been cited in section 6.2.

This paper is primarily composed of information regarding how nodes share time and other background information important to our project. This gives a basis for our team to work off of but does not describe the software system in any way.

The research group has previously built a two-node simulator, which we may use as a reference while working on our design project. There are also many open source live dashboard/visualizer projects that may have a similar design to our goals that we may eventually reference during development.

3.2 Design Thinking

During the "define" stage of our design process, we first discussed Dr. Duwe's research and what needed to be done to facilitate the research. The core issue of our client was the inability to trace the errors that occur from inaccurate time predictions. Dr. Duwe needed software to help him trace all communications between nodes and visualize the propagation of errors from one node to another. These communications would be recorded by a sniffer device, but the device does not exist yet, so a need for a simulator came up. We then created a high-level diagram that depicted our understanding of visualizer and simulator. Now we focus on each component separately to refine the knowledge gained from the discussions with Dr. Duwe.

Much of the ideate phase was completed for us by Dr. Duwe and Vishal Deep as they had already been working on the project and were looking for a specific debugging application to suit their needs. We worked through the ideate phase of our design as a team and through our discussions with Dr. Duwe. One of the notable ideation moments happened when we discussed our high level design diagram. We talked through possibilities for how the different pieces of the project would connect and what would be included in the diagram. When we first showed the diagram to Dr. Duwe, he was able to provide more ideas for us to use, especially with the backend design.

3.3 Proposed Design

The system will consist of two main components: Simulator and Visualizer. The Visualizer, as a web application will consist of two applications: backend and frontend.

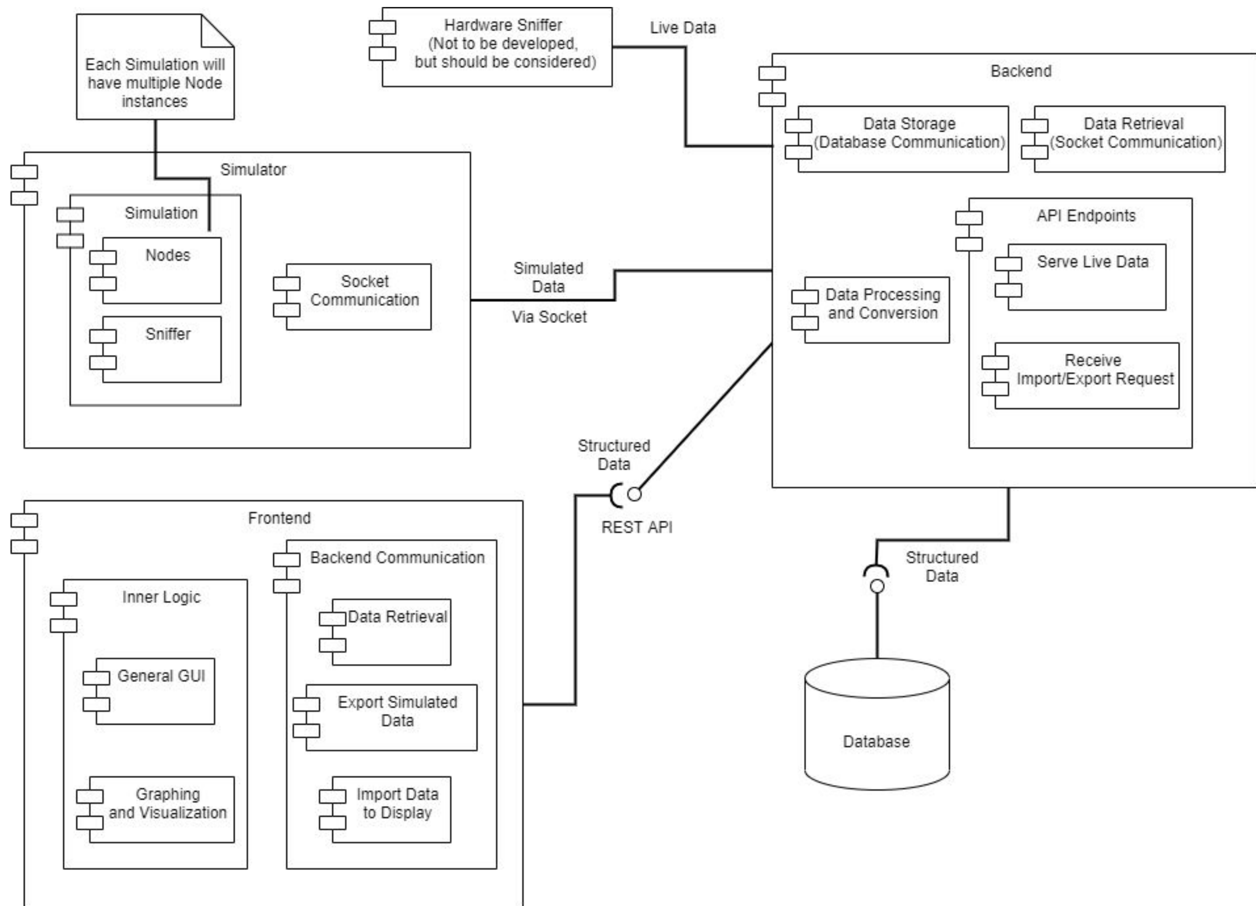


Figure 3.1. System Block Diagram

Simulator

The simulator will run as a server-side application and will be controlled via the command line; it will not have its own graphical user interface. The simulator will consist of the following classes with the attributes listed below:

- Simulation
  - Sets up system with a sniffer and multiple nodes
  - Maintain the true time within the simulation
  - Runs the main loop for the simulation
    - Maintains a priority queue of events (priority is event time)
    - Pops events off queue and processes events at current time
    - Determines time of next (possible) event(s) and inserts them into the queue
  - Outputs trace file
  - Communicates with visualizer via UNIX-domain sockets
- Sniffer (one)
  - Uses the true simulation time
  - Polls nodes for their events and estimates of time
  - Compares node times with the true time

- Node (multiple)
  - Maintains its own clock
  - Keep track of when its capacitor power reaches the on or off threshold
    - Keep track of ambient energy being harvested
    - Keep track of rate of power consumption when performing tasks (incl. boot-up)
  - Determine action it takes when on (and when the related event will occur)
  - Keep track of state across periods of off-time using system checkpointing

Each of these classes will exist in its own Python file where its attributes and behavior are defined. The simulation class is the core component of the simulator. It handles command-line arguments, using them to perform the initial setup of a simulation and executes the main loop, which tracks the true time within the system and calculates variation through the use of event-driven simulation logic using the SimPy module. The sniffer, controlled by the simulation, will periodically collect data from the sensors (nodes) in the simulated network.

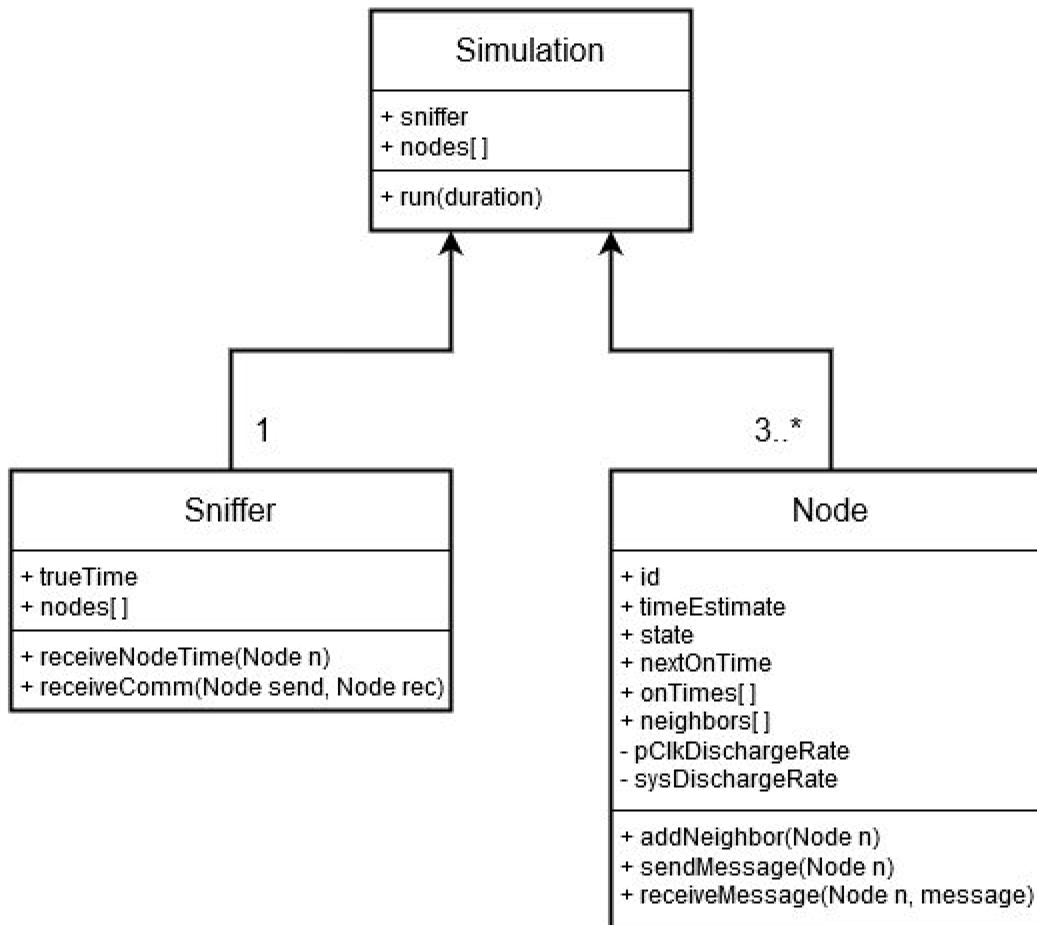


Figure 3.2. Simulator Class Design

The nodes will cycle through three main states: OFF, BOOT_UP, and ON. In the OFF state, the nodes' system capacitor will harvest power at a variable rate affected by environmental factors.

Once the system capacitor reaches its ON threshold voltage, the node will enter the BOOT_UP state, setting up the system over some interval. At the end of the BOOT_UP state, the node will measure the voltage of the persistent clock capacitor and calculate its new sense of time based on this value. Immediately afterwards, it will transition to the ON state, where it will send event messages to other nodes, reflecting its shared sense of time estimates. During the ON state, the persistent clock capacitor will harvest power. This continues until the system capacitor reaches the OFF threshold voltage, at which point the node transitions back to the OFF state and begins charging again. More details can be found in Appendix A.

Backend

The backend will run server side and be accessed by our frontend. The backend will retrieve all the data output from the node simulator. It will then store that data into our database and also output that data for the frontend to access. There will be API endpoints so the frontend can make HTTP requests and access data output from the simulator. The backend will have a method of retrieving data from the sniffer or simulator and displaying the live stream of data within a second. In the backend, the data will be processed and cleaned so that the data will be ready to be displayed by the frontend and be stored in the database. Additionally, the backend will have functionality to handle the imports and exports of trace files from the user.

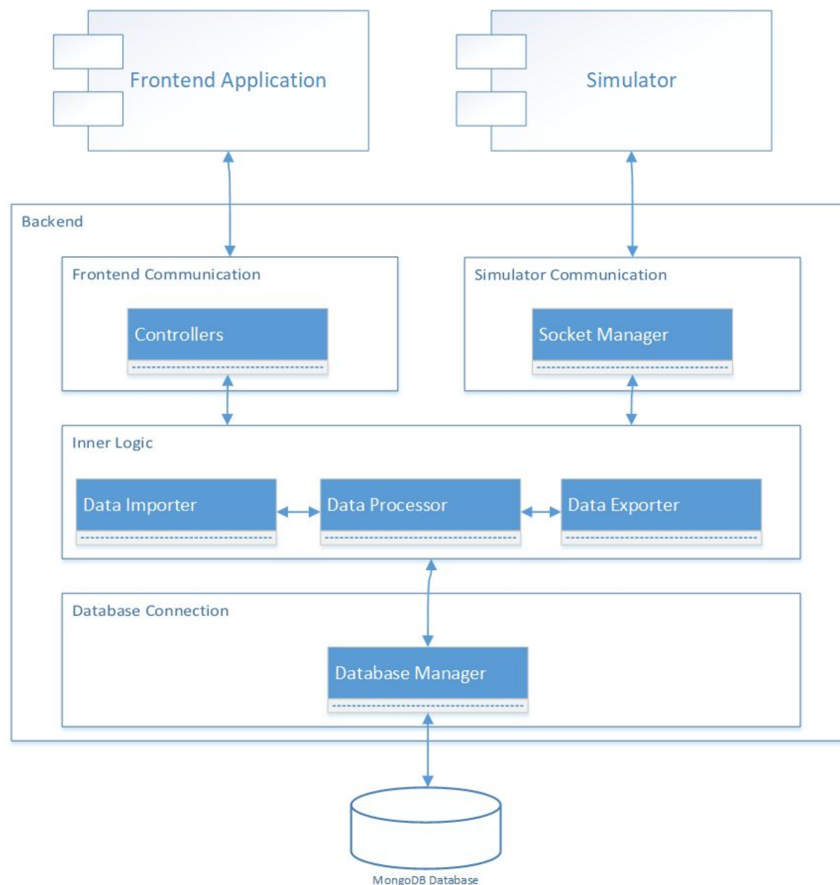


Figure 3.3. Backend Block Diagram

Frontend
The frontend will run as a web-based application that will allow it to be accessed from any device. It will get all the necessary information it needs to run the visualizer from the RESTful APIs provided by the backend. Frontend will send HTTP requests to the backend requesting new data every second to update node information. There will be a global clock displayed for the real time of the system and an individual clock for each of the sensor nodes displayed. The user will be able to start/stop the visualization of the data, move backwards in time to view data again, and see up to 15 sensor nodes displayed on the screen. Instead of live visualization, the user will be provided with an option to upload and visualize previously saved data. Statistics will be displayed to the user in an appealing way. Each communication between two nodes can be examined in greater detail. The application will be divided into two main components - Presenter/Adapter and Request Sender. The Presenter/Adapter component will be responsible for maintaining the state of the application as well as processing the data to be displayed. The Presenter/Adapter is logically one component because React will automatically rerender what is displayed when the state of the component changes. The Request Sender component will be responsible for sending HTTP requests to the backend application and will pass on the data to the presenter through callback functions.

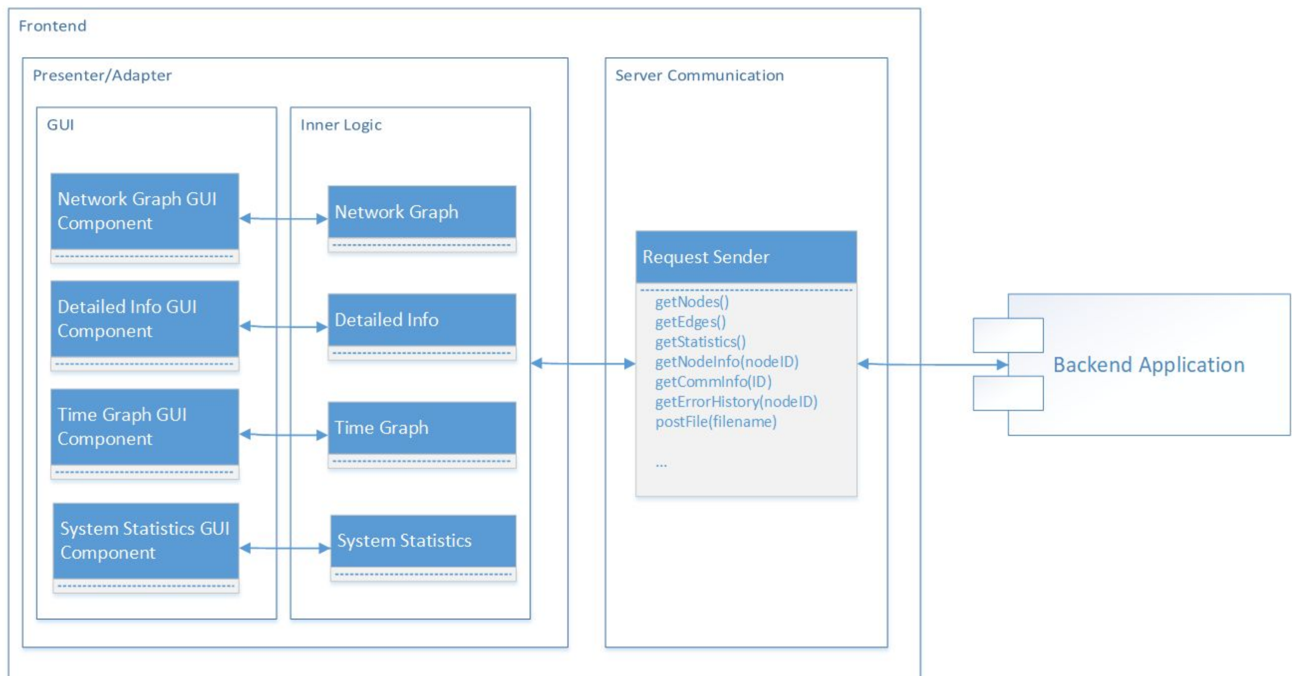


Figure 3.4. Frontend Block Diagram

3.4 Technology Considerations

Simulator

- Python programming language
    - Python standard library
    - socket library
    - SimPy
    - pytest
    - pytest-mock

The simulator will be built in Python since it is simple and powerful. A major strength comes with the availability of Python libraries, including the standard library and the socket module that will enable communication between the simulator and the visualizer. One alternative that we considered was using C++ rather than Python to implement the simulator, but we decided on Python because the research team had more experience with Python. We also looked at using the WebSocket protocol as an alternative to UNIX-domain sockets, but we went with UNIX-domain sockets because they are better suited for Python through the socket library which has more support than most WebSocket libraries for Python. We are also using the SimPy module for running the discrete-event simulation required for this application [4]. SimPy has many advantages, mainly being able to run extended events more quickly; this will enable us to simulate the life of a sensor network efficiently. For unit testing, we plan to use the pytest framework because it is well-supported and easy to use [5]. The pytest-mock library will allow us to mock up the backend to simulate communication [6].

Backend

- ExpressJS (NodeJS)
- MongoDB database
- Jest

These languages will allow us a simple yet robust methodology for proving REST APIs to our frontend. Utilizing the Express framework will allow us to bring up the backend quickly, and without too much learning curve being written in NodeJS [7]. We considered Django (Python) and Phoenix (Elixir) as other alternatives, but both were based on an MVC pattern which were irrelevant for our use cases. MongoDB was chosen over other databases including flavors of SQL or CouchDB as it excels at high volumes of read/writing which will be prevalent in the live portion of the application [8]. Although our knowledge is stronger in other databases, MongoDB's benefits outweighed our strengths [9]. Jest is simple yet effective at mocking API calls. With this technology we can write many tests to cover all our code and make sure our backend is working properly at all times.

Frontend

- HTML & CSS
- Javascript - ReactJS
- Selenium
- Jest
- MirageJS

For the frontend application, we decided to use HTML & CSS and JavaScript since these are the most common and the most suitable technologies for developing web applications. Using these languages, we will be using fairly basic web development tools, but they can be quite powerful. On top of that, we decided to use the ReactJS framework that provides numerous powerful tools and libraries for improved user experience with applications. We also considered using AngularJS as the development framework instead of ReactJS, but our team has no prior experience with AngularJS, and the learning curve vs. added benefit does not seem worth it at this time. Another consideration was to use WebGL for displaying the nodes, but this project does not require extensive use of graphics, so we decided that it was not appropriate for our purposes.

3.5 Design Analysis

Simulator

Our proposed simulator design should work as planned. It contains definitions for each of the sub-modules of the simulator that represent the physical components - the nodes and sniffer. This will allow us to use an object-oriented design to model the sensor network and implement a working simulator using an event-driven architecture. We will iterate over our design to add functionality to the simulation as we give the components more features and define more complex behaviors, such as variable power consumption. The modularity of our design should be a strength that will allow us to build and test separate components to ensure each one works before connecting them together.

Backend

Our backend design will work. We have had some experience with making a backend previously, and considered those experiences when discussing this design. The design currently meets our needs. It will take in data from the simulator and store it into the database and display it to the frontend. We will most likely discuss the design and to decide if we should add more packages or libraries to make the backend more efficient. Our backend will be very efficient in taking and storing data. A weakness we might have will be with live data. We are not sure how fast live data can be output with a simple API call, so we will have to do more research on that to meet the live data criteria.

Frontend

We believe our general design for the frontend will work based on the weekly conversations we've been having with Dr. Duwe to learn about the systems and the implementation he is looking to have. The design meets all of our current requirements, and leaves room for growth as needed. We will definitely be iterating over the UI design of the application as we come to find what looks the best, functions the best, and provides the best user experience, but our fundamental system design from section 3.3 should not change.

3.6 Development Process

We decided to use an Agile development methodology, specifically the Scrum variation with two-week sprints. This approach will allow us to work iteratively on the project, so we can build on existing developments with each consecutive sprint. By breaking the project into smaller sections, it will make our development process more manageable.

3.7 Design Plan

Figure 1.1 shows the high level design of our three main modules, the simulator, the visualizer and the backend. The use cases of the user all end at the visualizer, the product that will be in their hands, but begin at the simulator. The simulator must produce data that closely mimics a real system of nodes, which is then communicated to the backend. The backend must receive and process the data before storing it and sending it to the frontend for the user. This is the general flow of data and dependencies; however, there are a few alternate use cases for importing and exporting data. These use cases include the ability to upload and replay a trace file rather than view current simulator data, and the ability to output the trace file that is currently being viewed for future use and replays. The user may opt to import their own trace file to view, which eliminates the backend's dependency on the simulator and instead just replays the data to the frontend. An export does not change the module's dependencies, but does require the communication of a larger data set between the backend and frontend.

4 Testing

4.1 Unit Testing

Simulator

We will individually test the simulation, sniffer, and node modules to ensure each unit works on its own. We will also execute unit tests for the socket connection logic. The following shows a list of tests for each unit of the simulator:

- Simulation
    - Setup of a simulated sensor network with different numbers of sensors
        - Verify that individual nodes can be created for a variable number of sensors
        - Verify the duration of the simulation matches the duration given in the input
    - Collection of data from a simulated sniffer
        - Verify that the simulation receives the data inputted by the sniffer
    - Processing of events and calculations of subsequent events
        - Ensure that the event processing occurs at the calculated time
    - Data outputting to ensure the format is correct
        - Ensure the output file is in the same format that should have been written
- Sniffer
    - Gathering data from a node event
        - Verify that the data from the sniffer matches the data emitted by the node
    - Calculating error from a node's sense of time
        - Verify that the error is within reason (similar magnitude - ms or sec)
- Nodes
    - Sending an event message
        - Verify the data at the receiving node matches the data at the sending node
    - Transitioning between stages
        - Verify that the node switches to the correct state when power is low
- Sockets
    - Establishing a connection
        - Check to make sure no error occurs when the connection is made
    - Sending data over a connection
        - Verify that data sent matches data received
    - Serializing and de-serializing event data
        - Verify the event data matches the event that has occurred

Because we are developing the backend (which receives data through the socket connection) concurrently, we plan to use mocking to simulate the reception of data so the simulator can be tested without a finalized backend. Mocking will be done using the pytest-mock module.

Backend

Our testing package Jest will allow for mocking of API calls to give full coverage of the provided endpoints. This will be the majority of the testing of the backed, it will just have to be developed to be as comprehensive as each test requires. Internal functionalities of the backend that are not

externally provided will be unit tested with Jest as well. Unit testing will be especially valuable during integration testing. If tests are performed using the Backend but only comparing values from the simulator and frontend it will confirm the functionality of the Backend.

Frontend

Many visualizer components will be tested in isolation from the other parts of the system. Sensor nodes, replay functionality, slide through time, displaying statistics and errors in the node times, displaying active nodes, nodes communicating, etc. The underlying classes to these components, and the RestAPI requests/responses to the backend will also be tested. We will be testing using the Jest Framework for mocking and Selenium Framework for UI unit tests. To test backend connectivity without the backend working yet we will use Mirage JS to mock it.

4.2 Interface Testing

The following list shows the interfaces that need to be tested:

- Simulator and visualizer (frontend/backend)
- Frontend and backend

We will set up interface testing using mocking for the frontend and backend. It will ensure that the data is formatted and handled correctly when it passes from the simulator to the visualizer and between the two components of the visualizer (the frontend and backend).

4.3 Integration Testing

Integration testing will be an important element of this three part system. This test will ensure that all three modules are working together to complete use cases. The system must complete them up to our standards, before we bring similar use cases to our client. The following use cases will need be tested across the system:

- Importing a trace file to the frontend and visualizing the simulation it describes
    - This will confirm the input functionality of the frontend
- Exporting a trace file from the frontend that describes a previous simulation
    - This test verifies that the frontend can properly export simulated data
- Viewing a graph of a node's local sense of time over the course of the simulation
    - This shows the simulator, backend, and frontend can correctly propage the time data to the user for debugging
- Viewing a graph of the error between the node's local sense of time against the true time
    - This confirms the simulator, backend, and frontend can manage the error calculations
- Viewing the sources of error for a node's local sense of time
    - This establishes that the system can reflect which nodes contribute to error
- Creating a sensor network and running a simulation to view the deviation of a node's local sense of time against true time
    - This verifies the simulator is properly built, the backend is functioning, the frontend is displaying correctly, and the connections are working.

4.4 Acceptance Testing

We will involve our client in the process for their acceptance. This will be present after each feature is implemented, as it is important that we do not wait until the end of the project to receive acceptance. Incremental demos will be an important method of receiving acceptance, so we can present our ideas and explain it instead of having our client blindly navigate the application. The use cases listed above as a part of Integration Testing will be used again as tasks for our client to complete.

4.5 Results

As a result of the testing stage of our software development lifecycle we expect to have a system with minimal bugs that satisfies all of the system requirements, and the client agrees that the work on the system is complete. The output of this stage will include testing artifacts (e.g. test results).

5 Implementation

Simulator

Currently, we have been working on a prototype for the simulator. Using SimPy, we have been able to generate a simulation of simple sensor networks. This prototype sets up a sensor network by defining a set of nodes and linking neighbor nodes to allow for simulated communication between nodes. The SimPy module allows us to use discrete-event simulation to jump ahead to future events that happen in the lifecycle of the network. In the current iteration of the simulator, the nodes have a minimal set of properties for defining their behavior. As we continue to implement the design, we will add more characteristics for each node that will allow it to perform closer to a real sensor node. These added properties will incorporate the random error and variation that is seen in the network.

Backend

A simple backend setup has been implemented. It has an endpoint capable of serving a file to the frontend (Trace File). This has been mainly to ensure that setup and use of the frameworks will work. Most of the work in the backend of cleaning data, receiving a socket and providing a socket will be done next semester.

Frontend

The work on the frontend application has begun. The web application is being developed using React and screen shots of the prototype are shown below. We are using the library react-digraph [10] to create nodes and the bi-directional arrows between them. The slider at the bottom of the screen changes the Real Time clock on the bottom right panel, and the nodes on screen will update their color, connections, and information in the top right panel. The graph can be zoomed in and out and moved around. The individual nodes can be moved around and selected to bring up their information in the top right panel. The Import Trace File button allows the user to select a trace file from their file explorer to be processed by the backend and then displayed here in the frontend.

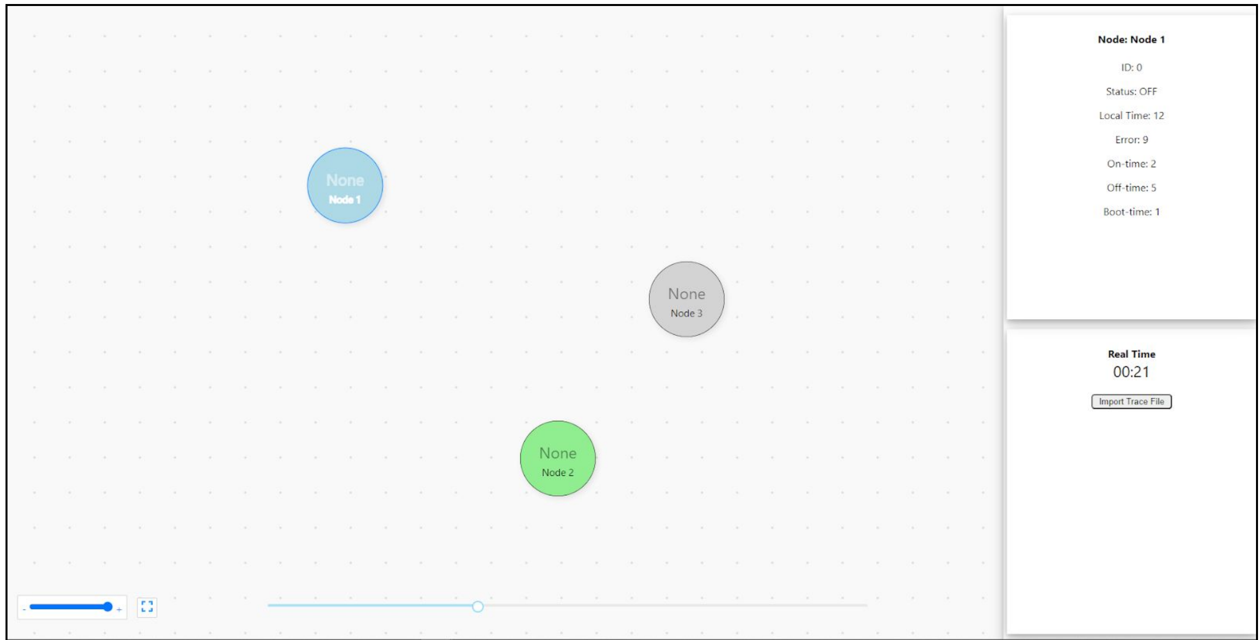


Figure 5.1. Dashboard Prototype. Node States

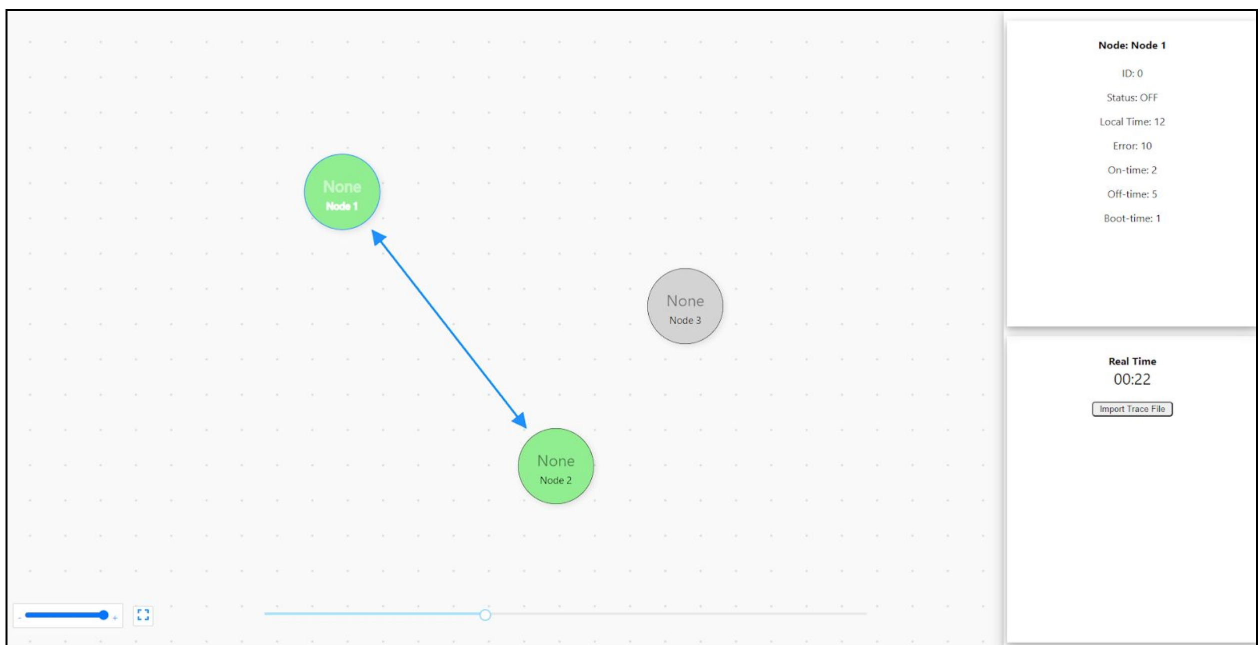


Figure 5.2. Dashboard Prototype. Communication

6 Closing Material

6.1 Conclusion

In the project's current state, we have developed a thorough design plan that gives us a flexible schedule for implementing the project. We have also developed baseline prototypes for the frontend and simulator, which we will continue to iterate on to reach the full functionality of the system. Our goals are to complete the simulator and visualizer applications with the features desired by Dr. Duwe's research team, which means we will continuously incorporate their feedback into our implementation to ensure it suits their needs. Our plan of dividing work based on the separate system components (simulator, frontend, backend) will allow us to efficiently develop this project in the second semester.

6.2 References

[1] Fielding, R., Gettys, J., Mogul, J., Frystyk, H., Masinter, L., Leach, P. and T. Berners-Lee, "Hypertext Transfer Protocol -- HTTP/1.1", RFC 2616, June 1999.

[2] *Systems and software engineering - Vocabulary*, IEEE Standard 24765, 2017. [Online]. Available: https://www.iso.org/standard/71952.html

[3] V. Deep, V. Narayanan, M. Wymore, H. Duwe and D. Qiao: 'HARC: A Heterogeneous Array of Redundant Persistent Clocks for Batteryless, Intermittently-Powered Systems'. Proc. The 41st IEEE Real-Time Systems Symposium 2020 [Accepted].

[4] "SimPy - Discrete event simulation for Python," Read the Docs. Accessed: 11/15/2020 [Online]. Available: https://simpy.readthedocs.io /en/latest/

[5] "pytest: helps you write better programs," Pytest. Accessed: 11/15/2020 [Online]. Available: https://docs.pytest.org/en/stable/

[6] "pytest-mock," Python Package Index. Accessed: 11/15/2020 [Online]. Available: https://pypi.org/project/pytest-mock/

[7] "Express - API Reference", Vers 4.x. 2017. Accessed: 11/14/2020 [Online]. Available: https://expressjs.com/en/api.html

[8] "The MongoDB Manual", Vers 4.4. 2020. Accessed: 11/12/2020 [Online]. Available: https://docs.mongodb.com/manual/

[9] "The MongoDB Node Driver", 2020. Accessed: 11/12/2020 [Online]. Available: https://docs.mongodb.com/drivers/node/

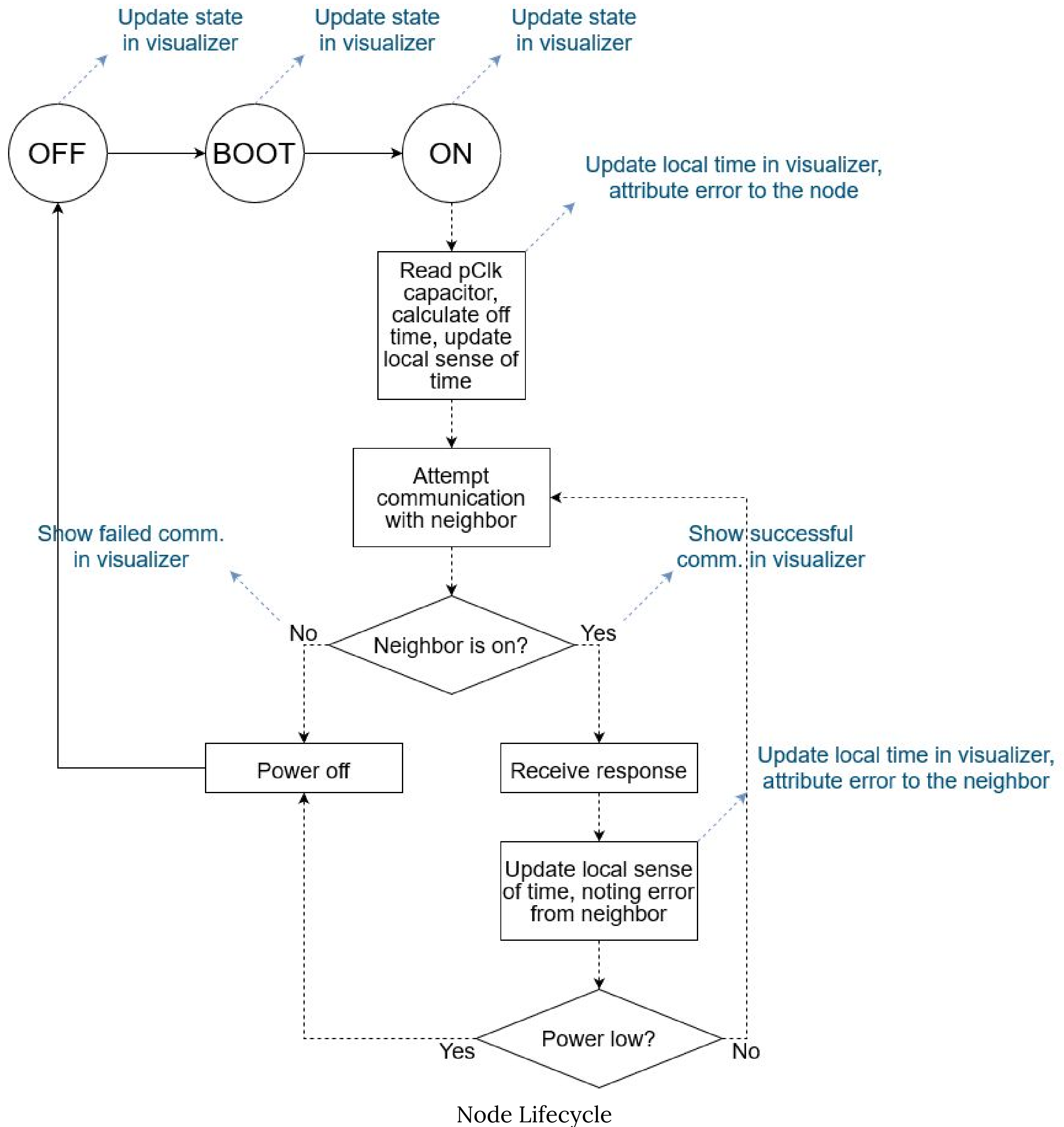
[10] R. Bula, Tim, "React Digraph - CodeSandbox," Codesandbox.io, 23-Jul-2019. [Online]. Available: https://codesandbox.io/s/dkvjy. (Accessed: 15-Nov-2020).

6.3 Appendix

The following supplementary information may be used to better understand the project.

6.3.1 Appendix A: Lifecycle of a Node

The following image shows the states and transitions of a node in the sensor network. The blue arrows represent information being sent from the simulator to the visualizer.



Node Lifecycle