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

Final Report

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: April 25, 2021 (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

- Layered architecture
- RFC 7231: HTTP [1]
- RFC 793: TCP [2]
- IEEE Standard 24765: Systems and software engineering - Vocabulary [3]
- GitLab
- Agile methodology

Summary of Requirements

- Track the local times of sensor nodes
- Generate event trace files of sensor network simulations
- Process and store trace files on the backend for reusability
- Simulate the data in the same format as real data
- “Replay” trace file 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 and Knowledge Acquired

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

Table of Contents

List of figures

List of tables

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 a network
Sniffer	A 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

1.2.1 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.

1.2.2 Proposed Solution

Our team's solution was 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 utilizes optimizations where possible to reduce computation time and enable scalability across larger numbers of nodes. The dashboard utilizes a locally hosted web application to ensure compatibility across all operating systems and enable use by multiple users simultaneously. With these tools, we hope to create a set of utilities that 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.

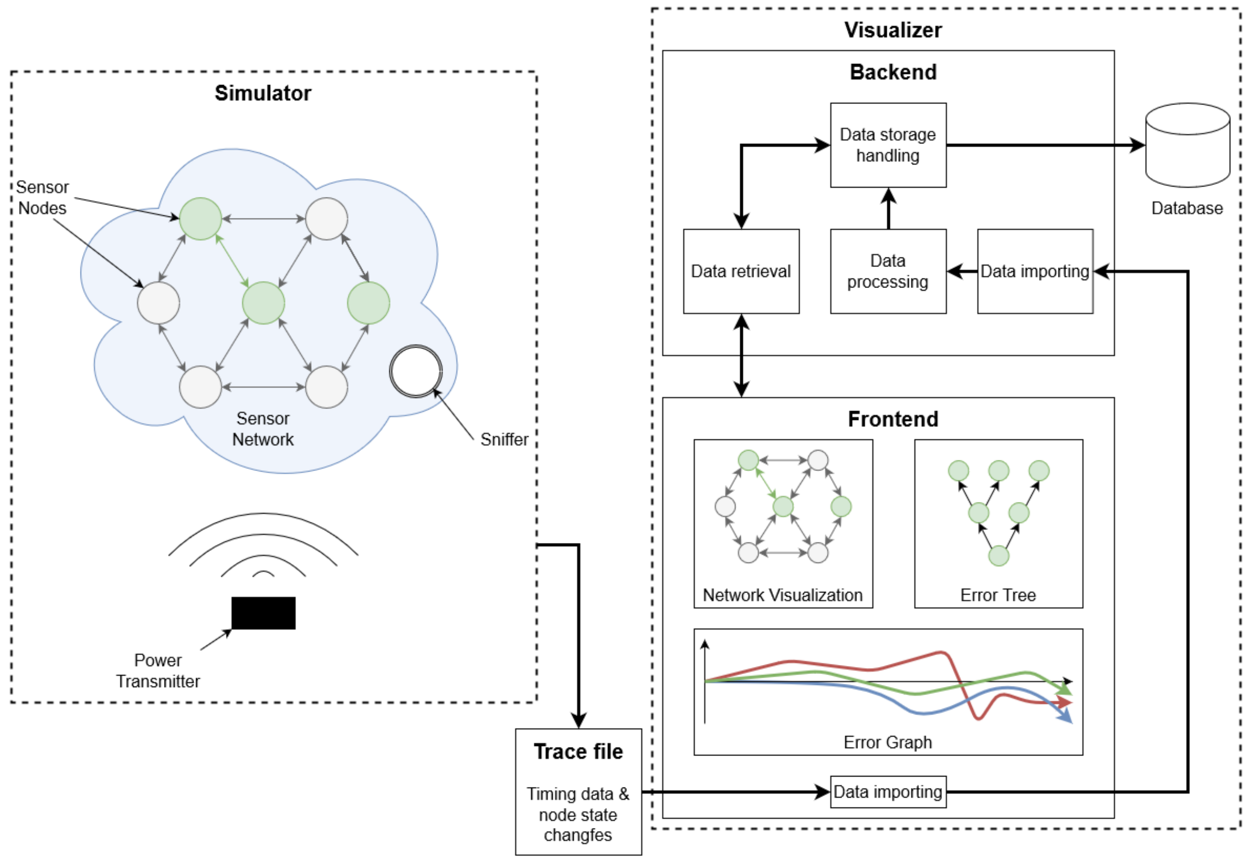


Figure 1.1. High-level system diagram

1.3 Operational Environment

Due to the necessity of generating, storing, and analyzing large sets of data, our team has developed our software tools with high-performance computing hardware in mind. The simulator will run natively in a Linux environment and is operated primarily through a command-line interface. The visualization dashboard runs as a locally hosted web application and thus requires a web browser with support for modern web standards such as HTML5, CSS 4, and JavaScript. Since the dashboard web app is 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 process and store trace files on the backend for reusability.
- The system shall monitor which nodes are currently communicating.
- The system shall record any successful/unsuccessful communications between nodes.
- The system shall track the local times of sensor nodes.
- The simulator shall generate a trace file describing events in the sensor network, such as interactions between nodes.
- The simulator shall produce on-time/off-time data from a user-selected energy model.
- The simulator shall generate the data in the same format as real data.
- The visualizer shall display up to 15 sensor nodes on the screen.
- The visualizer shall display previously saved simulation data.
- The visualizer shall visualize the statistics of system communication.
- The visualizer shall display which nodes can communicate with one another.
- The visualizer shall display whether communications between nodes were successful.
- The visualizer shall display the propagation of time error throughout the network over time.
- The visualizer shall keep track of simulations that can be displayed.

Non-functional requirements:

- The system shall be modular to allow for maintainability.
- The simulator shall run natively in a Linux environment.
- The simulator shall maintain sub-second accuracy of timing.
- The visualizer shall be implemented as a web application.
- The visualizer shall be accessible from any OS.

Economic requirements:

- 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
- Simulate a network.
- Save all events from a network simulation to a trace file.
- “Replay” past data for the network as a whole.
- “Replay” past data for an individual node.
- See the statistics of system communication.
- Interact with the dashboard to view specific statistics.
- See the propagation of error from one node to another via Error Trees.
- Import multiple trace files and move between them.

1.6 Assumptions and Constraints

Assumptions

- The minimum number of nodes to be simulated is three.
- The project will only be used for academic purposes and does not require security constraints.
- The visualizer will be used on a screen that is of size at least 24”.

Constraints

- With the pandemic going on, we will have to adapt to work in a virtual environment.
- There is no actual sniffer, so it will be hard to do an integration test of the visualizer with the sniffer.
- Per our clients’ request, we will be using open source libraries.

2 Design

2.1 Related Work and Literature

Vishal Deep and Dr. Duwe's research group has shared a paper they had published on the topic of maintaining a shared sense of time in a network of batterless nodes [4]. It is cited in section 5.2.

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 were able to reference while working on our design project. However, our simulator was developed with a larger focus on node communication rather than modelling individual nodes, so we did not rely very heavily on the existing simulator.

2.2 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 the 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.

2.3 Design

The system will consist of two main components: a simulator and a visualizer. The visualizer, a web application, will consist of two applications: backend and frontend.

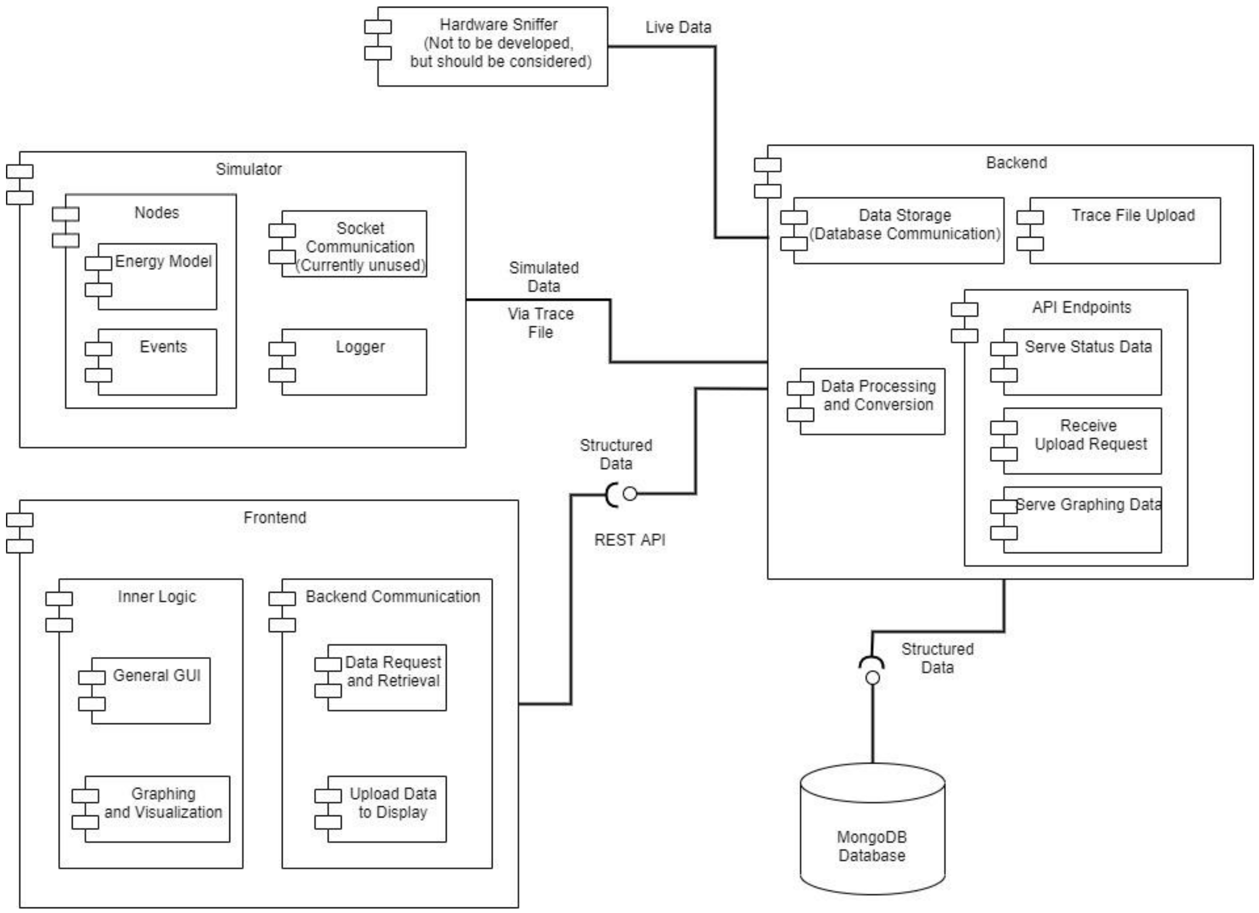


Figure 2.1. System block diagram

2.3.1 Simulator Design

The simulator will run as a Python application 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 multiple nodes
    - Maintain the true time within the simulation
    - Runs the main loop for the simulation
    - Outputs a trace file of events that occurred in the simulation

- Node (multiple)
    - Maintains its own clock
    - Uses an energy model to track when it turns on or off
    - Tracks its own state (off, boot, or on)

- Energy Model
    - Determines how long a node will remain in each state (off, boot, on)
    - Models the time for a sensor node to harvest and consume energy
    - Has variation to emulate real sensor networks
- Events
    - Store the details of events that occur in the sensor network (transitions between states, communication with other nodes)
- Logger
    - Outputs the details stored in an event
    - Configurable to log details in different ways

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 performs 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 details of these classes are shown in Figure 2.2.

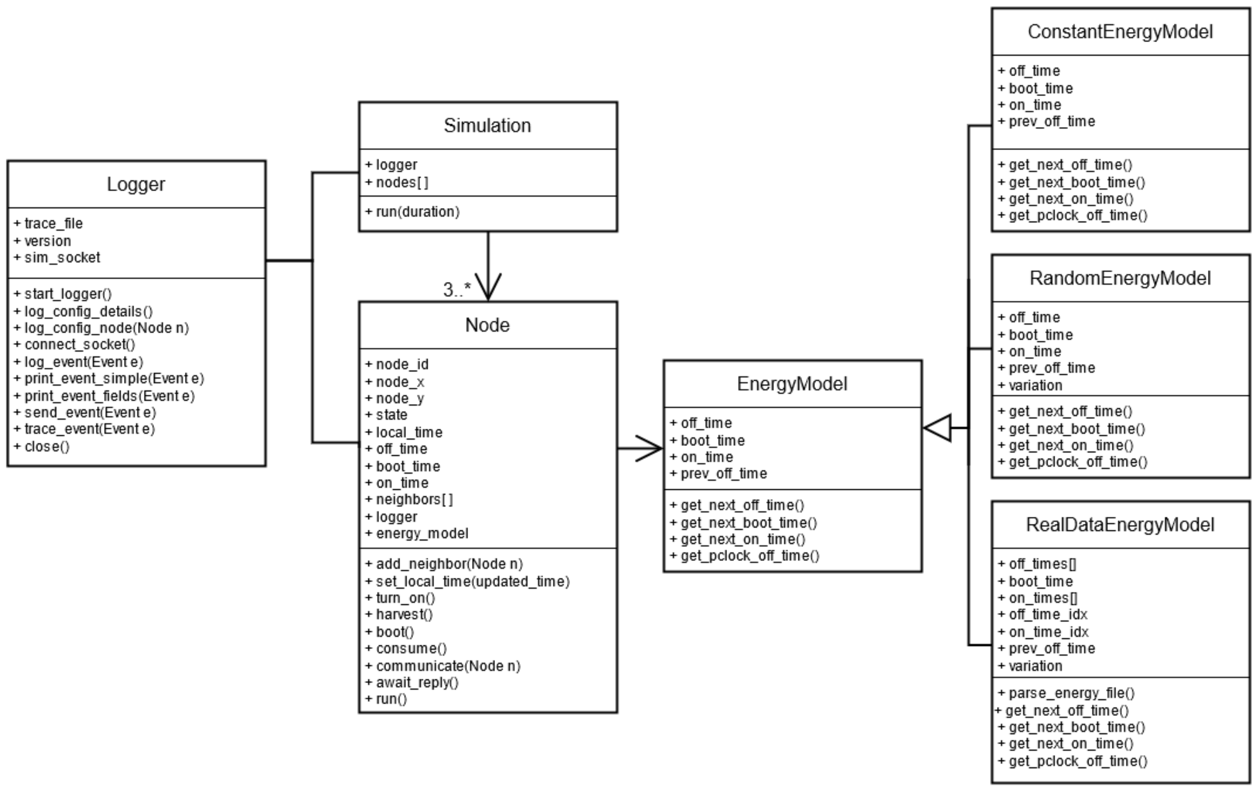


Figure 2.2. Simulator class diagram

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 III: Other Considerations - Lifecycle of a Node.

While cycling through the states of a node, the simulator will generate several events to record what is happening in the system. The events record transitions between states and communication between nodes. The events are shown below in Figure 2.3.

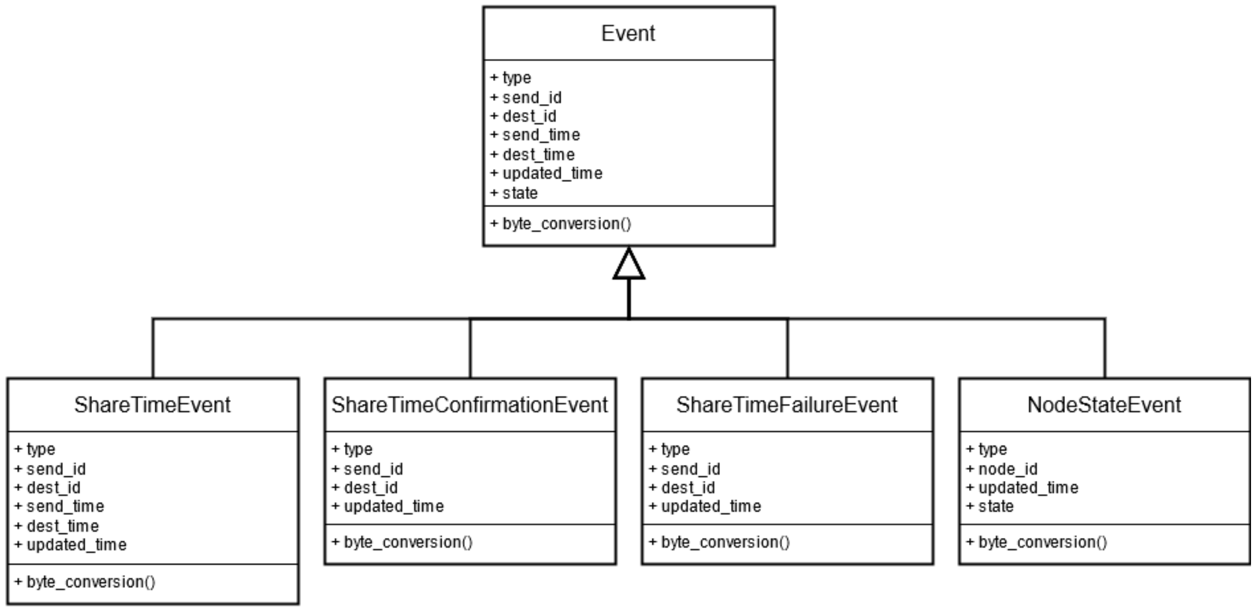


Figure 2.3. Simulator events class diagram

These events are logged with the Logger class. There are multiple formats for logging events, including a plain-text format, a list of events with fields, a message to the backend via sockets, and logging to a trace file. Logging to a trace file will be the most common format for recording events because the visualizer uses trace files to view the events of the network. Table describes the trace file format.

Configuration Details Simulation characteristics	**Version (1 byte)**	**Resolution (1 byte)**	**Duration (8 bytes)**	**Number of nodes (1 byte)**
	Trace file version number	Smallest time division in the simulation	Simulation duration in terms of time steps	Number of nodes in the simulated network
Node List List of nodes and their characteristics	**Node A ID (1 byte)**	**Node A x-position (1 byte)**	**Node A y-position (1 byte)**	**Number of neighbors (1 byte)**
	ID for labelling the node	x-position to place the node in the visualizer	y-position to place the node in the visualizer	Number of neighboring nodes
	Neighbor 1 ID (1 byte)	**Neighbor 2 ID (1 byte)**	**...**	
	ID of neighbor node	ID of neighbor node	More neighbors	
	...			
	More node details			
Event List List of events that occur during the simulation	**...**			
	See Table 2.2			

Table 2.1. Trace File Format

The event list section of the trace file lists the different events that occurred during the sensor network simulation. The event fields depend on the type of events as shown in Table 2.2.

Event	**Share Time Event (35 bytes)**	**Share Time Confirmation Event (19 bytes)**	**Share Time Failure Event (19 bytes)**	**Node State Event (35 bytes)**
Description	Indicates attempted communication between nodes	Confirms successful communication	Indicates failed communication	Indicates a node state transition (i.e. off-to-boot, boot-to-on, on-to-off)
Fields	Real time (8 bytes) Event type (1 byte) Sender node ID (1 byte) Destination node ID (1 byte) Sender time (8 bytes) Destination time (8 bytes) Updated time (8 bytes)	Real time (8 bytes) Event type (1 byte) Sender node ID (1 byte) Destination node ID (1 byte) Updated time (8 bytes)	Real time (8 bytes) Event type (1 byte) Sender node ID (1 byte) Destination node ID (1 byte) Updated time (8 bytes)	Real time (8 bytes) Event type (1 byte) Node ID (1 byte) Updated time (8 bytes) State (1 byte)

Table 2.2. Trace File Events Format

2.3.2 Backend Design

The backend runs server side and is accessed by our frontend. The backend receives a trace file from the frontend. The frontend can also choose to send more than one trace file to the backend to process, which gives them the ability to run multiple simulations at a time. The backend immediately pumps out the configuration for that simulation for the frontend to display immediately while still processing and storing the data events to the database. Once the simulation is ready to go and processed, the user in the frontend can make an HTTP request to fetch all the important time events where a communication event occurred. There, a user can click on any node, and the backend will spit out important data and statistics using an HTTP request.

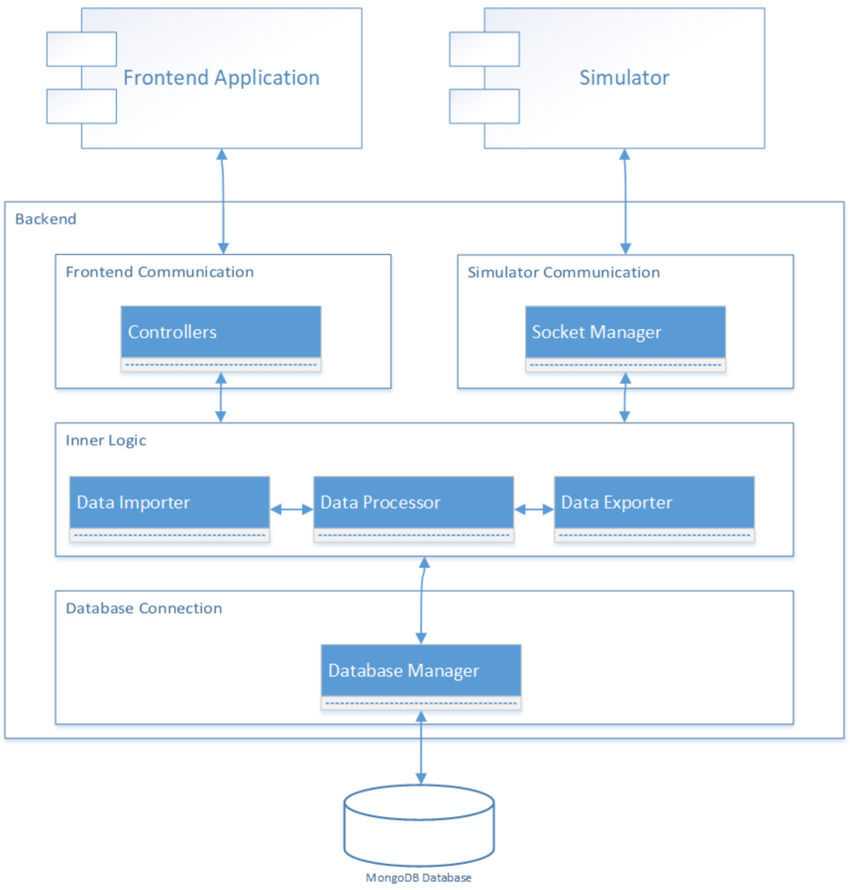


Figure 2.4. Backend block diagram

2.3.3 Frontend Design

The frontend is a web-based application, which allows it to be run on any device with a web browser. It gets all the necessary information it needs to run the visualizer from the RESTful APIs provided by the backend. The frontend sends HTTP requests to the backend requesting data to update the state of the network being displayed and the various graphs. The application consists of many components that handle their own state. The Presenter/Adapter component (see Figure 2.4 below) is responsible for maintaining the state of the visualization as well as processing the data to be displayed. The Presenter/Adapter is one logical component because React automatically rerenders what is displayed when the state of the component changes. The Request Sender component is responsible for sending HTTP requests to the backend application and passes on the data to the presenter through the EventBus (an implementation of the publisher-subscriber pattern).

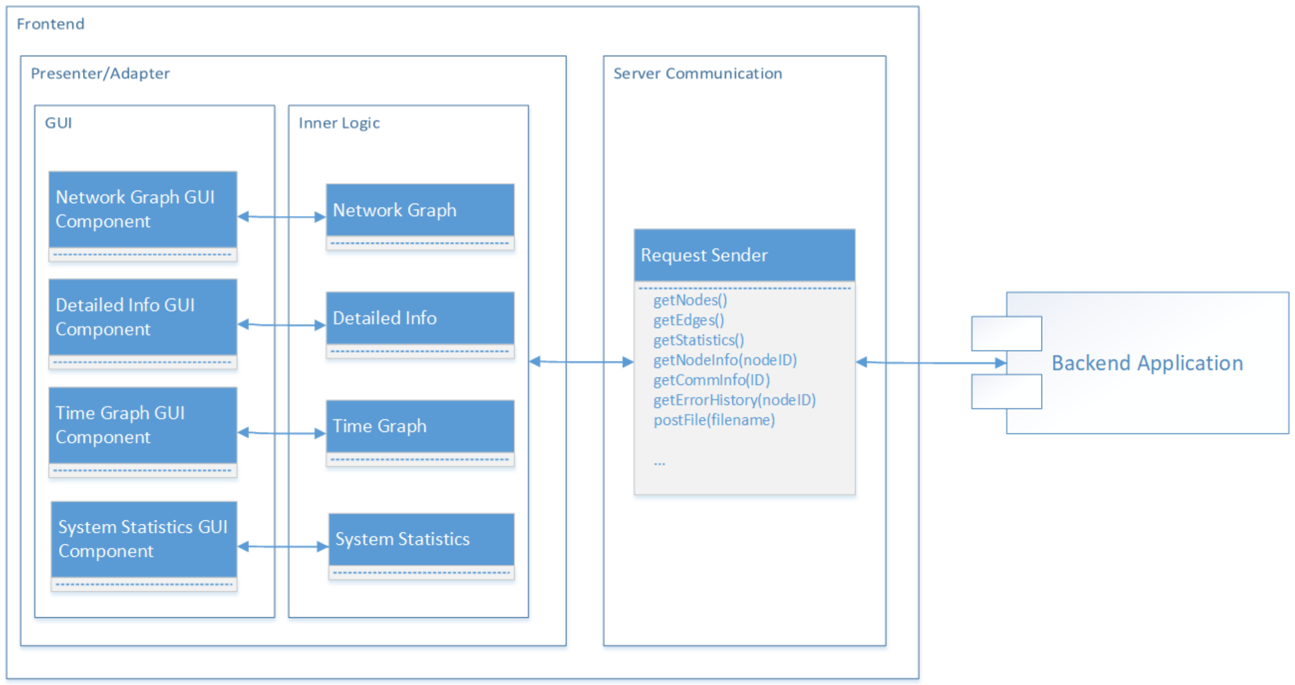


Figure 2.5. Frontend block diagram

The Graphical User Interface consists of two main pages: the Simulation Management Page, and the Main Visualization Page. On the Simulation Management Page, the user can see the list of all processed simulations that can be visualized; visualize previously saved simulations; upload new simulations; and delete simulations.

On the Main Visualization Page, there is a global clock displayed for the real time of the system. The user can move through time to view the state of the network at a particular time. The network of batteryless sensors is visualized as a graph with nodes representing the sensors, and edges representing communications between sensors. The communications are displayed as directed edges, and the color of the edge represents the status of communication (green for successful communication and red for failed communication). The ability of two nodes to communicate is displayed as a bidirectional gray edge between two nodes. The user can see the status of each node (ON, OFF, and BOOT), nodes' local times, nodes' errors, and other information. The user can also examine each node's history of communication up to a point in time (displayed in a separate graph as a rooted tree). Statistics are displayed to the user in an appealing way. The final graph displayed to the user is a line graph depicting the mean and median error of all nodes in the network. When selecting a specific node, this graph updates to show that node's local error as well as the other two lines.

2.4 Technology Considerations

2.4.1 Simulator

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

The simulator is built in Python since it is a simple and powerful programming language. A major strength comes with the availability of Python libraries, including the standard library and the socket module that enables 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 [5]. 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.

2.4.2 Backend

- ExpressJS (NodeJS)
- MongoDB database

These languages allow us a simple yet robust methodology for proving REST APIs to our frontend. Utilizing the Express framework allows us to bring up the backend quickly, and without too much learning curve being written in NodeJS [6]. 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 are prevalent in the live portion of the application [7]. MongoDB specifically allows us to keep a record of all events in one collection event though they likely do not all have the same fields. Although our knowledge is stronger in other databases, MongoDB's benefits outweighed our strengths [8].

2.4.3 Frontend

- HTML & CSS
- Javascript - ReactJS [9]
- Selenium [10]
- Jest [11]

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 are 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. When it comes to testing, we decided to use Selenium for GUI testing and Jest for unit testing.

2.5 Design Revisions

2.5.1 Simulator Revisions

Previously, the simulator was designed to include a specialized node that worked as the sniffer. The sniffer would model a physical hardware device that monitors a network to record communication between nodes. The simulator no longer relies on this node to handle communication between nodes. because the SimPy library can be used to track global time and the system has full knowledge of all nodes.

The initial design also planned to manually handle discrete-event simulation through the use of a priority queue that stored network events. However, the event priority queue is no longer required because the SimPy library provides that functionality automatically.

Additionally, the simulator design has been changed so that the nodes in the sensor network do not have to model their own energy harvesting and discharging rates. This was removed as this is beyond the scope of the project. Instead, the nodes get their on- and off-times from the energy models that can be configured as needed. With this change, the EnergyModel class was added for abstraction. The Event and Logger classes were also added to the design to abstract more details away from the node and simulation modules.

Similar to the original design, the simulator still runs from the command line; however, there are no command-line arguments to configure the system. In place of command-line arguments, the simulator uses a configuration file to set up the system. The previous design also planned to rely more on a socket connection between the simulator and backend to facilitate the flow of data to the visualizer. While this still has been implemented, the primary method for feeding data to the frontend is via a simulator-generated trace file that can be loaded by the frontend.

2.5.2 Backend Revisions

Our design for the backend has changed slightly since our initial design. The technologies and general setups have worked well for our use cases. However, there have been some other changes that have been made.

One main point that has changed is the priority of the socket connection to the simulator. Our client decided this real-time connection was not a priority, as he could instead just upload a trace file produced by the simulator within the frontend. Thus, our code is written in such a way that adjustments to adding a socket would be straightforward, but we do not actually implement it.

A second larger design update is based on the oversimplification of data processing. The backend module now contains a large segment of code that processes the events produced by the simulator to produce a summary of what is actually occurring in the system in a new data format. This processing was described, but not in as much detail originally, partially because we didn't fully know the formats going to or from the Frontend and Simulator.

Finally, the backend has had to add endpoints to provide the data structures behind the error graph and error tree. These again have involved more logical consumption of the history of the simulation.

2.5.3 Frontend Revisions

The high-level design for the frontend application has changed slightly. We have broken up the Presenter/Adapter into two logical components: Presenter and Adapter. This happened because we need to process the response from the backend to make it compatible with the format required by the libraries we are using.

Our detailed design had changed a few times throughout the development process due to the changes in our clients' requirements. Initially, the Visualizer was supposed to display all the information on one main page. Now, we have added a separate page for managing all the simulations, per our clients' requests. Other changes to our detailed design were regarding the libraries we were using. Initially, we wanted to use GoJS, which is proprietary, and our client requested that we use only open-source libraries.

3 Implementation

3.1 Simulator

We have developed a simulator that can model a simulated sensor network. Using SimPy, we have been able to define the behavior of each of the nodes as they cycle between off and on states and as they communicate with other nodes. The simulator 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. The components of the simulator are described in more detail in the following sections.

3.1.1 Simulation

The Simulation class is implemented in simulation.py. It is the main section of the simulator that drives the simulation with its run() function. The run() function calls the run() function for the discrete-event simulation library, which starts the simulation. Before running the simulation, it is set up with a reference to the discrete-event simulation environment, a Logger object for logging the simulation results, and a list of nodes that describe the network.

3.1.2 Nodes

The Node class is defined in node.py and contains logic for driving the lifecycle of a node. Each node has a set of properties for defining their behavior, namely off-, boot-, and on-times; a list of neighboring nodes; and an energy model. The nodes also have an ID and a position that are used to visualize each node. The nodes are instantiated in simulation.py with details derived from the configuration file.

The lifecycle of a node is driven in the run() function within the class. It cycles between the off, boot, and on states. These states are defined in the NodeState enumeration. In the run() function, the node harvest() function is called to simulate the off period where a node is turned off while harvesting energy. This runs for the duration of the off time. Next, the boot() function executes when a node starts to turn on. In boot(), the node updates its local time based on the reading from the persistent clock. Then, turn_on() is called to transition a node into the on state. After turning on, a node will attempt to communicate with all of its neighbors. Finally, the consume() function is run to represent the remaining on period until a node runs out of energy and returns to the off state. This cycle repeats continuously until the end of the simulation.

3.1.3 Energy Models

Each node has an energy model which controls how long it will remain in each state (off, boot, and on). The energy model also controls the variation in time for each state. To get the times for each of these states, there are corresponding functions: get_next_off_time(), get_next_boot_time(), and get_next_on_time(). To produce the local error that occurs when a node reads its persistent clock, the get_pclock_off_time() is used by a node to estimate how long it was off. The node uses this to update its local sense of time. This value will vary depending on the type of energy model. There are multiple energy model types, including the constant energy model and the random energy model, which are defined in the ConstantEnergyModel and RandomEnergyModel classes in energy_model.py.

3.1.4 Configuration

The config.py file includes several shared definitions that are used throughout the project, including timing values, logging configurations, and sensor network configurations. This file also includes the format of the trace file previously described in Section 2.3.1.

3.1.5 Events

The events that may occur during a simulation are defined in the event.py file. Each event type has a separate class with corresponding fields and a byte_conversion() function. There are four event classes, one for each event type: ShareTimeEvent, ShareTimeConfirmationEvent, ShareTimeFailureEvent, and NodeStateEvent. These event types are also enumerated in the EventType class. The NodeStateEvent occurs every time a node transitions between states (i.e. off-to-boot, boot-to-on, and on-to-off). It has fields to record the ID of the node, its updated time, and its state.

The ShareTimeEvent occurs when nodes attempt to communicate with their neighbors. It has fields to store the IDs of the sending and destination nodes, their local times, and the updated time. For every ShareTimeEvent, there is a corresponding response that comes as a ShareTimeConfirmationEvent or a ShareTimeFailureEvent. If the destination node is on, the response is successful, meaning a ShareTimeConfirmationEvent occurs. Otherwise, a ShareTimeFailureEvent occurs. Both of these response events have the same fields: the IDs of the sending and destination nodes and the updated time. If communication is successful, the destination node updates its local time to the updated time from the event field.

3.1.6 Logger

To record events that occur during a simulation, we developed the Logger class, which resides in logger.py. There are four supported logging formats that the logger can handle: a simple, easy-to-read format, a plain-text log of event fields, a data transfer over a socket connection, and a byte-by-byte log of event fields to a trace file. Each logging format has a corresponding function that handles it: print_event_simple(), print_event_fields, send_event(), and trace_event(). The logging methods are configured in config.py. In addition, there are helper functions to set up logging, handle the logging of configuration details, and end logging. See Appendix III: Other Considerations - Plain-Text Logging Example for an insight into the logger’s plain-text output.

3.2 Backend

The backend server has been implemented using the express.js framework and MongoDB. Once the frontend has received a trace file the frontend can upload that file for the backend to process and store. With MongoDB, the frontend can store multiple trace files and can look at different simulations at a time. The backend also has endpoints that provide meaningful data about the simulation like displaying the mean of nodes or figuring out where the error originally came from. Here are the multiple endpoints that the backend provides for the frontend to use.

Endpoint Call	**Type**	**Parameters**	**Description**	**Output**
/file	POST	File, name	Uploads the trace file to the backend for the backend to process and store into the database	The generated simId, and the name you selected.
/getAllSims	GET	No parameters	Serves all the simulations that is currently stored on the database	An array of objects showing simId, name, processed status
/getConfig	GET	simId	Will serve configuration data of the simulation.	An object showing duration, noOfNodes and an array of Node objects and their properties.
/deleteSim	GET	simId	Will delete the simulation that the user chooses	A boolean to see if the simulation was deleted
/getTimeObject	GET	simId, queryType, realTime, simId	Queries the database for the selected realTime	Returns the realtime and an array of node objects and their time values
/meanAndMedErrors	GET	simId, realTime, numPoints	Grabs the mean and median of the group of nodes at each realTime up to the numPoints	Returns an array of real time, meanError,median Error
/nodeErrors	GET	simId, realTime, numPoints	Get the error for the selected node at relevant realtime points of numPoints/2 above and below	Returns an array of realTime and errors for the nodeId

/errorTree	GET	simId, realTime, nodeId, numGraphNodes	Creates a tree of error for the selected node at a realTime and shows the user where the error could have originated from depending on numGraphNodes	Returns the error tree with having the selected nodeId as a root
/getSimStatus	GET	simId	This endpoint will grab the state of the simulation and tell the user whether it's processed or not.	A boolean variable showing the status of the simulation.

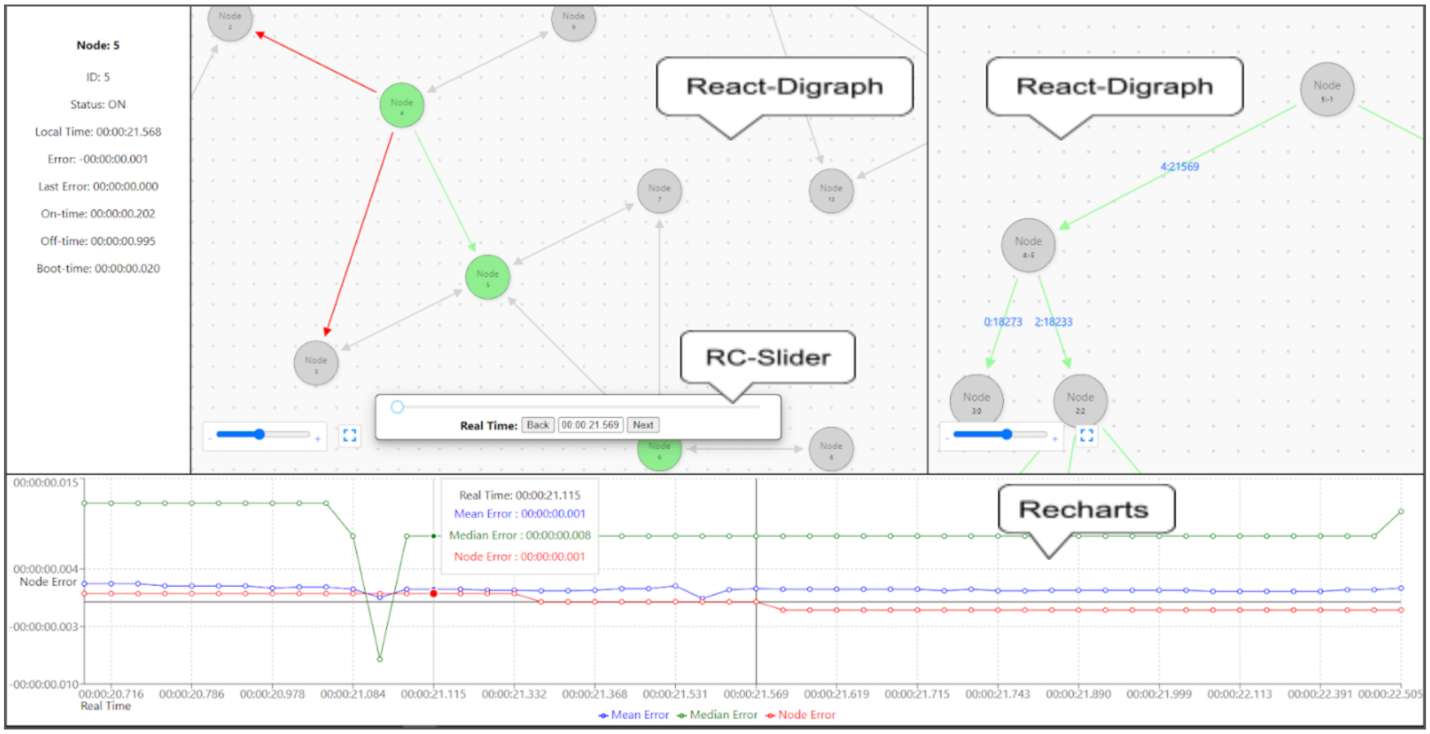
Table 3.1. API Endpoints

3.3 Frontend

The frontend application has been implemented using the ReactJS framework and open-source JS libraries. The application consists of two main pages: Trace File Input Page (previously mentioned "Simulation Management Page" in section 2.3.3), and Main Visualization Page (*figure* 3.1). These pages are broken up into multiple components that encapsulate and manage their own information, which makes the application modular and easily maintainable.
The Main Visualization page can only be accessed after the user has uploaded at least one simulation. There is one class responsible for sending all HTTP requests to the backend application's API endpoints, and one class that processes all the responses from the backend and transforms them to the format required by the libraries used. The application implements the subscriber-publisher pattern to update the GUI upon receiving a response from the backend. Each component is subscribed to different event types, and once the event is fired, a callback function is used to update the UI.

3.3.1 Resources

For displaying the network graph and error propagation tree graph, our application uses an open-source library called React-Digraph [12]. This library provides a convenient way for us to draw directed graphs using our custom styles. It requires that the nodes' coordinates are provided within the Node object, so the Buchheim algorithm for assigning coordinates to nodes in rooted trees in linear time was implemented [13]. This algorithm assigns the coordinates in such a way that most users will find the tree appealing. The Recharts library [14] is used for graphing the network error statistics, and the RC-Slider library is used for the slider control panel [15]. Utilized libraries are labeled in Figure 3.1 below.



Node: 5

ID: 5

Status: ON

Local Time: 00:00:21.568

Error: -00:00:00.001

Last Error: 00:00:00.000

On-time: 00:00:00.202

Off-time: 00:00:00.995

Boot-time: 00:00:00.020

Real Time: 00:00:21.115
Mean Error : 00:00:00.001
Median Error : 00:00:00.008
Node Error : 00:00:00.001

Figure 3.1. Main visualization page

4 Testing

4.1 Unit Testing

4.1.1 Simulator Testing

For our test plan, we decided to individually test the simulation and node modules to ensure each unit works on its own. We also executed 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
    - 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
- 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 developed the backend (which receives data through the socket connection) concurrently, we used mocking to simulate the reception of data so the simulator can be tested without a finalized backend.

4.1.2 Backend Testing

We have two main segments of the backend to test, first the consumption of trace files and the endpoints themselves. To test the consumption of trace files the simulator was equipped with the ability to show a "plain text" version of the simulation. It describes in words. "Node 2 boots at time 1300" or "Node 3 failed communication with Node 7 at time 1900". This way many trace files have been able to be used as unit tests. Our collection of system statuses can be tested by cross checking to the plain text version. Although this is not an automated unit test, it was used to ensure the consumption of the binary trace files was completely working. The endpoints had a more straightforward approach to test. Postman was utilized to mock calls that the frontend may make, then the output could be checked for formatting and content according to what we could manually see within the database. This was also our methodology of providing mock data to the frontend to begin to use, meaning we can copy the exact output and allow them to use it, before actually making the calls.

4.1.3 Frontend Testing

Many visualizer components have been tested in isolation from the other parts of the system. The underlying classes to all components, the RestAPI requests/responses to the backend, and the utility components have been unit tested with the help of the Jest testing framework. To test backend connectivity without the backend working yet, the frontend team developed their own primitive version of the backend application that provided identical endpoints to those that the backend application was developing. This application was then used by the frontend team to make sure that all the RestAPI requests worked in integration with the backend application. This approach made the actual integration almost conflict-free. The GUI testing was also performed by the frontend team. For more information on the GUI testing, see the next section.

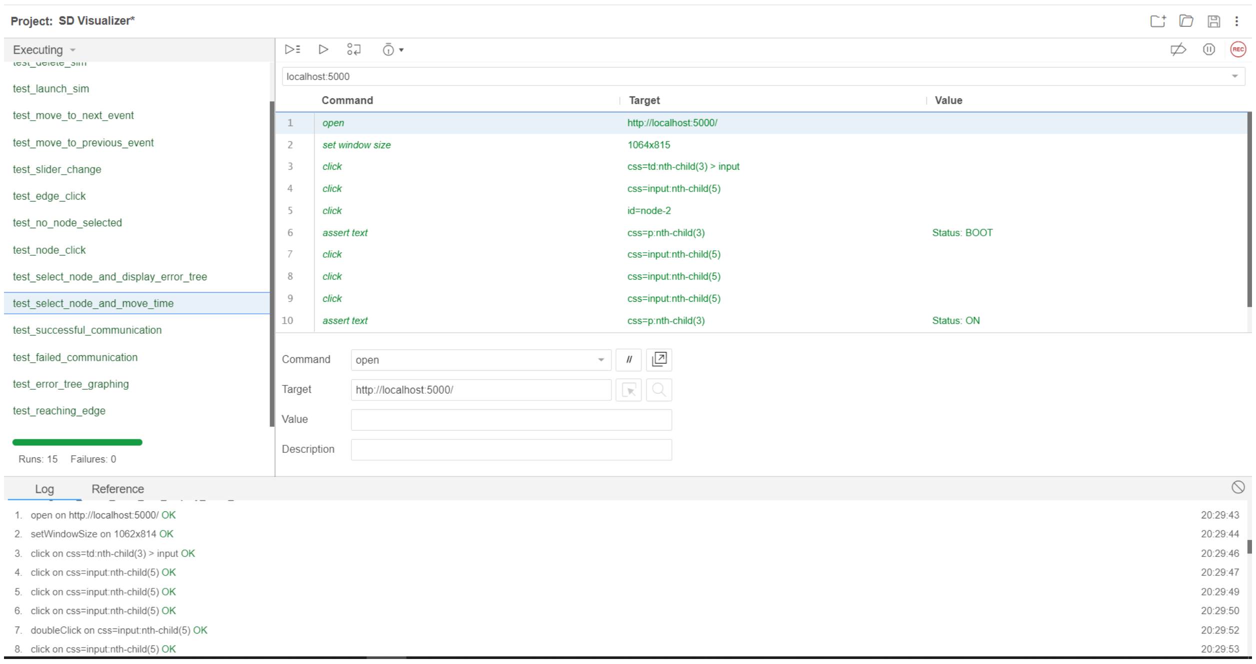
4.2 Interface Testing

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

- Simulator to visualizer
- Backend to Frontend

The initial testing of these interfaces was done via mocking on each segment. From there, the actual interfaces were tested via several common use cases and some edges cases (large files or uncommon requests). It was important to test each interface separately before advancing to further full system integration testing.

The Graphical User Interface (GUI) is the main point of contact between the user and our system. We have done extensive automated GUI testing using Selenium. We created a test suite that tests all main graphical components that the user will be using. Attached is an example of one of the tests in our testsuite, as well as the successful execution of the testsuite.


Project: SD Visualizer*

Executing

test_launch_sim
test_move_to_next_event
test_move_to_previous_event
test_slider_change
test_edge_click
test_no_node_selected
test_node_click
test_select_node_and_display_error_tree
test_select_node_and_move_time
test_successful_communication
test_failed_communication
test_error_tree_graphing
test_reaching_edge

Runs: 15 Failures: 0

localhost:5000

	Command	Target	Value
1	*open*	http://localhost:5000/	
2	*set window size*	1064x815	
3	*click*	css=td:nth-child(3) > input	
4	*click*	css=input:nth-child(5)	
5	*click*	id=node-2	
6	*assert text*	css=p:nth-child(3)	Status: BOOT
7	*click*	css=input:nth-child(5)	
8	*click*	css=input:nth-child(5)	
9	*click*	css=input:nth-child(5)	
10	*assert text*	css=p:nth-child(3)	Status: ON

Command	open
Target	http://localhost:5000/
Value	
Description	

Log / Reference

```
1. open on http://localhost:5000/ OK                        20:29:43
2. setWindowSize on 1062x814 OK                             20:29:44
3. click on css=td:nth-child(3) > input OK                  20:29:46
4. click on css=input:nth-child(5) OK                       20:29:47
5. click on css=input:nth-child(5) OK                       20:29:49
6. click on css=input:nth-child(5) OK                       20:29:50
7. doubleClick on css=input:nth-child(5) OK                 20:29:52
8. click on css=input:nth-child(5) OK                       20:29:53
```

Figure 4.1. Visualizer GUI test suite

Extensive manual testing of the GUI was also performed by the frontend development team as well as other developers. Although the complexity of the GUI interactions will not allow us to have a complete testsuite, we are sure that the GUI meets the client's requirements.

4.3 Integration Testing

Integration testing was an important element of this three-part system. This testing ensured that all three modules are working together to fulfil the use cases. The system must complete them up to our standards before we bring similar use cases to our client. The following use cases were tested across the system:

- Importing a trace file to the frontend and visualizing the simulation it describes
    - This confirms the input functionality of the frontend.
- 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 display 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 the 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 involved our client in the development/testing process for their acceptance. We conducted incremental acceptance testing after each feature had been implemented. It was important that we did not wait until the end of the project to receive acceptance; thus, we met with our client biweekly, or sometimes weekly, to get feedback on our design and implementation. These demos were an important method of receiving acceptance, so we could present our ideas and explain them instead of having our client blindly try to navigate the application.

4.5 Results

As a result of the testing stage of our software development lifecycle, we 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 includes testing artifacts (e.g. test results).

4.5.1 Simulator Tests

The simulator was tested on a variety of sensor network configurations, ranging from simple 3-node networks to complex 15-node networks. It was also tested using simulations of varying durations, running from 10 minutes to 24 hours with a time step of 1 millisecond. This test range allowed us to verify that our simulator could handle different scenarios, and it allowed us to check performance against different configurations. The following tables shows the results from these tests:

Duration (simulated time)	10 minutes	2 hours	24 hours
Actual run time	1.82 sec	21.26 sec	4 min, 19.23 sec
Seconds of run time per simulated second	0.00303	0.00295	0.00300
Seconds of run time per simulated second per node	0.00101	0.00098	0.00100

Table 4.1. 3-Node Simulation Performance

Duration (simulated time)	10 minutes	2 hours	24 hours
Actual run time	5.86 sec	1 min, 9.94 sec	14 min, 9.05 sec
Seconds of run time per simulated second	0.00977	0.00971	0.00983
Seconds of run time per simulated second per node	0.00122	0.00121	0.00123

Table 4.2. 8-Node Simulation Performance

Duration (simulated time)	10 minutes	2 hours	24 hours
Actual run time	11.03 sec	2 min, 11.65 sec	26 min, 50.46 sec
Seconds of run time per simulated second	0.01838	0.01828	0.01864
Seconds of run time per simulated second per node	0.00123	0.00122	0.00124

Table 4.3. 15-Node Simulation Performance

These results show that the run time for a simulation is roughly proportional to the duration of the simulation and the number nodes in the simulation. More specifically, the simulation run time is approximately 0.0010-0.0012 seconds for each second of simulated time for each node, meaning the run time can be estimated by multiplying this value (about 0.0011) by the number of nodes and the simulated time. Using the 8-node, 2-hour simulation as an example, the simulated time would be estimated to be about 0.0011 sec/sim-sec per node x 8 nodes x (2 x 60 x 60 sim-sec) to get an approximation of 63.36 seconds, which is very close to the observed run time of 69.94 seconds.

While setting up the communication interface between the simulator and the backend, we created a mock_backend.py file that imitated the functionality of the backend to test the connection. First, the mock backend established the connection with the simulator, and then it received the data that the simulator was sending. It printed out this data to confirm that the received format matched the data that the simulator was sending.

4.5.2 Backend Tests

The backend has been tested successfully. The consumption of trace files is working successfully, it exactly matches the "plain text" output and serves that consumable data in such a way that the frontend can display it. Additionally, each endpoint, including those that generate based on an endpoint call work successfully. The integration from the simulator and to the frontend has been successful because of these testing successes.

4.5.3 Frontend Tests

The frontend testing has been completed with both unit tests and manual feature tests after each new feature had been implemented. Unit tests have been created using Jest, and Selenium is being used to test the GUI of the frontend. The system has minimal bugs related to the requirements of the final product. Manual testing was also done by the implementer of the features and by the other frontend developer who did not work on the feature. We have tested the frontend and backend applications together by uploading trace files of various sizes. The smallest file was for a three node ten minute simulation and the largest was a fifteen node 24 hour simulation.

4.5.4 Integration Tests

We conducted integration testing in two stages. First, we performed smoke testing to see if there were any severe issues with the system integration. We have managed to get all three systems to communicate with one another. We then manually tested each integration point discussed in section 4.3 above. We performed manual integration testing before committing the code to the remote repository. After the last commit, we performed final integration testing with the entire team present. As a result, we have a fully functional system that is ready for acceptance testing.

5 Closing Material

5.1 Conclusion

The project is in a working and releasable state and is being delivered to the client. It has been tested up to the initial baseline specifications and beyond. We have iterated over the design throughout our development to settle on the final design detailed above. Our team is satisfied with our progress and excited to see how the project will continue to be used in this research space.

5.2 References

[1] R. Fielding and J. Reschke. "Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content", RFC 7231, June 2014.

[2] J. Postel. "Transmission Control Protocol", RFC 793, September 1981.

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

[4] 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].

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

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

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

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

[9] "React - A JavaScript Library for Building User Interfaces", Vers 17.0.2, Facebook Open Source, Facebook, 2021, Accessed: 04/25/2021 [Online]. Available: https://reactjs.org/

[10] "The Selenium Browser Automation Project", Selenium HQ Browser Automation, 2021, Accessed: 04/25/2021 [Online]. Available: https://www.selenium.dev/documentation/en/

[11] "Jest - Delightful JavaScript Testing", Facebook Open Source, Facebook, 2021, Accessed: 04/25/2021 [Online]. Available: https://jestjs.io/

[12] “A Library for Creating Directed Graph Editors”, Uber Open Source, Uber, 2021, Accessed: 04/25/2021 [Online]. Available: https://github.com/uber/react-digraph

[13] C. Buchheim, M. Junger and S. Leipert “Improving Walker’s Algorithm to Run in Linear Time”, Universitat zu Koln, Koln, Germany. [Online]. Available: http://dirk.jivas.de/papers/buchheim02improving.pdf

[14] “Reacharts - A Composable Charting Library Built on React Components”, Recharts Group, Vers. 2.0.9. Accessed: 04/25/2021 [Online]. Available: https://recharts.org/en-US/

[15] “React Slider”, Schrodinger, Inc., 2021, Accessed: 04/25/2021 [Online]. Available: https://github.com/schrodinger/rc-slider

5.3 Appendix

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

5.3.1 Appendix I: Operation Manual

Execute these instructions sequentially to set up and operate the simulator and visualizer, starting with project setup, then simulator, followed by backend, and finally frontend. The simulator will generate a trace file that will be needed when running the frontend.

Project Setup

Prerequisites:

- Git is installed

Get the project files:

- Navigate to a location to store the project
- Clone the git repository with "git clone https://git.ece.iastate.edu/duwe/sst-sim.git"

Simulator

Prerequisites:

- Python is installed
- The sst-sim repository exists on the local machine

Configuration:

In config.py, configure the following:

- Specify the logging configuration
- Specify the time resolution
- Specify the duration
- Specify the number of nodes in the simulation
- Define all nodes in the network with the following characteristics
  - Node ID
  - (x,y)-position for placement in the visualizer
  - Off-, boot-, and on-times
  - List of neighbor nodes
  - Energy model
- Populate the network list with all of the nodes
- Specify the reply delay

Execution:

- Run the simulation from the simulator folder with "python simulation.py"
- If the logging configuration outputs a trace file, it will be located in the trace_files subdirectory. It will be needed when running the frontend.

Backend

Installation:

- Install the most recent version of MongoDB Community Edition (https://www.mongodb.com/try/download/community). Keep note of the installation directory, you will need it later.
- Additionally, ensure Node.js is installed on your system (https://nodejs.org/en/). Then run npm install inside of the /backend folder, several packages will likely be installed.

Execution:

- Copy the "start-backend-TEMPLATE.sh" included in the "/backend" folder and create a copy of it still within the "/backend" directory. Rename the copy to "start-backend.sh" as this is ignored by git so we only share the template. Edit "start-backend.sh" to make the two paths reflect your system, explanations are in the comments of that script.
- Run ./start-backend.sh in a capable command line. If on Windows this may require Cygwin, Git Bash, or similar.
    - This script starts both the mongo server process and the ExpressJS project.
    - The Mongo server output is put into /backend/database.log. The ExpressJS Server output will continue to be shown in the command line.
    - Ctrl-C will terminate the Mongo server and the Express project

Frontend

Installation:

- Prerequisites:
    - *node.js* and *npm* (Node.js)
      Note: *npm* is installed with *node.js*.
- To install the program:
    1. Download the source code and navigate to the "frontend/visualizer" folder in the command line terminal.
    2. Run "npm install".

Compilation:

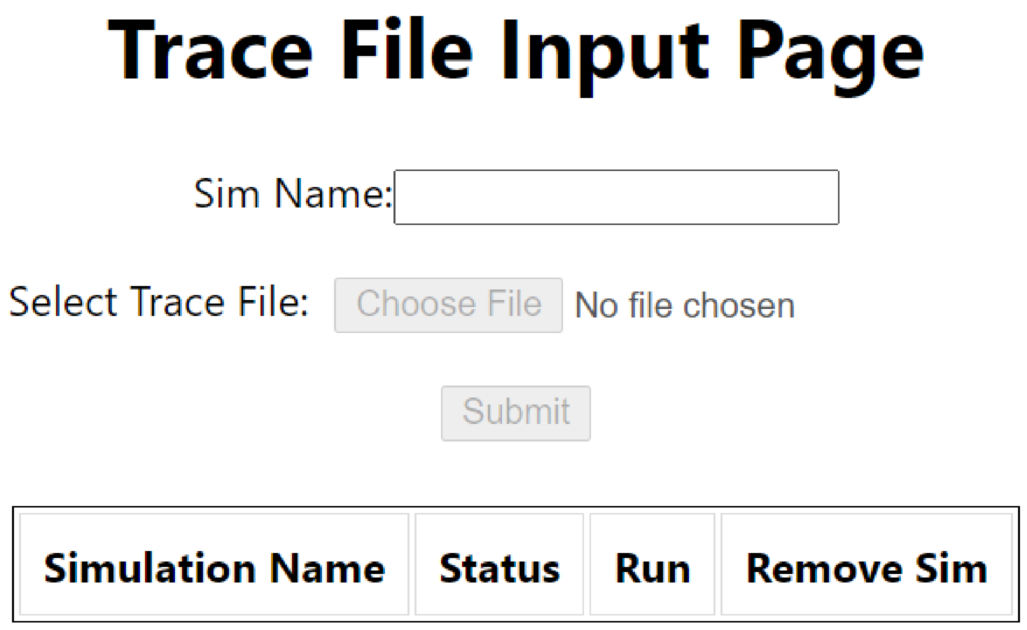
- To compile the program, run "npm run build".

Execution:

- Prerequisites: *serve* ("npm install -g serve" or "npm install serve"); *compiled frontend application; running backend application; at least one .trace file generated by the simulator.*
- To run the visualizer, execute "serve -s build".

Using the program:

- Navigate to http://localhost:5000/ in a web browser of your choice, and you will see the simulation management page (*figure* 5.1).


Trace File Input Page

Sim Name:

Select Trace File: Choose File No file chosen

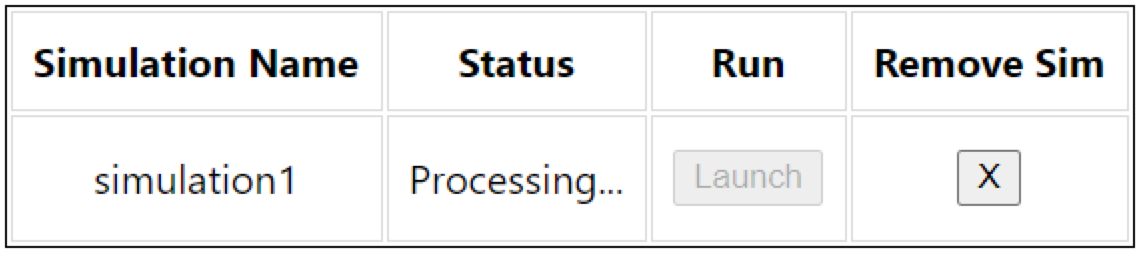
Submit

Simulation Name	Status	Run	Remove Sim

Figure 5.1. Simulation management page

1. *Import a simulation.*
    a. Enter simulation name.
    b. Select a trace file containing the simulation.
    c. Click "submit" to move to the main page for the inputted simulation

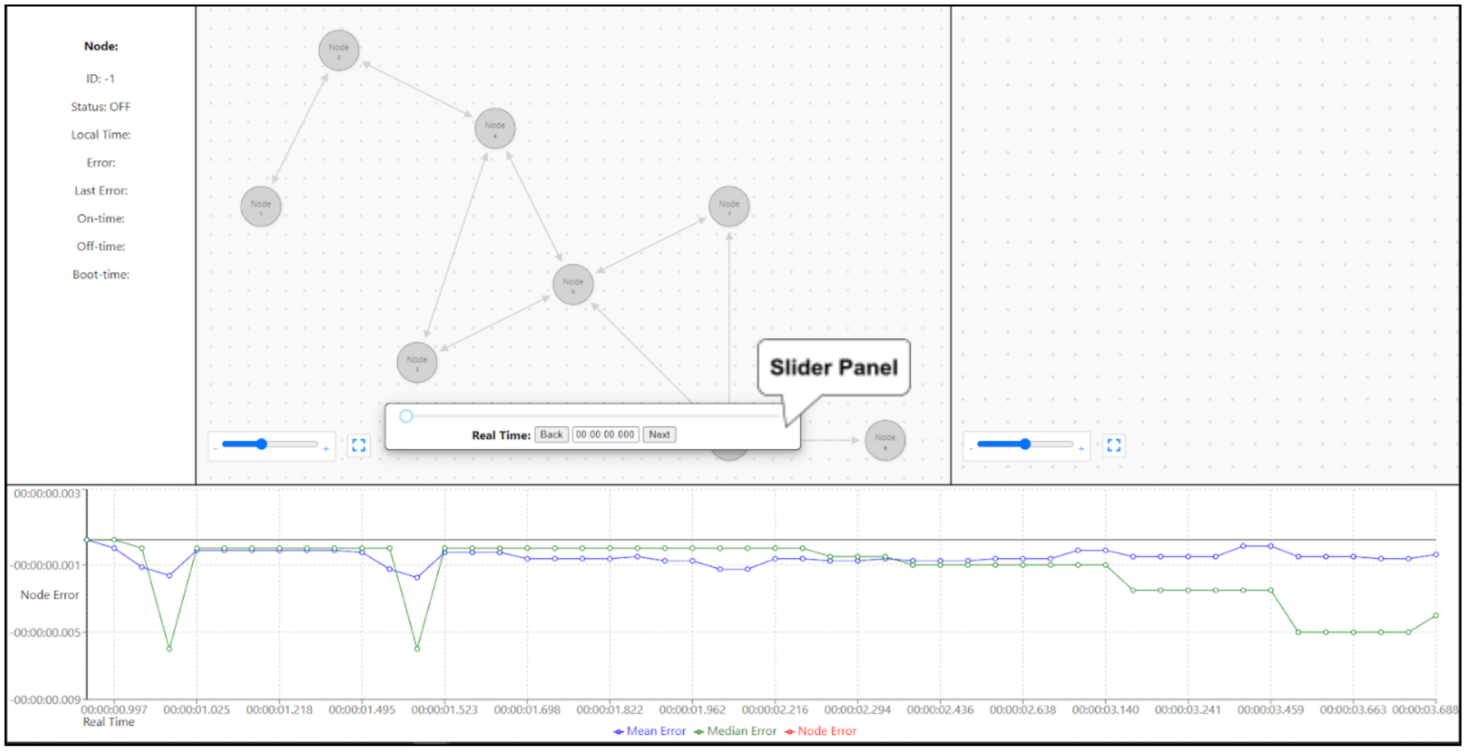
   The page will update to show that the simulation is being processed (*figure* 5.2).



Simulation Name	Status	Run	Remove Sim
simulation1	Processing...	Launch	X

Figure 5.2. Updated simulation table

2. *Run a preprocessed simulation.*
   Once a simulation has been processed, the page will update, and the "*Launch*" button will enable. Click the "*Launch*" button next to the simulation you want to run. You will be forwarded to the main page (Figure 5.3). This page displays the simulation at time 0, and the mean and median errors for all the nodes at the beginning of the simulation.



Node:

ID: -1

Status: OFF

Local Time:

Error:

Last Error:

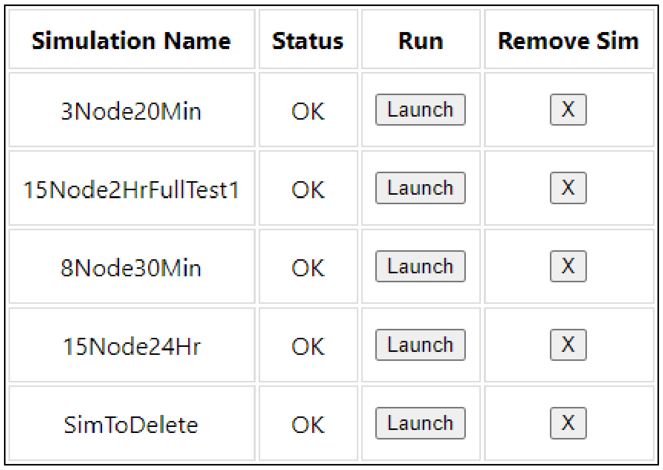
On-time:

Off-time:

Boot-time:

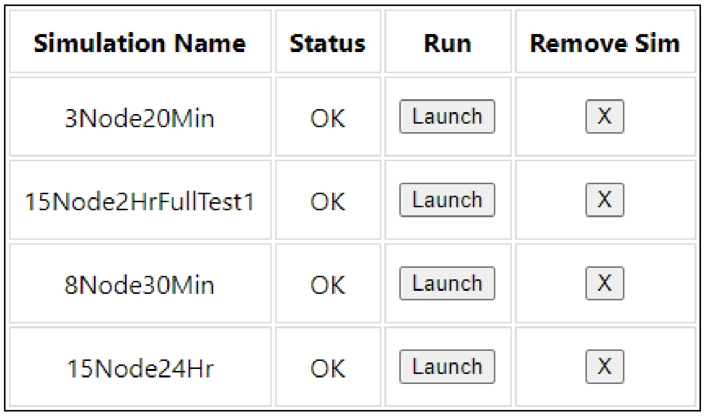
Figure 5.3. Main visualization page

3. *Delete a simulation.*
   To delete a simulation, click the "X" button next to the simulation you want to delete. The simulation will be deleted from the database, and the table will update to reflect that.



Simulation Name	Status	Run	Remove Sim
3Node20Min	OK	Launch	X
15Node2HrFullTest1	OK	Launch	X
8Node30Min	OK	Launch	X
15Node24Hr	OK	Launch	X
SimToDelete	OK	Launch	X

Figure 5.4. Simulation table before deletion

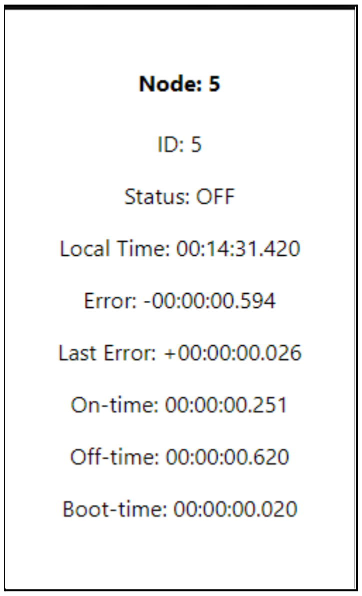


Simulation Name	Status	Run	Remove Sim
3Node20Min	OK	Launch	X
15Node2HrFullTest1	OK	Launch	X
8Node30Min	OK	Launch	X
15Node24Hr	OK	Launch	X

Figure 5.5. Simulation table after deletion

4. *See the next "interesting" event.*
   To advance to the next "interesting" event, click on the "Next" button in the *Slider Panel* (*figure* 5.3).

5. *See the previous "interesting" event.*
   To advance to the previous "interesting" event, click on the "*Back*" button in the *Slider Panel* (*figure* 5.3).
6. *Advance to a desired point in the simulation.*
   To advance to a desired point in the simulation, do one of the following:
   - Move the slider to the desired point of simulation time.
   - Enter the desired time in the time input field (time format: hh:mm:ss:mmm).
7. *See detailed node information.*
   To see more detailed information about a node, click on the node. The panel on the left will update with the detailed information.



Node: 5

ID: 5

Status: OFF

Local Time: 00:14:31.420

Error: -00:00:00.594

Last Error: +00:00:00.026

On-time: 00:00:00.251

Off-time: 00:00:00.620

Boot-time: 00:00:00.020

Figure 5.6. Node details panel

8. *See error propagation through time for a node.*
   To see error propagation through time for a node, click on the node. The graph on the right will update with the history of communications for the node going back from the current real time.

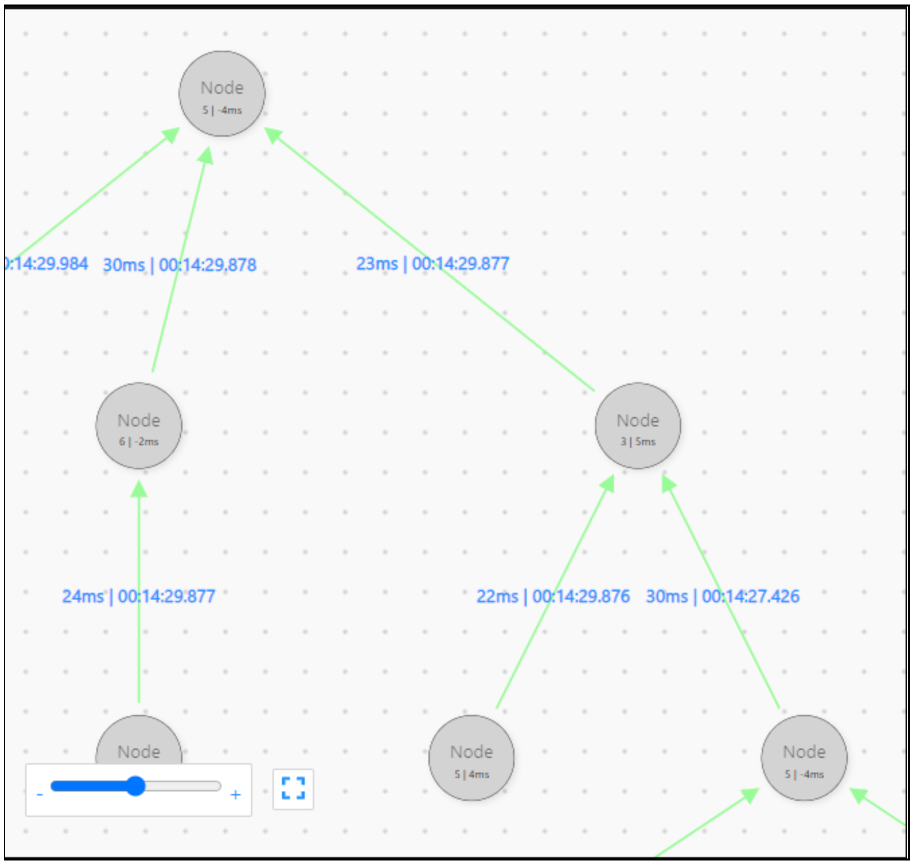


Figure 5.7. Error propagation tree

9. *See node's error over time.*
   To see node's error over time, click on the node. The graph at the bottom of the page will update to show the node's error over time. Moving throughout the simulation will move the error graph accordingly.

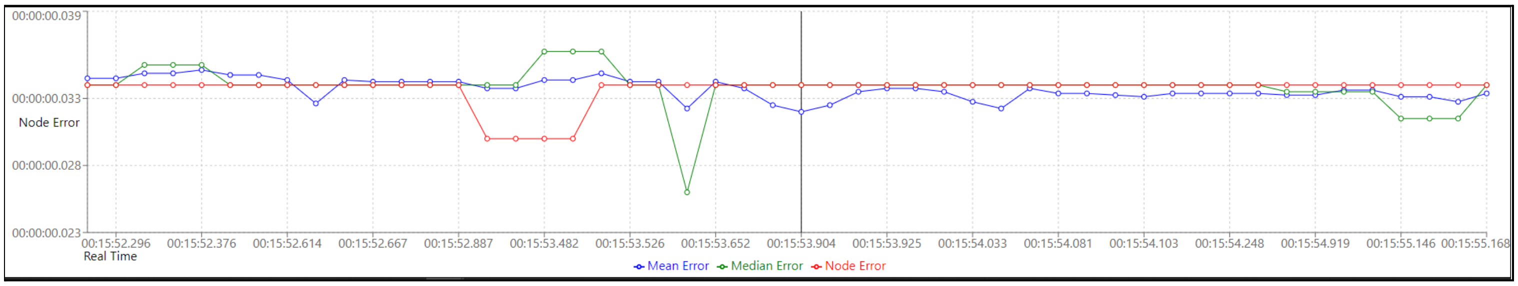


Figure 5.8. Error over time graph

Stopping the program:

- To stop the program, press Ctrl+C in the terminal window, in which you ran "serve -s build".

5.3.2 Appendix II: Alternative Designs

The frontend went through two alternative designs. Both of these designs involved a different graphing library used to display the nodes in the network graph and the error tree graph on the main page of the visualizer. We used the GoJS library for some time, but it was a proprietary library and our client then asked us to use an open source library. We shifted to using ReactFlow for a time, but eventually decided to switch again because the edges and arrows connecting nodes did not have the desired functionality we needed them to have. We settled on using React-Digraph to display these graphs on the visualizer as it's open source and has the features we need.
Another alternative for the frontend design was to code the frontend using AngularJS and not ReactJS. The deciding factor was our frontend developers having some experience with ReactJS, while having no experience with AngularJS.

5.3.3 Appendix III: Other Considerations

Plain-Text Logging Example

The following text is a sample excerpt from the output of a 15-node simulation when plain-text logging is enabled. It shows several interesting events during the simulation, including nodes booting, turning on, turning off and attempting communication.

```
...
00587761: N2 booting (local time is 587778)
00587771: N6 booting (local time is 587761)
00587781: N2 on and consuming energy (local time is 587798)
00587781: Node 2 with local time 587798 sending to Node 1 with local time 586511, attempting to
update 1's time to 587155
00587781: Node 2 failed to receive time from Node 1
00587781: Node 2 with local time 587798 sending to Node 4 with local time 586334, attempting to
update 4's time to 587066
00587781: Node 2 failed to receive time from Node 4
00587791: N6 on and consuming energy (local time is 587781)
00587791: Node 6 with local time 587781 sending to Node 5 with local time 587786, attempting to
update 5's time to 587783
00587791: Node 5 received time from Node 6, updating its local time to 587783
00587792: Node 5 with local time 587784 sending to Node 6 with local time 587782, attempting to
update 6's time to 587784
00587792: Node 6 received time from Node 5, updating its local time to 587784
...
```

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.

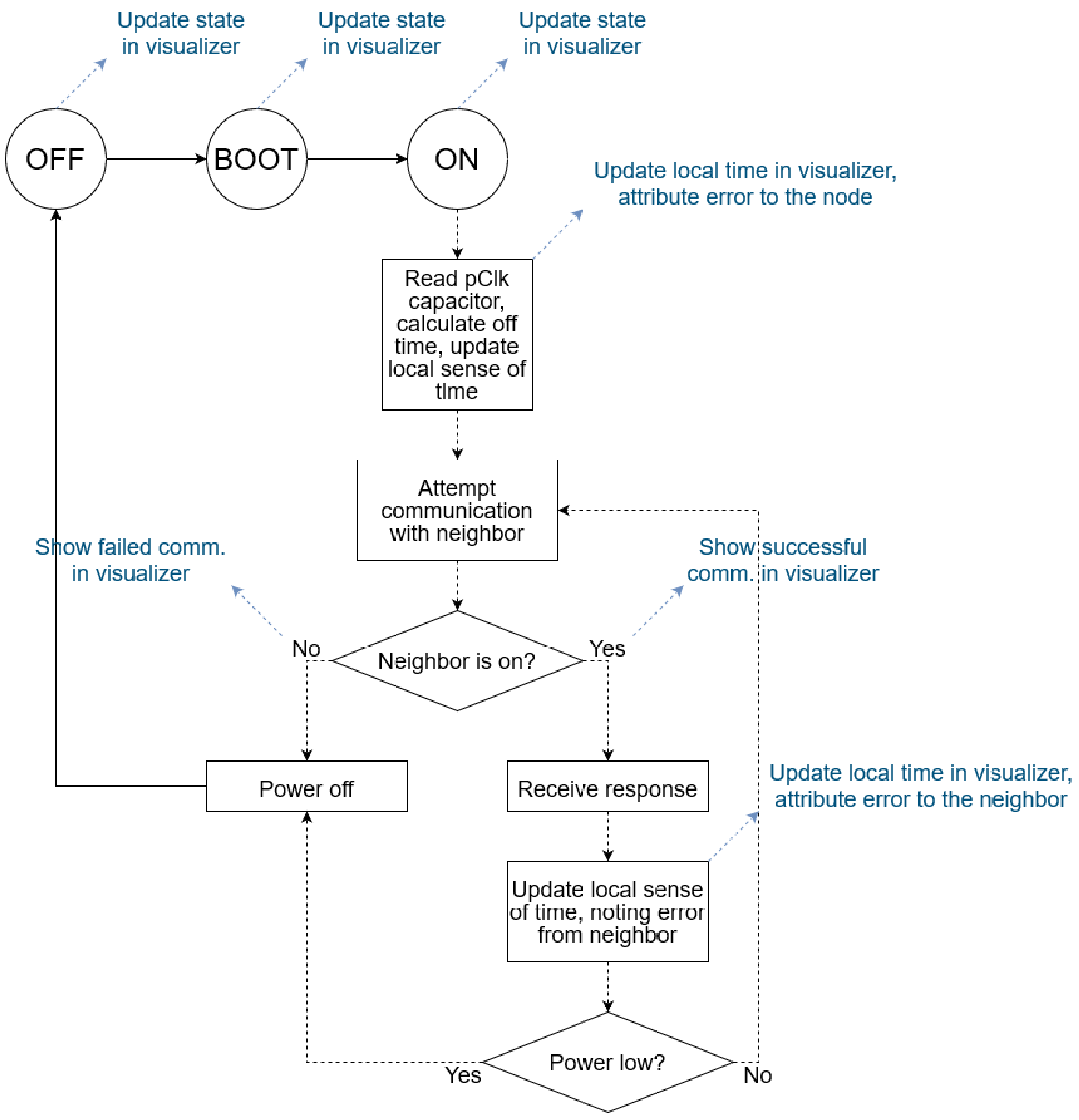

Update state in visualizer
Update state in visualizer
Update state in visualizer
OFF
BOOT
ON
Update local time in visualizer, attribute error to the node
Read pClk capacitor, calculate off time, update local sense of time
Attempt communication with neighbor
Show failed comm. in visualizer
Show successful comm. in visualizer
No
Neighbor is on?
Yes
Power off
Receive response
Update local time in visualizer, attribute error to the neighbor
Update local sense of time, noting error from neighbor
Yes
Power low?
No

Figure 5.8. Node lifecycle