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

Design Document

Team 15 (May 2021)

Client/Advisor

Professor 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: October 25, 2020 (v2)

Executive Summary

The proposed solution is to create a web-application that will allow the user to visualize and debug shared sense of time on batteryless sensor networks. 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 microservices architectural style to allow for code maintainability and easy additions of new functionality. Microservices will provide RESTful APIs that will be used to provide access points for the frontend application and for communication between microservices.
- This communication will take place using the HTTP requests. We decided to use HTTP instead of HTTPS because the client does not have any requirements regarding security of the data.
- 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 (e.g. Javadoc).
- We decided to use GitLab for both source control and task tracking. The plan is to also integrate a CICD pipeline for our project in GitLab.
- 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. For risk management, we will use Jackson's principle of commensurate care.

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
- 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

List all new skills/knowledge that your team acquired which was not part of your Iowa State curriculum to complete this project.

- GoogleTest
- CMake
- UNIX-domain sockets

Table of Contents

List of figures

List of tables

List of symbols

List of definitions

Backend	Application for storing and processing time data to be exported or sent to the Frontend
Frontend	Web application for viewing the sensor network's time data
GUI	Graphical User Interface
Simulator	Application that produces time data that simulates the time data that would be coming from a sensor network

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.

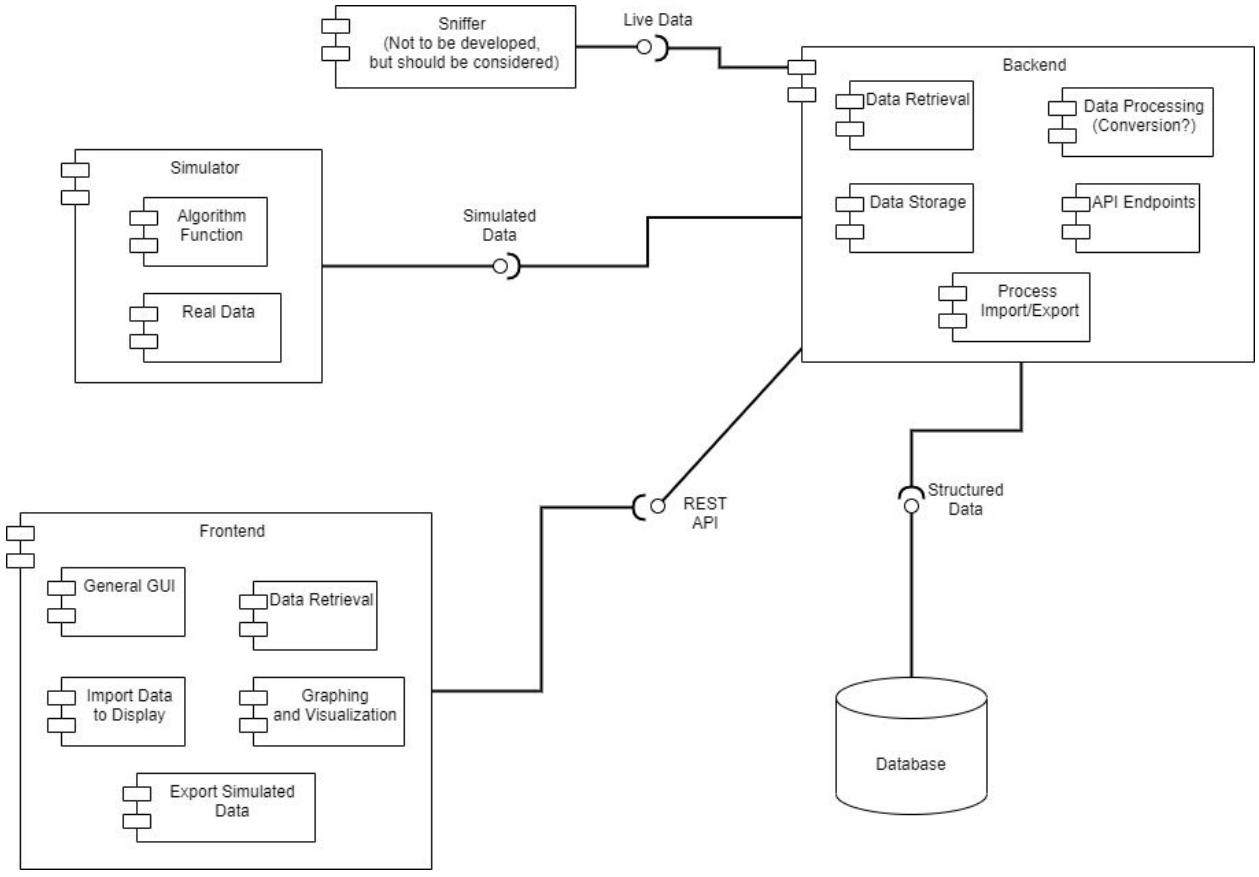

Figure 1.1: High-Level System Diagram

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, JavaScript, and WebGL. 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 "replay" past data.
- 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 Professor Duwe and his graduate students.

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 Professor 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

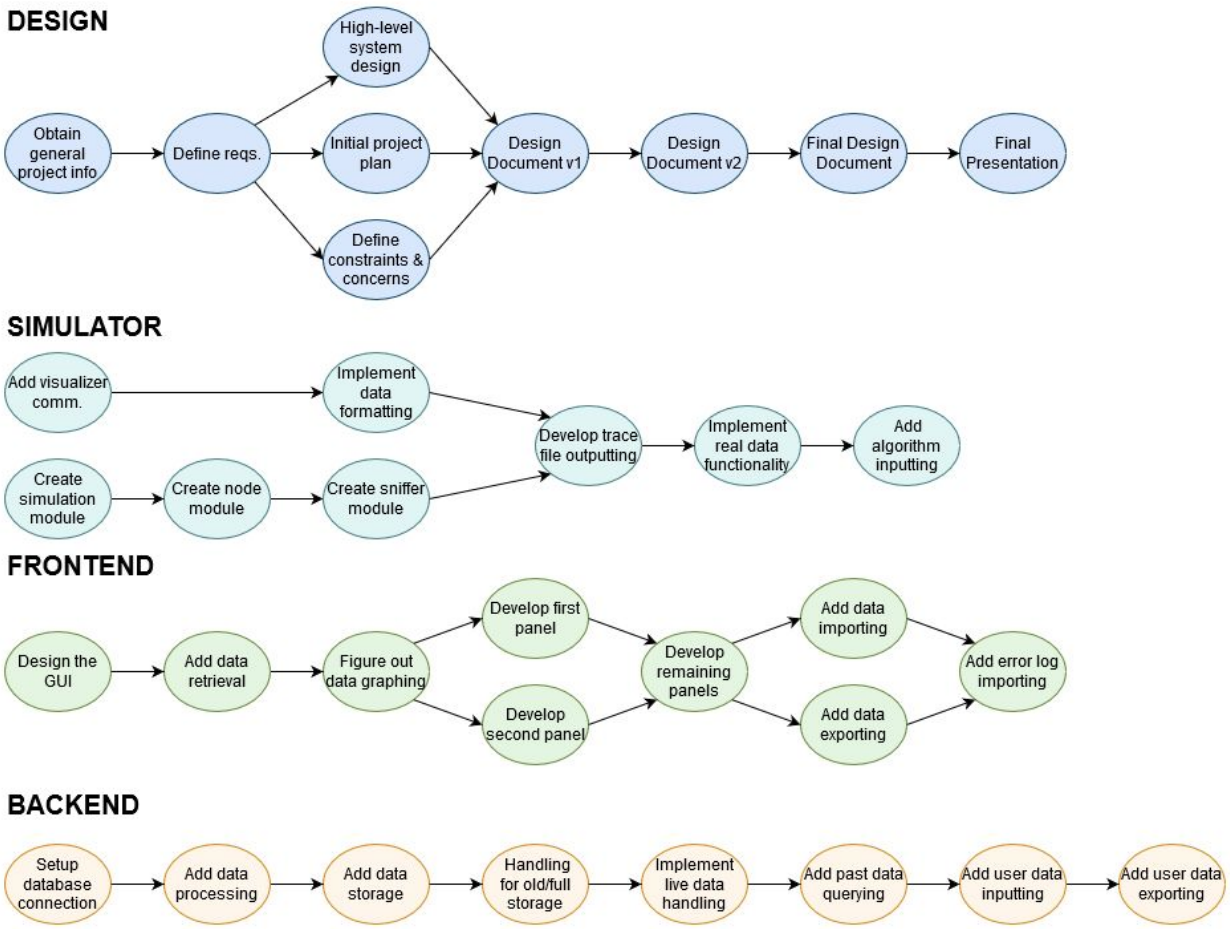

Figure 2.1: Task Graphs

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		**16**		
	Create simulation module	2		0.5
	Create node module	1		0.25
	Create sniffer module	1		0.25
	Add communication with visualizer via sockets	2		0.5
	Implement data output to a trace file	2		0.25
	Develop trace file outputting	2		0.25
	Implement real data functionality	3		0.5
	Add algorithm inputting for simulations	3		0.25
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		**12**		
	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
	Drop data when database is full, or it is too old	2		0.25
	API Endpoints:	**7**		
	Provide live current data	1		0.75
	Query and return past data to “replay”	2		0.75
	Accept input data from user	1		0.25
	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 good 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 setup and working

2.4 Project Timeline/Schedule

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

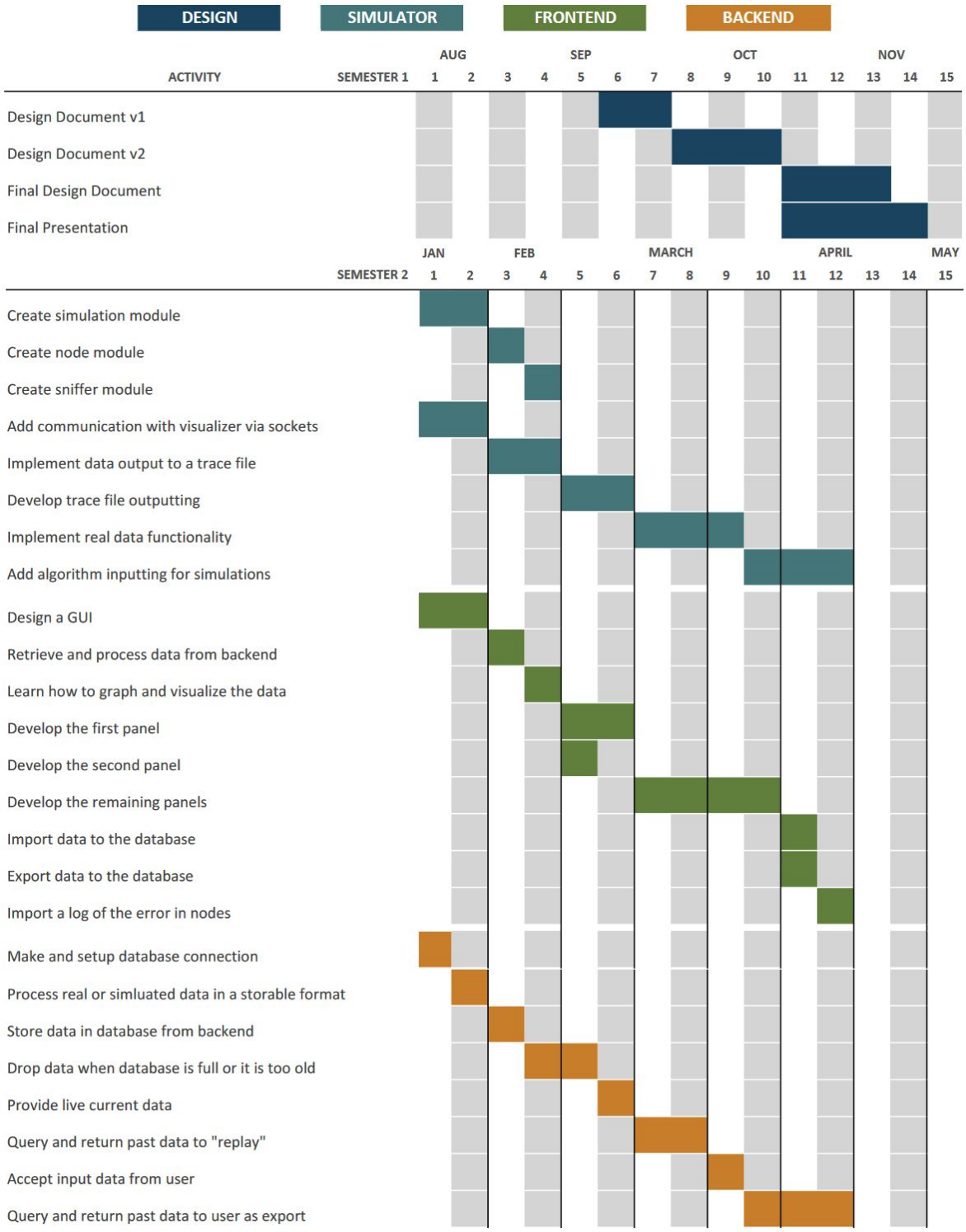

Figure 2.2: Gantt Chart

All of the Design tasks will be completed in the first semester. The work on consecutive design documents will likely begin while the previous document is being completed. The Simulator tasks 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. The next section of the Gantt chart 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. The final section shows the Backend tasks; 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.

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 (16 person weeks) - Anthony, Quentin
Frontend (14 person weeks) - Riley, Maksym
Backend (12 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

Professor Duwe and Vishal Deep are the main researchers on the topic of batteryless nodes' shared sense of time that have shared their prior work with us. Their research group is currently working on a paper and have given us access to it. However, it is not yet published, so it cannot be publicly cited here.

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 Prof. 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. Prof. 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 Prof. Duwe.

Much of the ideate phase was completed for us by Professor 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 Professor 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 Prof. Duwe, he was able to provide more ideas for us to use, especially with the backend design.

3.3 Proposed Design

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 C++ 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 and a priority queue of events organized by time of occurrence. The sniffer, controlled by the simulation, will periodically collect data from the sensors (nodes) in the simulated network.

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.

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.

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 rest API connected to 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.

3.4 Technology Considerations

Simulator

- C++ programming language
    - C/C++ Standard Library / STL
    - <sys/socket.h>
    - CMake
    - GoogleTest

The simulator will be built in C++ since it is convenient to use for developing applications for UNIX systems. A major strength comes with the availability of C++ libraries, including the standard library and the socket library that will enable communication between the simulator and the visualizer through UNIX-domain sockets. One alternative that we considered was using Python rather than C++ to implement the simulator, but we decided that it would be better to use C++ because it has improved performance over Python which is important for our application. 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 C++ through the <sys/socket.h> library which has more support than most WebSocket libraries for C++. We will use CMake for the build system for our project, including the testing components. For unit testing, we plan to use the GoogleTest framework because it has support for mocking.

Backend

- ExpressJS (NodeJS)
- MongoDB database

- Jest

These languages will allow us a simple yet robust methodology for proving REST API's 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. 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. Although our knowledge is stronger in other databases, MongoDB's benefits outweighed our strengths. 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
- WebGL
- Selenium
- Jest
- MirageJS

Using these languages we will be using fairly basic web development tools, but they can be quite powerful. We thought about using ReactJS or Typescript, but the learning curve vs. the benefit just does not seem worth it at this time. WebGL may be fun to learn and use for better visualizations of our simulation, but we'll have to see how easily it integrates with everything before we fully commit to learning it.

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 for the full system.

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 Professor 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 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. 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 perform some 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
    - Collection of data from a simulated sniffer
    - Processing of events and calculations of subsequent events
    - Data outputting to ensure the format is correct
- Sniffer
    - Gathering data from a node event
    - Calculating error from a node's sense of time
- Nodes
    - Sending an event message
    - Transitioning between stages
- Sockets
    - Establishing a connection
    - Sending data over a connection
    - Serializing and de-serializing event data

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.

Frontend

Many visualizer components will be tested in isolation from the other parts of the system. Sensor nodes, replay functionality, start, stop, displaying statistics and errors in the node times, etc. The underlying classes to these components, and the RestAPI requests/responses to the backend will also be tested.

4.2 Interface Testing

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

- Simulator and visualizer (front-end/back-end)
- Front-end and back-end

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

4.3 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.

4.4 Results

We have planned the tests for our project, but at this point, we have not completed the testing phase so we do not have results to report.

5 Implementation - TODO for the final design document

Describe any (preliminary) implementation plan for the next semester for your proposed design in 3.3.

6 Closing Material - TODO for the final design document

6.1 Conclusion

Summarize the work you have done so far. Briefly reiterate your goals. Then, reiterate the best plan of action (or solution) to achieving your goals and indicate why this surpasses all other possible solutions tested.

6.2 References

List technical references and related work / market survey references. Do professional citation style (ex. IEEE).

6.3 Appendices

Any additional information that would be helpful to the evaluation of your design document.
If you have any large graphs, tables, or similar data that does not directly pertain to the problem but helps support it, include it here. This would also be a good area to include hardware/software manuals used. May include CAD files, circuit schematics, layout, PCB testing issues, software bugs, etc.