S.A.R.C.I

Search And Rescue Coastal Intelligence

Final As-Built Report

5/03/2022

Team Mentor: Han Peng

Group Members: Dylan Woolley, Vidal Martinez, Randy Duerinck, Jabril Gray

Team Sponsor: General Dynamics Mission Systems

Table of Contents

1. Introduction

The United States Coast Guard (USCG) oversees 296,000 nautical miles of sea, utilizing the Rescue21 system for Search and Rescue missions. General Dynamics Mission Systems (GDMS) assists the USCG by maintaining the Rescue21 system. The Rescue21 system is made up of a system of antenna towers called remote fixed facilities (RFFs), each placed strategically off the coast of the Continental US (CONUS), major rivers, lakes, and islands (Hawaii, Puerto Rico, and Guam). Each RFF features very high-frequency radios capable of picking up distress signals. Last year's capstone team developed a Site Weather And Power Recorder (SWAPR) device for GDMS to collect power and weather data at an RFF; however, the current functionality produces data that is difficult to access and read. Our project expands on the functionality of the SWAPR device to provide software capable of collecting, storing, and analyzing weather data collected by SWAPRs. It displays this information through a clean and concise Graphical User Interface (GUI) for technicians and coast guard operators to access and use. This information is then used to improve time spent on maintenance and help reduce maintenance costs. The project is broken into five different subsystems: the Simulator, Reader, Database, Website, and Orchestra Subsystems. Each subsystem has been categorized to serve specific functions.

The essential user-level requirements of this system include actions across all five subsystems. The Simulator Subsystem must create data imitating the SWAPR device's output to emulate the real-world SWAPR hardware, collecting data and producing a string list of data to be sent to the system's database. The Reader Subsystem must receive this SWAPR data, and then it can be securely transmitted to the database and stored. The Website Subsystem must establish a secure website environment with authentication. On the website, there must be two summary views for the SWAPR devices: the list view and the map view. The list view includes a website page presenting every SWAPR device visualized as informative boxes showing real-time data. The map view similarly offers every SWAPR device, but as a node on a map providing the geographical location and operational status of the RFF. The website must also provide a graphical view of historical data stored in the database. Historical data refers to data that has been collected previously. The views will include graphs or charts such as a line, bar, scatterplot, and wind rose compass. The Website Subsystem must create a notification if an RFF has operational issues. The notification will be viewable by admin users and will give a general status code for the RFF site and a short description of what caused the status change. The Website Subsystem must also provide the option of exporting data as a .csv file for users who need to store the data locally on their machine. The Orchestra Subsystem is considered a stretch goal for our project. The Orchestra Subsystem will allow us to emulate a network containing hundreds of SWAPR devices to stress test our system to find any bugs or vulnerabilities.

We also have several functional requirements for each subsystem. The Simulator Subsystem needs to be able to generate SWAPR data for valid and invalid data randomly. It also needs to be able to send data over virtualized com ports via RS-232 protocol. Next, our Reader Subsystem must be able to read data over virtualized com port via RS-232 protocol and establish a TCP client connection with the database queue. For our Database Subsystem, there are several functional requirements. The Database Subsystem needs to establish a connection with the Reader Subsystem. It must queue collected SWAPR data in AWS Message Queuing Service (SQS). The Reader Subsystem will send data to the AWS SQS queue, which will be pulled to create database entries. Finally, our Website Subsystem is made of both the frontend and backend. There must be secure authentication between subsystems and user accounts and role-based permissions on the website. The website will have the ability to query the database server safely. It will have a notification system that detects a problem with the RFF and creates modals for specific user groups. Modals are windows that show up in front of the main screen for particular actions and deactivate the part of the screen not occupied by said modal. The Website Subsystem should have the ability to create a list-view, map-view, and a historical data summary of the latest data from all SWAPR devices in the network. When in a graphical view of historical SWAPR data, the user should be able to export the selected data to a comma-separated file. The Orchestra Subsystem needs to be able to connect any number of Simulator and Reader Subsystems over virtual com ports for lab environment stress testing. To achieve the functionality of our system features, different tools were used to facilitate the process.

2. Process Overview

The development of our team's project required tools to assist in task management, communication, record keeping, and code organization. Our team used GitHub's version control feature to manage and order updating our project. GitHub also allowed us to manage our code integration using a release branch, develop branch, and working branches off of the develop branch. Together, this streamlined updating code and merging commits between multiple team members. Our team primarily used Discord for direct messaging and communication, and Discord was also used for document sharing and record-keeping for meetings. This platform allowed our team to record critical events and notes and communicate effectively and efficiently. Google Drive was also used to handle document sharing and act as the main repository for our documents and deliverables.

Our team assigned Dylan as our team leader. In his role, he acted as the leader in communication with our client, led the conversation on significant decisions, and arranged events for the capstone deliverables. Jabril, Randy, and Vidal all filled support roles assisting in submitting deliverables. The functions of architect and release manager were initially set to Vidal and Jabril, respectively, but these roles became a shared responsibility within the team as time went on. The recorder role was given to Randy, who maintained the recording of meeting notes.

3. Requirements

As stated in our introduction, our system consists of five main subsystem components that form our project. Our aim is for this project to provide functionality for a secure website with retrievable weather data for creating informative visuals. The performance requirements specify our need for speed and ease of use. The Simulator, Reader, Database, Website, and Orchestra subsystems are responsible for meeting our project requirements.

- Create data imitating the SWAPR device's output
- Take data from a SWAPR device and send it securely off-site
- Store and serve SWAPR data in a secure manner
- Establish a safe website environment with authentication
- Create summary and historical views for the SWAPR devices and data in the network
- Create a way to notify an operator when there is a problem with an RFF site

The Simulator Subsystem directly connects to our Reader Subsystem, and together they drive our natural environment emulation by providing simulated SWAPR output and secure data transfer. The Reader Subsystem must read data, then package the received data over a TCP connection to our Database Subsystem. The data from our database is then retrievable by our Website Subsystem. Authorization is required to access our database by user-based role authentication from the Website Subsystem. The retrieved data can then be viewed in different historical and summary views on our Website through the graphical user interface. The historical perspective includes graphs or charts generated using data from the database. The summary view provides the list and map view; both show details of each SWAPR device and give the user an understanding of summary information such as their location, current temperature, wind speed, and more. Notifications are needed for our system so admins are up-to-date on the status of the devices. Data must also be stored locally from the Website so users can save data when necessary. The Orchestra Subsystem is a stretch goal for our project. It utilizes the Simulator and Reader Subsystem to create an easily reproducible instance of the simulated SWAPR device output and data transmission. The Orchestra Subsystem will allow hundreds of instances to be made to mimic a SWAPR device network. This communication occurs between the instance in the orchestra and the Database Subsystem. The Orchestrator Subsystem will allow us to stress various test aspects of the sending and receiving data from device to database.

3.1 Simulator Subsystem Requirements

The Simulator Subsystem considers the random generation of SWAPR data and data transmission over a virtualized com port on the RS-232 protocol as the prominent functionalities. Random data generation gives our environment a reproducible, ostensibly non-deterministic model to provide concise and practical test cases for assessment as we develop our project. An example of this is wind direction. Our simulator will have constantly changing wind speeds for a given SWAPR device-produced output to simulate a real-world scenario. The Simulator Subsystem will generate random values ranging from 0 to 360 degrees to test handling and viewing of this data output from a given SWAPR device. Additionally, the value generated will not be the same across multiple devices to emulate a natural environment.

Sending this generated data from our simulated SWAPR devices over the virtualized com port allows additional testing conditions to evaluate. Altogether, we build a subsystem capable of assessing the expected circumstances the Rescue 21 system will operate under within a controlled environment without needing a physical SWAPR device. This is a vital function in our system. Without a virtualized serial port in our simulation, we would have no way of sending data between our applications, the reader software, and the database. This complicates simulating the SWAPR device's mode of data transmission but would also leave us incapable of testing this part of our system and potentially developing an incompatible system to work with the hardware version of SWAPR devices.

When considering these functional requirements, we also evaluate the performance requirements. The SWAPR device simulation will need to be capable of generating output data in five-second intervals, or more frequently if possible. This data must be generated at such a high frequency because this data from the SWAPR devices is the basis for all analysis and visualization the users will execute and produce, respectively. Suppose data generated to simulate a SWAPR device is produced on an interval different from reality. In that case, we may face a skew in our systems expectations of the SWAPR device's performance which will likely impact the development of our system.

The output data generated from the simulator must be transmitted by the virtual com port to the reader software in the Reader Subsystem at a 9600 baud (Bd) rate. A baud rate refers to the rate of bits per second. It is essential for the rate of bits transmitted from a SWAPR device simulation to the reader software through the virtual com port to imitate real standards for bit rates. As a maximum bit rate in the serial port, this data transfer rate guarantees our simulation imitates a natural and expected standard. A 9600 Bd rate is an adequate speed capable of streaming over ten lines per second.

3.2 Reader Subsystem Requirements

Our team needs two functional requirements in the Reader Subsystem: the data needs to be read over the virtual com port using the RS-232 protocol, and the TCP client connection needs to be established with the database. The read data transmitted over the com port comes from the Simulator Subsystem. The necessity of this functionality is the same for the Simulator Subsystem's functionality for sending data over the virtual com port. Using the virtual com port imitates a real-world activity a SWAPR device will have, allowing our team to test this subsystem with realistic cases.

Establishing a TCP connection with the database queue is a process used in the actual implementation of our system. This connection is necessary for transporting all data from a SWAPR device to our database queue for usage in the Website Subsystem. The Reader Subsystem is expected to handle data transportation with the assertion that our database always receives generated SWAPR device data from our Simulated Subsystem when a TCP connection with the database queue is acknowledged.

The functional requirements we are concerned with also impose a couple of performance requirements for our subsystem. The virtual com port connection between the Simulator Subsystem and Reader Subsystem must have a 9600 Bd rate; this requirement is identical to the Simulator Subsystem performance requirement. A standard rate for communication between both subsystems is a standard for com port data transmission and should be met as any other adequate system requiring data transmission would.

The data received from the Simulator Subsystem needs to be transported over the TCP connection to our database in at least five-second intervals. This means that our connection must have a round-trip time (RTT) of five seconds, where our database queue fully receives a data entry from the Reader Subsystem every five seconds. Our time restraints are calculated based on storing data in our database within a feasible time to avoid delays in other subsystems down the pipeline, such as the Database Subsystem or the Website Subsystem.

3.3 Database Subsystem Requirements

The functional requirements for the Database Subsystem are primarily concerned with TCP connection and building the database. The TCP host connection needs to be established with the Reader Subsystem to allow data transmission between the Database and Reader Subsystem.

The data incoming from the Reader Subsystem will be placed in a queue using the Amazon Web Services Simple Queue Service (AWS SQS). Storing the data from the TCP connection in a queue using the AWS SQS is the first step in storing SWAPR device data in the database and making it usable in the Website Subsystem.

Once in the queue, AWS Lambda Functions will make database entries from the SWAPR data queue. The database entries are required for building our database on the AWS server. With a database populated with entries acquired from the TCP connection, we will have all SWAPR device data accessible by users in the database.

The hosting for the database server will use the Amazon Web Services Relational Database Service (AWS RDS). The RDS will be necessary for hosting our Database Subsystem, which can be used with our Website Subsystem. With a secure TCP connection between the server host and Reader Subsystem client, an established database, and hosted by our AWS server, our Website Subsystem can access any available data generated by the Simulator Subsystem.

The Database Subsystem performance requirements deal with the specifics for the data queue in our database and the total storage expectation of our database. The data from the TCP connection established with the Reader Subsystem must be received and placed in a queue in five-second intervals. This time interval is necessary to guarantee adequate time for this Subsystem to acquire data and place this data in a queue without underestimating the time it will take.

As mentioned with the Reader Subsystem, we want to guarantee a sufficient turnaround time for generated data from the SWAPR devices. The database needs to be capable of communicating with more than 200 facilities, each with its own SWAPR device. Our database should be capable of handling inputs from every connected device in an uninterrupted, real-time stream, with each SWAPR device's data being stored in their own data entry within the database.

We need the data entries to be built within five seconds to maintain access to our database's data. This is important to maintain a consistent speed from data generated in the Simulator Subsystem to entry creation in the Database Subsystem. With every interval of data entry creation and insertion into the database taking approximately 20 seconds, the data stored in the database should be accessible reasonably and accurately. We believe this length of time may be generous, but once our team completes our prototypes, we should understand precisely how long this process can take.

Not only is the speed of our data generation and storing important in performance, but the capacity of data that can be stored is essential in our Database Subsystem. The total amount of data stored in our database should be capable of reaching at least a year. This would mean that with a 24-hour run-time with data generated and stored from a single SWAPR device in five seconds, we should expect to keep approximately six million data entries, with each entry being generated from a single SWAPR device's output. This will be done for 250 SWAPR devices, which creates about 1.5 billion data entries. Our estimate of a given file is about 50 bytes. Therefore, the total amount of data stored on the server will reach 75 gigabytes after a year.

3.4 Website Subsystem Requirements

The functional requirements for the Website Subsystem Backend are hosting the Blazor Server on AWS, generating a graphical view of historical SWAPR data, generating notifications with a notification system, and establishing secure authentication between subsystems as well as "faking" user accounts and role-based permissions.

We need to create a Blazor Server application that will take and display information retrieved from our database. The Blazor Server can be hosted on AWS to allow features covered in the following functional requirements to be achievable.

Once we can query data, we will need to use the historical data from the database to create graphs for the user's view. Graphical views of the historical data will allow users to view the data in organized and summarizable ways to improve the user experience. For example, a line graph, a wind rose compass, or a radar chart.

Our notification system will be responsible for notifying users when there is problematic information or a response signifying a critical event in a SWAPR device's output data once received by the backend of the Website Subsystem. Our client requires the notifications to be displayed when the following admin user logs in to the website. Upon detecting a critical event in the backend of the Website Subsystem, a notification is created instantly in collaboration with the frontend interface.

There are four status types for a SWAPR device. There is green which means that everything is working correctly and the data is in acceptable ranges. Yellow represents an issue with the weather data, which is detected from data outside of acceptable ranges. Orange means that there is a problem with the recorded antenna power. Red means that there was no response received from the SWAPR device, so the site is offline, and the problem is unknown. This is important for informing technicians that are not on site of the issue and allowing the technicians to visit a site with the knowledge of its operational status beforehand.

The Coast Guard and GDMS need to ensure that only authorized users have access SWAPR devices and their data. We need a secure authentication system that consists of two roles: user and administrator. We need authentication compatible with the Windows operating system because this maintains consistency with General Dynamics' requirements. Currently, we will fake the user and administrator roles because of budget and time constraints. The software needed is available and applicable to the purpose, and the software will be recessed at the end of the project for implementation.

For our Website Subsystem Backend, we have several performance-based requirements. The Website Subsystem needs to retrieve data for summary and historical views in under five seconds, then render the graphical map view of the SWAPR device data taking no more than six seconds for retrieval and ten seconds for rendering. Our team is looking for a wait time of 30 seconds or less to view our database's most recent incoming data in the desired format.

Our backend also needs to take in historical SWAPR database information and extract it to a CSV file in under three seconds. Creating the CSV file in this time frame helps guarantee the data can be accessible in a downloadable, text-based format for a given user's usage.

The functional requirements for the frontend side of the Website Subsystem are focused on the ability to create a list-view summary of the latest data from every SWAPR device in the conceptual network of RFFs. The same is needed for our website Subsystem map visuals; we need to create a map-view summary of the latest data from every SWAPR device. Having this functionality provides users with alternative views to text-based displays.

Our team needs our website to create a static, graphic view of historical data for any given SWAPR device, and as an additional stretch goal, we want to have the graphical view be interactive. On the other end of the Website Subsystem, the frontend portion of functionality is specifically relevant to the interface the user sees and interacts with. The historical data in the database should be viewable as a graph, whether a wind compass, line graph, or bar graph, for example. The interactivity we look to add expands beyond the static images generated to include adjustable charts. This, for example, could be an adjustment of the range of dates or the content of values affecting either axis on a graph.

The performance requirements are necessary for defining the speeds to which data is created and updated on our website's front. Our website may suffer poor refresh times and load times without consideration and efficiency. The time to process and make our list-view summary widgets should be under 6 seconds and 3 seconds for the map-view summary.

The live widgets in the list view should be refreshed approximately every minute. This is the amount of time approximated based on data generation and retrieval of data from our database.

The map view will need to consider the Area of Responsibility (AOR). The AOR describes the expected capacity of RFFs to view an instance. The AOR for the SWAPR devices is currently set to 18. Our team will need a system to view these devices as their correct set sizes for clarity in their relative responsible sectors.

To generate data for visualizations on the front end of our Website Subsystem, we need to create requested graphs within five seconds for users.

Displaying critical event notifications generated in the backend of the Website Subsystem needs to occur quickly due to their importance in notifying the user of critical events. The event will need to be visible in under two seconds from the point of creation, and the notification should not be cleared until the appropriate user chooses to remove it. This goes hand-in-hand with the backend side of the Website Subsystem.

3.5 Orchestra Subsystem Requirements

The Orchestra Subsystem has several functional requirements necessary to work properly. This Subsystem needs to be able to connect to any number of simulator and reader subsystems over virtualized com ports for lab environment stress testing.

There are two non-functional requirements which are essentially simpler tests which will prove our functional requirements will work. The first non functional requirement is to be able to connect to at least one Simulator Subsystem over RS-232 protocol on a virtual com port via the Null-modem emulator mentioned above. At least one established connection is the minimum requirement by our clients request.

Our second non-functional requirement is for the simulator to be able to connect to the reader subsystem in no more than five seconds per connection. The purpose of this requirement is to make sure our system works as an individual SWAPR device and also to stress test our system before adapting the reader software to a large-scale implementation with more SWAPR devices.

3.6 Environmental Requirements

Our project has been developed in a Microsoft Windows environment to avoid any complication with our other tools in our environment. The Windows Operating System (OS) is used by our client and necessary for implementation with their systems as well. Using this OS will allow easy implementation of Microsoft Visual Studio, the Integrated Development Environment (IDE) and interface we have been required to use by our client for creating our code base.

Visual Studio is needed, because it has a range of features useful to our team. Without Visual Studio we would face more complexity in finding and using dependencies to meet the specifications of our client. Two of these features are the .NET 5 and Blazor Server products. .NET 5 and Blazor Server are two Microsoft products that are needed by our team for our systems development and easily integratable with Visual Studio in a Windows environment. The purpose of .NET and Blazor Server in our development is to provide our team with the capability to create our systems website and database.

The usage of Visual Studio with .NET and Blazor server defines an ecosystem of development tools with a built-in run-time environment for debugging and running tests. Our client has also requested our team use .NET 5 with Visual Studio. .NET 5 includes an extensive library of C# functions that can be used in our websites backend and frontend creation, database entry creation, and data transmission through TCP client connection in communication with the reader software.

The Microsoft Blazor Server is the best tool for building our website's architecture because the Blazor Server is useful for handling live data dashboards. The Blazor Server provides faster load times compared to Blazor Web Assembly, because the Blazor server allows users to download files from the site as needed as opposed to all at once.

The C# programming language is an available development language for Visual Studio. We need to use C# not only because it is available to use in Visual Studio, but because C# is a main language supported and Visual Studio provides an expansive library for C#. C# is a high-level, easy to implement, and dependable language we can use to create all of the programmatic functionality in our system. Our team has been able to establish a C# based approach to all of our programmatic problems through researching Visual Studio, .NET 5, and Blazor Server documentation.

4. Architecture and Implementation

With this overview of how our team is implementing the features of our system and the tools we plan to use, we can now explain in more detail the architecture. The architecture consists of five subsystems each with their own important mechanisms, features, and control flows. The Simulator Subsystem is responsible for providing our system with accurate and testable data entries. The Reader Subsystem is responsible for acting as the communication service between the generated data from the Simulator or SWAPR and the storing location in the database. Once data has been placed in the database, the Database Subsystem is responsible for making the generated data accessible to the Website subsystem. The Website Subsystem is responsible for providing users and admins with an intractable interface where they can access data on each RFF site's power information and weather information through various graphical interfaces. Our Orchestra subsystem, if implemented, is responsible for creating multiple instances of an RFF site. Within each RFF site, there is a SWAPR device and Reader Subsystem. However, the Orchestra will connect an instance of the Simulator and Reader Subsystems over a virtualized com port to act like a RFF site. The Orchestra Subsystem will be capable of providing any number of simulated RFF sites to provide GDMS with a lab stress testing tool.

The Simulator Subsystem is responsible for generating realistic data identical to a real SWAPR device. The data sent from the Simulator must be formatted the same as the SWAPR device’s output format, contain data for every type of power and weather attribute, and generate values within an accurate range for each type of data. The Simulator Subsystem is the starting subsystem in the control flow which simulates real data expected to be generated by a SWAPR device, a hardware prototype connected to weather gathering tools at an RFF site. The data transferred will be sent as a stream of characters to the Reader Subsystem using a virtual com port to emulate data transfer that occurs over a serial port between the SWAPR device and the Reader software. You can find a diagram showing the components of this subsystem in Figure 4.A.

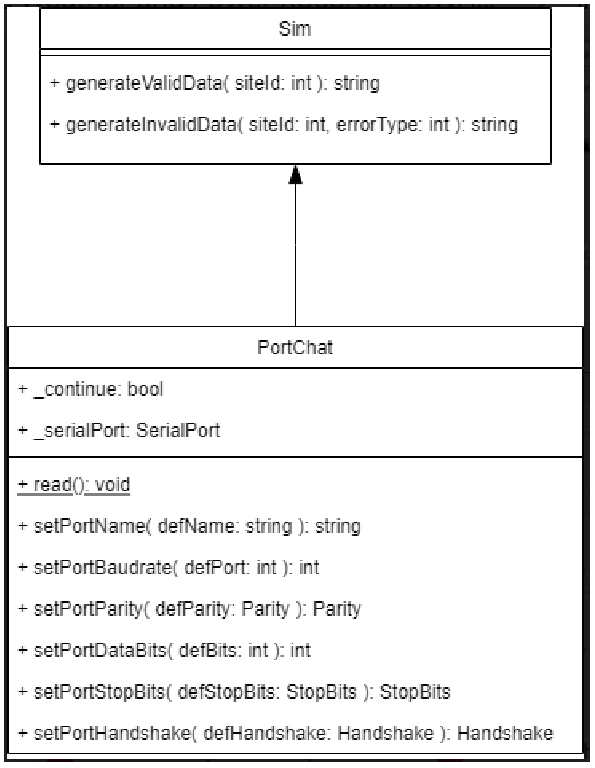
Sim
+ generateValidData(siteId: int): string
+ generateInvalidData(siteId: int, errorType: int): string
PortChat
+ _continue: bool
+ _serialPort: SerialPort
+ read(): void
+ setPortName(defName: string): string
+ setPortBaudrate(defPort: int): int
+ setPortParity(defParity: Parity): Parity
+ setPortDataBits(defBits: int): int
+ setPortStopBits(defStopBits: StopBits): StopBits
+ setPortHandshake(defHandshake: Handshake): Handshake

Figure 4.A: Simulator Subsystem Components

The Reader Subsystem is responsible for receiving the SWAPR device data and using a connection through database library tools to send the data to a database securely. As stated with the Simulator Subsystem responsibilities, the stream of data will be transmitted to this subsystem using the virtual COM port emulator. The Reader Subsystem will listen on the serial port to receive the generated data. Once the data is received it will be sent to the database using a secure connection. On the receiving end of the connection, a queue will store the data and create entries into the database from each element. This transmission of data between the Reader and Database Subsystems simulates the real-world communication between the SWAPR device's Reader software and an external database. You can find a diagram showing the components of this subsystem in Figure 4.B.

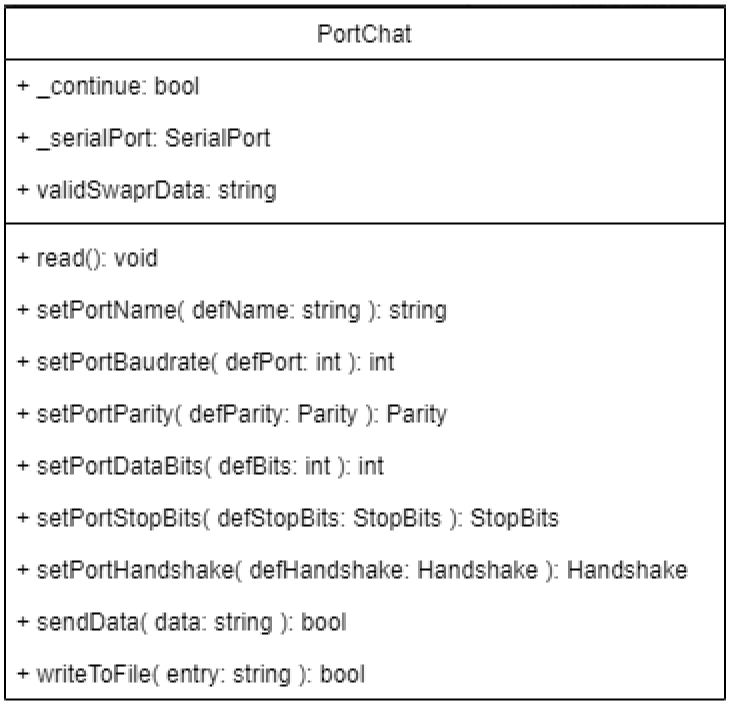
PortChat
+ _continue: bool
+ _serialPort: SerialPort
+ validSwaprData: string
+ read(): void
+ setPortName(defName: string): string
+ setPortBaudrate(defPort: int): int
+ setPortParity(defParity: Parity): Parity
+ setPortDataBits(defBits: int): int
+ setPortStopBits(defStopBits: StopBits): StopBits
+ setPortHandshake(defHandshake: Handshake): Handshake
+ sendData(data: string): bool
+ writeToFile(entry: string): bool

Figure 4.B: Reader Subsystem Components

The Database Subsystem will be responsible for hosting the database on a cloud server and listening for connections with the Reader software from the RFF sites. With the database hosted on a server, our team will be able to properly test our system in an interconnected network as it would persist in a real world scenario. The database will interpret the received query from the Reader Subsystem and insert the data into the database. The data will be stored into a data table with types for our site power and weather information. This data, once stored in the database, can be retrieved by our Website Subsystem for implementation in various graphics.

In the Website Subsystem, the website's backend will handle behind the scenes events such as notification message creation and authentication of users. The website’s frontend will provide users with visuals based on data stored in the database. The backend of the Website Subsystem must handle secure authentication between subsystems in our project and user accounts with their role-based permissions. These roles will be admin and user. The admin user will have unrestricted access to the website while the user will not see notifications and will only see their area of responsibility. The Coast Guard and General Dynamics need to ensure that only authorized users can access SWAPR devices and their data. The backend must also notify accounts with the administrator role in the event of a critical event that requires administrator intervention. To do this a modal will be created to display a message on the website interface. The frontend of the Website Subsystem must generate a graphical view in two forms: historical and summary. To display these views a query is created to request data from the database when an account attempts to view data in one of these views through the website graphical user interface. The historical view refers to the visualizing of SWAPR device information as graphs and charts. The summary view comes in two different forms. The first is a list view which shows each SWAPR device at its RFF site as a simple widget which displays the real-time information for the site. Each of these widgets will be side-by-side on a website page. The second is a map view where each of the RFF sites is placed on a map of the United States. Each of the RFF site icons on the map are collapsible with the ability to select the site and open a window providing details on the operational status of the selected RFF site. Once data is available in the database, a query can be created for the database to retrieve data to provide information on the sites and weather calculations at these sites by user or admin request.

After the mandatory system features have been implemented, we will provide the desirable Orchestra Subsystem. The Orchestra script we create will take the processes of the Simulator and Reader Subsystem's functionality and communication and recreate it in multiple instances thereby replicating that control flow from the Simulator to the Database subsystems. Creating an environment with at least 250 SWAPR devices and RFF sites will give our team the ability to stress test our Database and Website Subsystems. We chose 250 SWAPR devices because this is the estimate that our client gave us for the number of RFF sites in the Rescue21 system. By having this subsystem in our architecture, we will be able to provide a better tested and reliable system.

5. Testing

Unit testing is when small pieces of a program are tested individually to see if they are functioning as planned. In our case we tested small functions such as ones that create randomized values for our weather information, and other small tests that ensure the proper formatting of our output is met. We will possibly use unit testing C# in .NET Core using the dotnet test and xUnit to create our solutions and tests, but we do not have extensive experience with those yet so we will most likely create our own unit tests, since we have a clear understanding of our project and its inner workings already. As previously mentioned, our unit testing will be extensive for the Website Subsystem as is essential the data is produced properly and in the intended format. Ensuring this will make our integration testing and interconnectivity between subsystems much easier. The Simulator and Reader Subsystems are unnecessary to provide unit tests for. The function of these systems do not face variability in input or procedure. The generation of data in the Simulator is generated based on boundaries defined by our team and are deterministic meaning we always know what values will be generated and the appearance of the data output by the Simulator. The Reader Subsystem handles a standard and straight-forward process when transmitting data over a comport. Because the Reader uses the RS-232 protocol, a standardized and simple procedure, we do not consider it necessary to unit test this process. We will break down the different unit tests for the Website Subsystem, and the lack of unit testing for the Simulator, Reader, and Database Subsystems.

5.1 Simulator Unit Testing

In introducing unit testing, our team has stated the unnecessity in unit testing the Simulator Subsystem code. Our team evaluated the change in states and the behavior in Program.cs and Entry.cs; we concluded the methods and state changes are hands off for the user therefore there is no benefit from creating unit tests. The data generated by the Simulator does not involve user input capable of impacting the operations or outcomes of this subsystem.

5.2 Reader Unit Testing

Similarly, the Reader Subsystem code also does not require unit testing. Our team evaluated the relevant files to this subsystem and determined that the objects and methods do not need unit tests created for them. The methods for sharing information over the com0com RS-232 protocol connection do not require testing to evaluate accurate data transmission, however this will be a necessary part to perform integration testing.

5.3 Database Unit Testing

Our Database Subsystem does not need unit testing. Our team assessed the Database Subsystem and determined that data transmission was our main concern with this subsystem so we will solely focus on integration testing of the Database Subsystem.

5.4 Website Unit Testing

The Website Subsystem involves all the frontend interaction the user experiences. Testing each individual component of the data access, views, and notifications is integral to guaranteeing a robust and dependable system. The List view does not provoke any opportunity for erroneous values, because it depends on the data in the database as well as the effectiveness of the data retrieval method. This view will however be important for larger, modularized testing.

The Data Manager is vital to the operations of many of the components. To retrieve data to be used in the Website Subsystem the GetDatabaseEntries() must be called. Specifically, the variant that takes an integer and two DateTime objects as arguments. The integer represents a site identification number which is necessary for identifying sites. The DateTime objects are used to specify a set of entries based on a range of time with the first DateTime object being the starting time and the second being the ending time. The DateTime value can be converted to from both a string or a long data type variable. The usage for this function requires the DateTime objects include the Year, Month, Day, Hour, Minute, and Second. The Year is in the range of 0001 to 9999. The Month is in the range 1 to 12. The Day is in the range 1 to the number of days in the month. This is dependent on what month as the total can vary. The Hours are in the range 0 to 23. The Minutes are in the range 0 to 59. The Seconds are in the range 0 to 59. This is the valid partition our unit test will test for. Values that are negative or 0 in the case of the Year are considered invalid as are values that are over the stated upper bound. The DateTime object as the argument protects the method from having to handle erroneous input such as integer or floating point types. The DateTime is distinct and structured in a specific way. The Id parameter may be any integer in the range of 0 to 2,147,483,647 because of the memory capacity of an int data type. The Id must be a valid number for a site in the database. For this unit test, sites will be used with known Id’s to accurately test this method. We will also attempt to use an Id that is not in the database to verify nonexistent data cannot successfully be retrieved. This will give us three partitions for the case of a valid Id and a fourth partition to test an invalid Id.

To update notifications, the method SetNotificationStatus() is available for use by an Admin. This method enables the display of select notifications for all user roles. The first parameter is the notificationId. This is an int data type that represents a distinct value for the notification message for reference. The second parameter is a boolean value. The given status will be inverted by the method then enabling or disabling the display of the notification in the notification list. The notificationId must be a valid Id for an existing notification in our database.

A unit test will be created to test a valid Id, a known Id found in the database table, as well as an invalid Id to verify a nonexistent reference is appropriately handled. This will be what consists of our two partitions. One with a valid Id and one with an invalid Id. It is unnecessary to test the boolean argument, because it is in no way capable of being erroneous.

In the Website Subsystem, the views we feature are of high importance. Our team will need to test user interactive components in our system to evaluate the accuracy. To accomplish these operations, a data manager object is used to handle data services and the accessing of data.

The Map view provides a geographical display with site markers signaling the location of SWAPR devices. When clicking a marker on the map a redirect occurs that loads the historical view page for that site. The method NavigateInNewTab() has a single parameter, an Entry object. The Entry object contains a site Id value used to determine the page to load. A unit test will be created to test that an entry object can successfully load the historical view page for the given site. One valid partition for the case where an existing site Id is referenced and an invalid partition for the case where a site Id is given for a nonexistent site. Due to the entry object only using the site Id, there is no concern for other erroneous valued attributes in the Entry object. Additional to the map site marker locations being visible, the color of the marker will be shown. The color is associated with the status of the site. The color is set by a status variable where it can be: Green, Orange, Yellow Orange, Yellow, or Red. To determine this status, a function GetColor() is used. This function has a single parameter that is an Entry object. The Entry provides a status attribute and is checked to determine and return the status color of the site. A unit test will be necessary to evaluate the case where a passed in Entry object with a valid status and the case where a passed in Entry object with an invalid status. If the status code is not one of the listed codes, then the color should not be set and the exception should be handled.

The Historical view features a line, bar, and radar as available graphics to visualize entries from the sites in the database. All these views utilize a number of operations to perform their functions. The method CreateDataSet() is used to take data from the database and transform it into a usable list for the website. It has four parameters: an int data type datasetType, string data type label, Color data type colorType, and a list of doubles data. The datasetType parameter is used to determine the type of graphical view whether it be line, bar, or radar. The label parameter is used to provide a display name on the graph. The colorType parameter specifies the color of the line or bar depending on the graph type. Finally, the data parameter is a list of the values for the points that will be placed on the graph. The label and colorType are considered arbitrary in the context of this unit test, because they do not impact the outcome of the graphs. The list of doubles is important; however this method only passes the data along for a later operation. The datasetType must be one of the three mentioned values otherwise it is invalid. This unit test will include four partitions with one partition handling an invalid datasetType value and the other three partitions handling the three valid datasetType values.

The three Historical view graphs also utilize a method CreateTimeDataSet(). This method takes the same parameters as the CreateDataSet() method excluding the list parameter data. Instead of a list of doubles, this method takes a list of TimePoint objects. The list contains data for the timestamp of each value placed in the graph. The same operations are completed for this method otherwise. A list of the data is returned with the set label and color.

5.5 Orchestra Unit Testing

The Orchestra Subsystem does not require unit testing due to its purpose. The Orchestra is a batch script performing a set sequence of operations to run the simulation of the SWAPR data and the reading of the data to the database. With no input or test cases to handle our team will prioritize testing this subsystem in integration testing.

For our integration testing, we are using a combination of the Hybrid and Big Bang methods, which will accurately assess the work we have done so far.

Hybrid integration testing is an approach used to test a system of modules by testing the communication between each connected module. The test is structured in three layers: the main layer, top layer, and bottom layer. The approach utilizes two other approaches, the top-down and bottom-up approach. The top layer represents the top-down approach, testing from the highest level and down in the system, and the bottom layer represents the bottom-up approach, testing from the lowest level and up in the system. The main layer is the central component for communication in the system. The goal of the hybrid integration approach is to test a connected and completed application's working system components in the early stages of development.

Big Bang Testing is an approach to integration testing in which all the components or modules are brought together simultaneously and then tested as a single unit. During testing, the integrated set of features will be treated as a single object. The integration procedure will not run unless all the components in the unit have been completed. This ensures that our system works as a whole and will be able to detect connection issues between the main features.

The Hybrid integration approach will be implemented to test the communication in our system from both sides of the database. For our usage of this approach, we consider the main layer to represent the database as it is central to the functionality of our application. The top layer represents the website, because it is the interface for the user to communicate with the database. The bottom layer represents the simulator and reader, because these two subsystems communicate with the database to provide it data. The database is at the center of our system because it is the storage and access point of all information used by the system. The database contains data tables for the notifications, site entries, and sites. These database tables are populated using generated data from the Simulator that has been sent by the Reader and received by the Amazon Web Services (AWS) Simple Queue Service (SQS) lambda functions. Our database schema defines these data tables for usage on the website. The website will utilize data retrieval functions to pull data from the database and generate pages.

The Simulator will begin by generating random numbers within valid and invalid ranges. The range values we will generate are four antenna power ranges, humidity, temperature, rainFall, wind speed and direction. The simulator will create data every 5 seconds to send the reader. In the functions GenerateValidData and GenerateInvalidData, the System.random class will be used. The System.IO.Ports.SerialPort class is used to send the data to the Reader after it has been generated. We'll use the System.IO.Ports.SerialPort class in the Reader software to accept data from the virtual com port. When called on the SerialPort object, the function Open() establishes a connection to the Simulator. The data will be sent to the Reader where the Reader will serialize the data into a JSON string. The string will be printed to the console to check for the correct values and serialization happens. We will compare the generated simulator data to the json strings for accurate code

The Reader will then send the data to be used in AWS. Using the AWS.SQS C# class, the Reader will send the message to the Amazon Web Services (AWS) Message Queuing Service (SQS). AWS SQS will then store the data in a queue. Once in the queue, AWS lambda functions will be alerted of new entries, pull them, and utilize them to generate SQL queries to add rows to the Database's Entry and Notification tables. The SQS Lambda function is expected to take and deserialize a JSON string into an Entry object. The Entry object provides the site identification number with all of the weather and power information at the given timestamp. This information will then be evaluated for its status and a Notification will be generated if the status is anything other than green. To verify the outcome, we must input an established JSON string into the Lambda function and evaluate the database tables to confirm the new entries are identical to the expected outcome. The test harness in this module includes creation of database entries input as a JSON string. By verifying input into the Lambda function produces distinct entries in the database we can assert that data placed in the AWS message queue will be accurately stored in the database.

With a database populated with site, entries, and notifications, the website is then integrated to communicate with the database. The Website Subsystem features the website and the multiple graphical user interface elements. The Historical, List and Map Views, and Notification components are tested and evaluated using pulled entries from the database. The creation of a notification, as we previously overviewed, is completed using the AWS SQS Lambda functions and message queue. The data loaded into the database is reviewed and established by our team for the purposes of this integration testing. The notifications are stored in a notification datatable and are accessed from website functionality. It is necessary to verify all notifications pulled from the database are identical to the data stored in the database. The test harness includes the created notifications and entries stored within the database notification and entry datatable. This is the same case for the Historical View. The Historical View pulls the entries from the entries datatable.

The website uses the various weather and power fields from a range of entry dates and visualizes these in either a line, bar, or radar graph. The chart will have multiple tests to ensure proper display of these graphics. In this integration testing, viewing these graphs through the website pages will be done to evaluate the accessing and implementation of the entry data. The data, like the notification testing, will be previewed to certify our team knows the expected generated graphs. The generation of these graphs is specified by various parameters defined by a user. This means the Historical View must be carefully evaluated to guarantee the user entering graph specifics can produce the desired graph. With accurate graphs, we have verification that our website is able to pull data from the database correctly and use this data to generate graphs. For the test harness, the Historical View requires a user to select the parameters of the graph which adds one extra step of testing. The test harness also includes the created entries stored within the database entry datatable which must be retrieved using the input parameters. In addition to the Historical View, the List view must also accomplish a similar task.

The List View does not utilize user interaction like the Historical View, but instead implicitly accesses the database to retrieve the latest entry from all sites. The List view will be tested to determine if the access and retrieval of data from the entries works appropriately while also verifying every available site and its latest entry is provided in the form of an informative panel. To ensure this, as we will do with the other views, the entry data in the datatable will be known and verified in the website List View page to ensure the known sites are visible with accurate entry information. The test harness for the List View includes the latest created entries stored within the entry datatable from each site. Finally, the Map View will be tested in a similar fashion to the List View with an additional check.

The Map View accesses and retrieves site and entry data from the datatable, but only a certain amount of information is relevant to this view. The purpose of the Map View is to display the sites on a map of the United States based on the declared latitude and longitude of the physical site in the real-world. In addition to showing the location of the sites, a color system is used based on the status of the site. This status is determined at another location of the system, but the importance here is to give the user a clear understanding of the operational state of the site. The database site data retrieved will need to be tested to verify they utilize the latitude and longitude data to accurately place the site marker on the map. To color the marker the accurate operational status color, we will need the status of the entry being evaluated which should be the latest entry from the site. The Map View will need to provide one additional feature, interactivity from the user through the website GUI, that must be tested. By clicking on a marker the site must use the site identification number to determine the URL path to send the user to. This page the user is redirected to must be a page of the Historical View already set to the site Id and ready to generate a desired graph. Verifying this will be important to not only ensure the Map View displays accurate information from the database, but also guarantee the click of a site marker on the map redirects the user to the appropriate Historical View page. The test harness includes the stored sites within the database site datatable. By verifying these entries utilized throughout the website are accurately accessed and pulled to the website pages, our team can confirm the website retrieves correct data from the database in both the case of the site datatable as well as the entry datatable. This is integral for our system, because the primary purpose of the website is to provide methods for viewing information from the database.

Our project's Orchestra Subsystem will be used to test the system as a whole. This subsystem enhances the simulator and reader subsystems' capabilities by allowing multiple instances of each to be generated. Put another way, the Simulator Subsystem and Reader Subsystem pair represents a single SWAPR device. A simulator and reader pair will connect to the database using the reader software and send the SWAPR device data generated by the simulator to AWS. We can scale this up to hundreds of SWAPR devices using the Orchestra. For a full system test we will use the Orchestra to create 250 RFF Sites using the Simulator Subsystem and Reader Subsystem pair to populate the data for each SWAPR device. The pair will be continually running as the rest of the process runs; over time, we will be testing the data generation and storage part of our system to ensure that it can handle the heavy load of putting data in the database while also being able to give data to the website. Next we will discuss how we will implement usability testing with our system.

Now that we have explained our integration testing, we will explain usability testing and our plans for carrying out our tests. Usability testing is the testing of the experience of an end-user with the software from an outside perspective. The hypothetical end-user has no inside knowledge of how code works other than functionality built into the user interface. The goals of usability testing are to ensure that the integrated systems are functioning correctly from the end users standpoint and to iron out any bugs that occurred during the integration of our system. Usability testing is done by having a hypothetical user carry out a number of operations to ensure that the functionality that our system should provide is being carried out in the correct and appropriate way.

For our usability testing, we envision two different usability tests from two different users. One user will be a standard user, and the other user will act as an administrator. This will be done because our system has two main roles which are the user and administrator which have very slight differences in the permissions they are given. This will in turn slightly affect the way that functionality is provided to the two roles. The main difference is that the administrator has access to the user and notification management page where they can manage the users and notifications in the system. Besides this difference, the administrator and user roles are practically the same and can do all other functionality on the system. We will split the testing into two parts, one with the user role and the other with the administrator role. The testing will be done on a visual confirmation basis where we do an action and confirm that the result is correct through manual checks. For example, we will open the list view and confirm that the data being shown is done in the correct manner with only the latest entry being used from each site. We will manually confirm this by having manually entered data for each test that way we can be sure that the results are correct. This will allow us to have more control over the results making it easier to confirm proper functionality. Once we have confirmed that the testing is done correctly with static data then we will simulate the network and confirm that nothing weird occurred. If nothing weird occurs then we know that the system functions correctly.

To test the system, a member of our team first registers an account and login as that new user. They will then access each page of our website. This includes the list view, map view, historical view, and notifications. When the user accesses the historical view, they will also test the csv exporting functionality by downloading the data used to create the historical view. The user will then attempt to access the two restricted pages of the website being the admin account management and admin notification management pages. Both pages should fail to load for the user with a message saying that they need to be administrators to view the page.

A second team member acts as an administrator. We have preconfigured a username and password in our code and they will login with the admin username and password. Then they will access the list view, map view, historical view, and notification pages. They will test to make sure they can see each of the graphs properly and that data exporting works properly on the historical view. To test the administrative abilities, our team member posing as an administrator will access the admin account management page and will be able to view all existing accounts and modify them. They will test deletion by deleting an existing non-administrator account. They will also access the admin notification management page where they will be able to view all notifications and disable notifications and ensure they are correctly disabled by viewing the change in the database and admin notification management page.

We plan on doing the usability testing over a one week period that will happen during the week of UGRADS which starts April 18th. By doing the testing at this time, we will know what we can show to the general public while displaying our project as well as making our final list of tasks to complete by the end of the semester.

6. Project Timeline

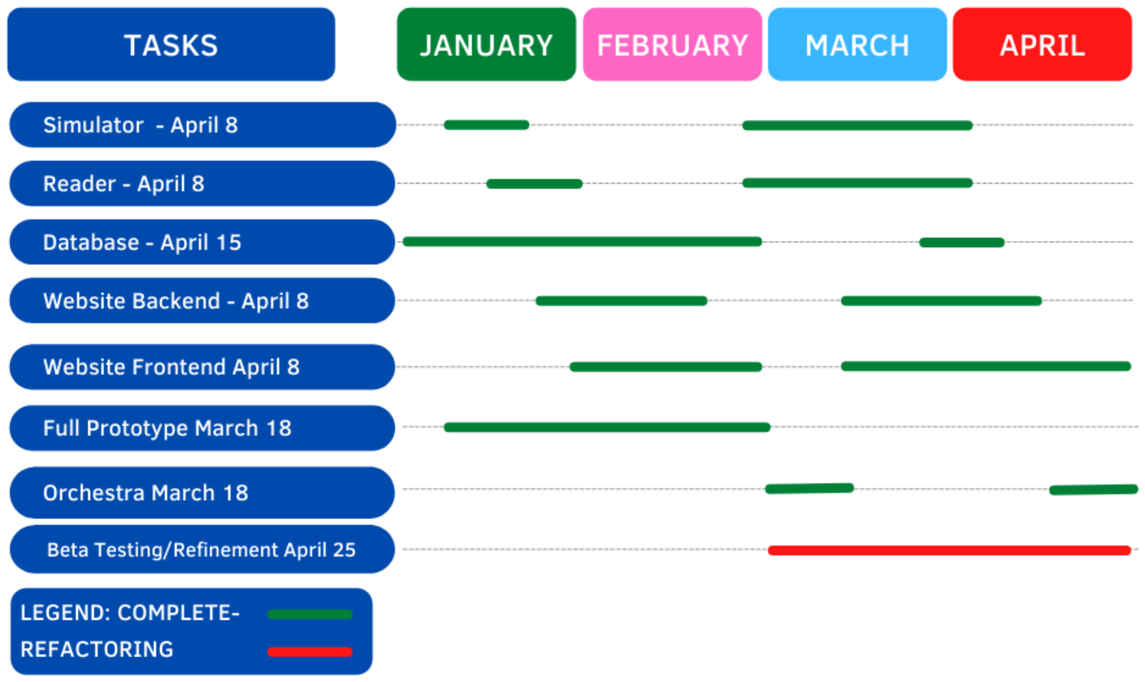

Figure 6.A: Gantt chart showing final semester project timeline

Our project has met the functional and performance requirements set by our client in the timeline shown by the gantt chart below. This chart shows that the initial completion of our Simulator and Reader occurred by the end of January. These two subsystems would be modified later with further updates to fit into our system in March. The database subsystem was updated during the majority of our development process. By march our database was a majority completed with some minor changes in early April. Our team had an issue with permissions which prevented our system from being fully implemented with a database hosted on remote Amazon Web Services. The website subsystem is the main component of our system because it involves all user interaction and displaying of SWAPR data and for this reason it has been updated all throughout our timeline. The full prototype was completed at the very start of March. This version of our system met all of the minimum viable product features and represents an important benchmark in the development of our system. The orchestra subsystem is a feature that was not priority, but set as optional by our client if we completed our full prototype. Because we completed our prototype, we implemented the orchestra to allow for multiple instances of our simulated SWAPR device and reader software. Following this prototype and orchestra implementation our team put all of our focus on refactoring and refining our system. This includes cleaning up the code base structure for readability, adding comments, fixing any bugs, and any other steps to produce clean and stable code. The end of our timeline completes with finalizing our project for the updated, tested, and refined prototype by the end of April or start of May.

7. Future Work

If we were to improve on our final product in Version 2.0, we would love it if General Dynamics Mission System installed SWAPRs at each RFF station in the Rescue21 system. This would allow us to test our reader subsystem, sending the real-world data to the database once it can be sent and analyzed with few problems on the website. Next, we would integrate our project into GDMS' architecture by replacing our Identity Framework process with GDMS' Active Directory service. Active Directory was too expensive for our budget therefore we did not use it. We would use GDMS' mapping software to improve our map view, which would allow our client to save time training employees on new software. The last thing we would do is migrate our AWS hosting to GDMS' internal hosting services. Doing this will allow for maximum privacy and control on their closed network.

8. Conclusion

In conclusion, GDMS wants some additional features to help them maintain the Rescue21 system. There are two main features that they need: the ability to record the power levels of the antennas at an RFF and the ability to record the weather information at the RFF sites. Knowing this information will help GDMS with determining the cause of RF interference. Knowing the cause of RF interference will help GDMS determine if they need to send a technician to an RFF site or not. The other reason that the weather information is helpful is when predicting damage. Knowing if there is a severe storm at an RFF site will help GDMS predict equipment damage before it occurs so they can schedule an engineer to visit the site before the outage starts. This will help reduce outage times, ensuring that the USCG can always help those in need. Knowing the weather also helps with scheduling maintenance on an RFF site. It is risky for GDMS to send someone to climb the antenna towers in the middle of high winds or storms. Knowing the weather will help GDMS reduce the number of times they send a technician to a site when they cannot perform maintenance. All of this will help GDMS save money, reduce outage times, and potentially save lives.

To create the additional features that GDMS has requested, we are building a secure web application for registering, configuring, and managing the SWAPR network, and displaying output in a clear graphical interface.

We are working on refactoring the code and converting it to connect with the other subsystems. We are slightly behind on our alpha product but we believe that we will be able to catch back up by spring break. As of right now, we are on track to provide a beta version product to General Dynamics by the end of April.

9. Glossary

Active Directory - It runs on Windows Server and enables administrators to manage permissions and access to network resources. Active Directory stores data as objects. An object is a single element, such as a user, group, application or device such as a printer.

AOR - Area of Responsibility

AWS - Amazon Web Services

Blazor - Blazor lets you build interactive web UIs using C# instead of JavaScript. Blazor apps are composed of reusable web UI components implemented using C#, HTML, and CSS. Both client and server code is written in C#, allowing you to share code and libraries.

C# - is an object-oriented, component-oriented programming language. C# provides language constructs to directly support these concepts, making C# a natural language in which to create and use software components.

Com0Com - is a kernel-mode virtual serial port driver for Windows. You can create an unlimited number of virtual COM port pairs and use any pair to connect one COM port based application to another. The HUB for communications (hub4com) allows to receive data and signals from one COM or TCP port, modify and send it to a number of other COM or TCP ports and vice versa.

CONUS - Continental US

GDMS - General Dynamics Mission Systems

GUI - Graphical User Interface

IDE - Integrated Development Environment

OS - Windows Operating System

Rescue21 - control and direction-finding communications system, was created to better locate mariners in distress and save lives and property at sea and on navigable rivers. By harnessing state-of-the-market technology, Rescue 21 enables the Coast Guard to execute its search and rescue missions with greater agility and efficiency.

RF - Radio Frequency

RFF - Remote Fixed Facilities

RTT - Round-Trip Time

SQS - Simple Queue Service

SWAPR - Site Weather and Power Recorder

URL - Uniform Resource Locator

USCG - United States Coast Guard

10. Appendix A: Development Environment and Toolchain

10.1 Hardware

Our team developed in a Windows environment by request of our client. The components of our team's machines varied between low-end and high-end. Any mid-level CPU with at least 32 Gb of RAM is necessary to develop our project because of the demand from Visual Studio.

10.2 Toolchain

Our team developed our project in Visual Studio, a product by Microsoft, and used multiple platforms to aid in development. Visual studio is a requirement by our client and necessary for building this web application. Visual Studio also allows integration with many of the other essential tools. In Visual Studio, our solution was created with a Blazor project to set up the basis for our web application development. The critical platform used in our project is .NET 5. This package includes many class libraries, application programming interfaces, and tools. These packages are necessary for creating objects for the front end of our application and data management in the backend. Using the NuGet package manager, our project integrated all of the packages necessary for our application. The Blazor Framework, ASP.NET Core, Identity Framework, and Chart.js are four special packages used with the .NET 5 platform used for our website's needed functionality. Blazor gives us the framework for creating the website, and ASP.NET Core is necessary for creating Razor pages for our website. In addition to these packages, we enabled the use of the Identity framework in our Blazor project for including roles and authentication in our prototype. Our client requested an example case of a user login window and user account authentication. The JavaScript package, Chart.js, is used in our application for creating graphics using retrieved data from our database. Chart.js is used for creating charts to visualize the data produced by the SWAPR devices in our system. The MySQL database system is used to make our database hosted on an Amazon Web Services server for remote storage. Database access through a cloud service was requested by our client to provide a remotely accessible database system and storage location.

10.3 Setup

What follows is a step-by-step guide to setting up a working environment for development of this project. First gain access to a Windows environment, Windows 10 or later. Install the Visual Studio integrated development environment. In Visual Studio, create a Blazor project and connect your GitHub account to allow for cloning the project repository. Once you clone the remote repository for the project, you will need to verify all of the correct dependencies have been installed and with a compatible version. By checking the NuGet package manager, there should be: Aspose.Imaging, version 22.3.0; Aspose.SVG, version 22.2.0; Blazor.Extensions.Canvas, version 1.1.1; ChartJs.Blazor.Fork, version 2.0.2; EPPlus, version 5.8.5; Microsoft.AspNet.Mvc, version 5.2.8; Microsoft.AspNetCore.Diagnostics.EntityFrameworkCore, version 5.0.12; Microsoft.AspNetCore.Identity.EntityFrameworkCore, version 5.0.12; Microsoft.AspNetCore.Identity.UI, version 5.0.12; Microsoft.AspNetCore.Mvc.Core, version 2.2.5; Microsoft.AspNetCore.Mvc.RazorPages, version 2.2.5; Microsoft.EntityFrameworkCore.Tools, version 5.0.15; Microsoft.Extensions.Configuration.Binder, version 5.0.0; Microsoft.Extensions.Configuration.Json, version 5.0.0; Microsoft.VisualStudio.Web.CodeGeneration.Design, version 5.0.2; MySql.EntityFrameworkCore, version 5.0.10; Once you have verified these packages are installed, verify the AWS hosted database is active. Once the correct packages are installed and the database is accessible, select the solution object in the explorer in Visual Studio and unload then reload the solution with dependencies. Once the solution is reloaded, the application is ready to execute and launch the website.

10.4 Production Cycle

With a ready environment to run and maintain the project, the production cycle can now be explained to walk through the steps to developing the project further. The general overview of the steps to this is to first create a working branch off of the develop branch, make the desired edits to the code base, then push the changes to the working branch. You can push these changes to the development branch with an updated working branch. Now that the development branch has the newly committed changes, it can be tested to verify the web application works as expected. When ready, the development branch can be merged with the main branch. This completes the overview of steps to making edits. As a specific example, let's propose the developer needs to add another Razor page to the project. They can start by creating a new branch of the developed branch with an appropriate and related name. The developer will then create a new Razor page by right-clicking the page's directory in the project and selecting to add a new Razor page. The necessary HTML and C# code can be added to a new page. It will be likely that the newly added code will require new classes or functions to be created. This will likely need to be stored in the DataManager.cs. Once these changes have been made and tested on the working branch, the developer can push the changes to the remote working branch, merge the working branch with developing, and merge the develop branch with main. Note that it is crucial to test the develop branch if other changes have been made in the process of this new Razor page being added. This is to avoid conflicts and bugs that may arise from multiple developers updating the codebase at once. Another note is to be mindful of the automatically generated dynamic link library files created when running the project. The dynamic link library files are specific to the developed system and should constantly be recreated with a reload and run in Visual Studio.