S.A.R.C.I

Search And Rescue Coastal Intelligence

Software Design Document Final Draft

2/11/2022

Version 1

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) is responsible for 296,000 nautical miles of sea, and this is a significant job that isn't done alone. Our client, General Dynamics Mission Systems (GDMS), is responsible for working with the USCG on the Rescue21 system for assistance with search and rescue (SAR) missions. Whenever there is a problem on water (such as being stranded or experiencing an engine fire etc.), and you need assistance, the USCG is the one to pick up the call, but how does the USCG even begin searching about 296,000 nautical miles of sea? The answer lies in the Rescue21 system, which is a system of antenna towers, called remote fixed facilities (RFF), that are placed strategically off the coast of the Continental US (CONUS), major rivers, lakes, and islands (Hawaii, Puerto Rico, and Guam). Each RFF site is equipped with directional finding very high-frequency radios that are used to pick up distress signals, the direction of the signal, the strength of the signal, and whether it is a hoax call. This information can be used to dramatically reduce the area that needs to be searched, saving time, money, and increasing the likelihood of those in distress being rescued.

GDMS is content with the Rescue21 system; however, they want some additional features to help them maintain their 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. RF interference causes static while communicating over the VHF signal, and a few different factors can cause it. One of those factors is that the radio equipment is broken and needs to be replaced. The other aspect is rain. Rain is conductive, so if there is rain in the air, then the signal will be absorbed by the rain, and the signal will fade over distance. 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.

GDMS contacted Northern Arizona University (NAU) to build the device to record the antenna power and weather data last year. The project went very well as they created a site weather and power recorder (SWAPR), which worked properly. The only shortcomings to the SWAPR device were that the data being given out is not human readable. That is why GDMS reached out to NAU again this year to build the software for the SWAPR network.

GDMS envisions a secure web application for registering, configuring, and managing the SWAPR network, and displaying output in a clear graphical interface. Through bi-weekly

meetings with our client, together we have created a list of key user-level requirements. A key user-level requirement is a functionality that a user can use in the final product. The list of key user-level requirements that was developed is given below:

1. Create data imitating the SWAPR device's output
2. Take data from a SWAPR device and send it securely off-site
3. Store and serve SWAPR data in a secure manner
4. Establish a secure website environment with authentication
5. Create a list view summary of the SWAPR devices in the network
6. Create a map view summary of the SWAPR devices in the network
7. Create graphs using historical SWAPR data
8. Create a way to notify operator when there is a problem with a RFF site
9. Export data from the database as a .csv file
10. Emulate the entire SWAPR device network for stress testing

Using the key user-level requirements and with the help of our client, we have designed a five-part system that can meet all user-level requirements. The project will be broken into the Simulator, Reader, Database, Website, and Orchestra subsystems. Each subsystem has its own functional requirements. Functional requirements are functionality that will be required for the user-level requirements to work. Listed below are the functional requirements split up for each subsystem:

- Simulator
    - Randomly generate SWAPR data that is valid or invalid
    - Send data over virtualized com port to Reader
- Reader
    - Read data over virtualized com port from Simulator
    - Send data to AWS message queuing service
- Database
    - Collect SWAPR data in AWS message queuing service
    - Create database entries using messages in queue

    - Serve data to Website
- Website
    - Secure authentication with accounts and role-based permissions
    - Ability to query database
    - Ability to export data to .csv
    - Ability to notify admin users of improperly functioning SWAPR's
    - Create a list-view summary of the latest entry from each SWAPR
    - Create a map-view summary of the latest entry from each SWAPR
    - Create graphs from historical data from one SWAPR
- Orchestra
    - Create any number of Simulator Reader pairs over virtualized com ports

Along with the functional requirements for each subsystem, the entire system has several environmental constraints which are constraints placed by our client to ensure the solution will work within their infrastructure. Our client has constrained the project to be a Windows based .NET 5 Blazor Server website written in C# using Visual Studios. These constraints are to ensure that when we deliver the final product to our client, they will be able to implement, modify, and maintain the project into the production environment of the Rescue21 system.

2. Implementation Overview

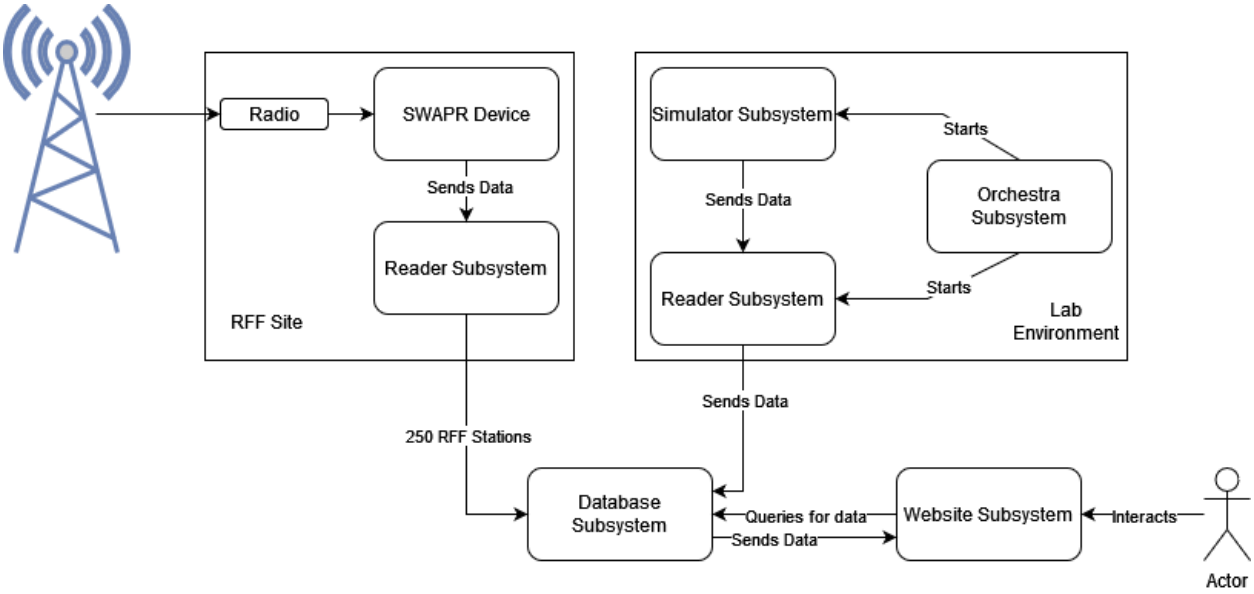

Figure 1: Overview of distributed system subsystems

To expand upon these requirements, technologies, and constraints mentioned previously, we will describe in greater detail our approach to implementation and how we will integrate everything into one cohesive system. Our solution will be a secure web application for registering, configuring, and managing the SWAPR network, and displaying output in a clear graphical interface. As shown in Figure 1, we envision our project broken up into five different subsystems, which all work together to create a solution for GDMS. Those five subsystems are called the Simulator, Reader, Database, Website, and Orchestra.

The Simulator will start by randomizing numerical values within valid or invalid ranges. Because we are using C# as our language for this project, we will use the System.random class under the system namespace. The System.random class will be used inside the functions GenerateValidData and GenerateInvalidData. After the data is generated it is sent to the Reader using the System.IO.Ports.SerialPort class. We will be able to set the port name and connect to the other com port by calling the Open() function on the SerialPort object.

For the Reader software to receive the data from the virtual com port, we will be using System.IO.Ports.SerialPort class. The function Open() establishes a connection to the Simulator when called on the SerialPort object. The Reader will send the data over a TCP connection to the Database using the System.Net.Sockets.TCPClient class. The database will then be responsible for creating an entry from the data provided.

The Database will receive the data in a queuing service provided by Amazon Web Services (AWS) called AWS Message Queuing Service (SQS). Once in the queue, AWS Lambda functions

will be used to create SQL queries to create entries in the Database. The Database will be made using MySQL and be hosted by AWS Relational Database Services (RDS).

The Website Subsystem is split into two parts, the backend and frontend. The backend will provide four main functionalities: Authentication, user-roles, and accounts, Database queries, Notifications, and .csv data exporting. The Authentication, user-roles, and accounts will all be managed by the C# Identity class. The Database Queries will be handled by the MySQL C# classes and DbContext class. The Notifications will be custom made and displayed using C# templates and custom C# methods. The .csv data exporting will be done using the EPPlus, C# models, and the IActionResult class. The frontend will have three different views being the list view, map view, and historical view. The list view will be created using a C# wrapper class of Canvas.js. The map view will be created using the functionality provided by SyncFusion and C# NavigationManager class. The historical view will be created using the C# wrapper class of Chart.js.

The Orchestra Subsystem will be responsible for simulating a RFF site by connecting an instance of the Simulator and Reader subsystem over a virtualized com port. The Orchestra Subsystem will use the com0com software to create the virtualized com port. The Orchestra will be a script written in C# if possible or Windows Bash if not possible. The script will first create the virtualized com port and then pass the ports to communicate with to the Simulator and Reader on creation.

3. Architectural Overview

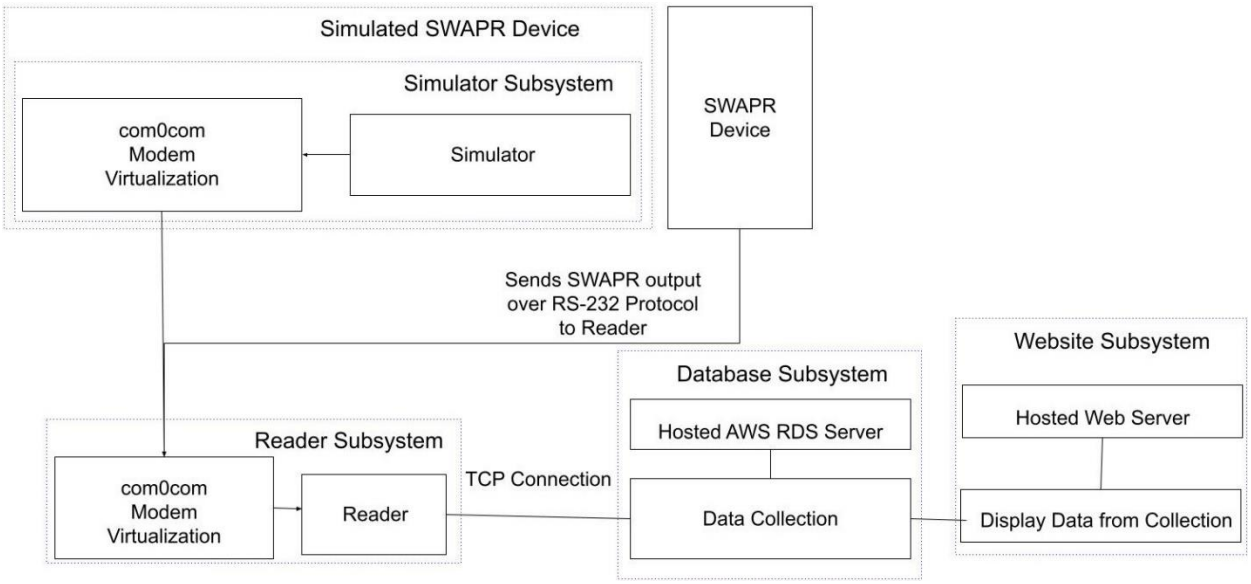

Figure 2: Overview architecture of system subsystems

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 3.

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.

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. You can find a diagram showing the components of this subsystem in Figure 5.

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. You can find a diagram showing the components of this subsystem in Figure 6.

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. You can find a diagram showing the components of this subsystem in Figure 7.

4. Module and Interface Descriptions

In the previous section, our team established an overview of our architecture and its five subsystems. We can now describe each subsystem in more detail using UML diagrams, a short natural description of responsibility, an outline of the user interface when applicable, and a description of how the subsystem will operate. We will discuss the subsystems in the order they were introduced.

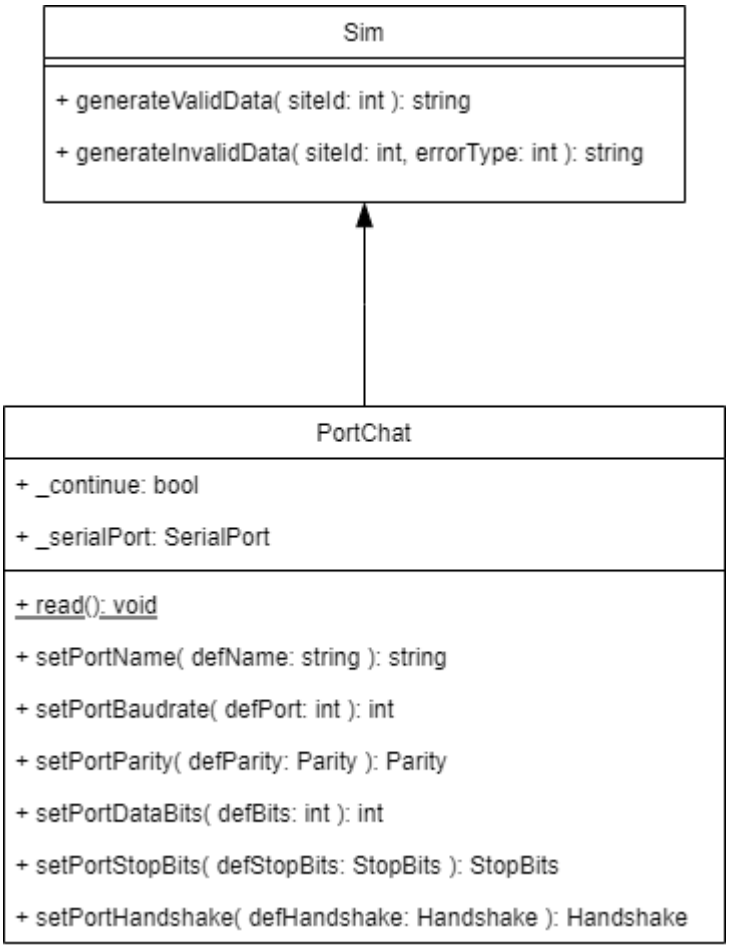

Figure 8: UML class diagram of Simulator Subsystem

The first subsystem of our system is called the Simulator. The Simulator is responsible for emulating the previous year's capstone project, The SWAPR, by generating valid or invalid weather and power data. This data is then sent over a virtual com port that is emulating the RS-232 protocol. This is done because the SWAPR device will communicate with the reader over the RS-232 protocol. The Simulator subsystem is the data core of the system meaning that everything starts with the Simulator generating SWAPR data. This data is then sent to the Reader subsystem for communicating that data in a way that eventually allows the rest of the system to access that data.

The Simulator will be managed through scripts and config files to make it easier for GDMS to start or stop the subsystem when doing stress testing in a lab environment. Since it will be controlled by a script, it will not need an interface besides a console window. The script will pass the id of the RFF site and the com port to communicate on. Once the Simulator knows this information, it can begin generating SWAPR data. The Simulator will run in an infinite loop that will call the generateValidData(int) or generateInvalidData(int, int) every five seconds. We choose five seconds because this is the interval that the original SWAPR device generates data. These methods will return a string that is in the form of a list that can then be used by the Reader subsystem to create a database entry. When we get the string back from the function calls, we will do the last step of the Simulator's loop execution which is sending the string to the reader over the virtual com port. This is done by calling the WriteLine() method of the Serial Port instance. When we receive the 'quit', 'q', or Ctrl+C in the console or an exit command from a script then we will break the loop, join the reading thread, and close the serial port instance effectively ending the Simulator subsystem.

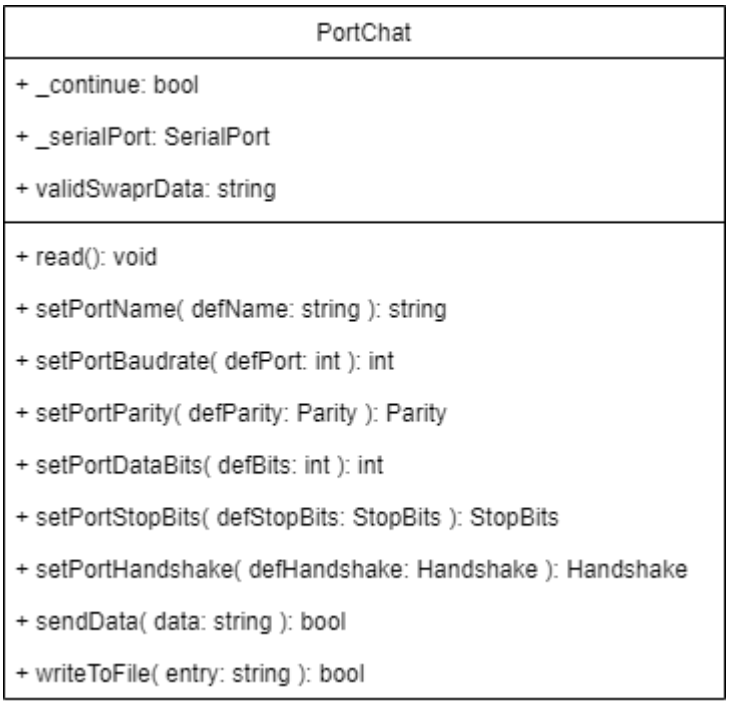

Figure 9: UML class diagram of Reader Subsystem

The second subsystem of the system is called the Reader. The Reader is responsible for reading data over a com port from the Simulator or SWAPR using the RS-232 protocol. That com port can be virtually created or physically created depending on if it is reading data from the Simulator or the SWAPR device. The Reader will then be responsible for sending the data to a queue hosted on AWS that will create database entries from the string received. This will happen every five seconds until the Simulator or SWAPR device stops sending data. The Reader can be considered as an extension of the Simulator as the Reader will always be paired with the Simulator

or SWAPR device. As such, the Reader can be considered as the data core of the system meaning that the system starts and depends on the creation of database entries that the Reader is responsible for doing.

This subsystem is very similar to the Simulator especially when it comes to the public interface which will be as minimal as a config file and console input. This is because the Reader only needs to know the com port on which to communicate and the connection information for the queue. The Reader will be used in the production version of the project using a script that can start or stop the instance as needed. For those reasons, an extensive public interface is not needed.

Once the Reader starts up, it will get the information it needs from the config file and then it will set up the com port. The Reader will then jump into an infinite loop that will read data over the com port and using the sendData(string) method it will send the data to the queue to be created into database entries. The sendData(string) method will return a boolean value that equates to the success status of the sending operation. If the sending operation is true, then it will continue like nothing happened otherwise it will try again up to three times and if it continues to fail then it will print out an error message and start writing the SWAPR entries to a data file instead. It will write to the output file using the writeToFile(string) method and it will continue to write to the file until the program is restarted or doesn't receive SWAPR entries anymore. The loop will be repeated until the Reader has not received data for some timeout period (~1-2 minutes) which if this happens then the Reader will create one more database entry that can be used to show that the RFF site is no longer going to be transmitting SWAPR data. If there is an error in connecting to the database, then the error will be printed to the file. Should the timeout period be passed, then the program will break the infinite loop and begin the cleanup procedure. The only other way to stop the program is through console inputs such as 'quit', 'q', or Ctrl+C.

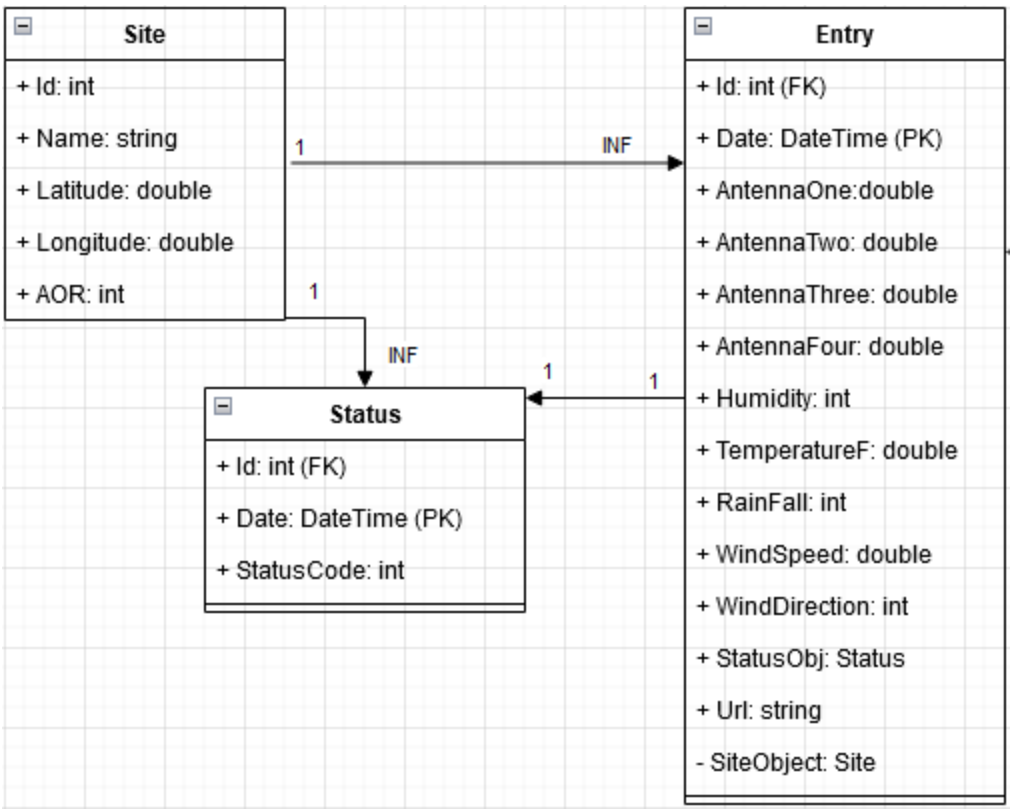

Figure 10: Schema for Database

The third subsystem of our system is the Database. The Database will be responsible for storing and serving SWAPR data and account information to the website to support the Website's functionality. The Database will have two main queries executed by the Website which will retrieve the latest entry from each site or all entries from one site over some time range. The only time that the Database creates entries will be when the Reader sends data to the queue. The Database will be the backbone of the project as it connects the Simulator Reader pair with the Website or the SWAPR Reader pair with the Website by gathering and passing data from Reader to Website.

The Database will be made up of three tables that are exclusively for storing SWAPR data. The template for the tables can be found in Figure 10 under the classes named Site, Entry, and Status. The Site uses it's ID as a primary key (PK), the Entry uses a composite key made of the foreign key (FK) of a site id and the PK as a DateTime, and the Status uses a composite key made of the FK of a site id and the PK as a DateTime. The Database will also have tables for the accounts that are authorized to use the website. The table for this operation can also be found in Figure 10 under the class name Account. These tables are named exactly as seen in Figure 10 and will be stored on Amazon Web Services (AWS) using the relational database service (RDS). As stated in the Reader, all the data collected will be stored in a queue also provided by AWS by a service called Message Queuing Service and Lambda Functions. The data will be collected in the Message Queuing Service and the database entries will be made using the lambda functions.

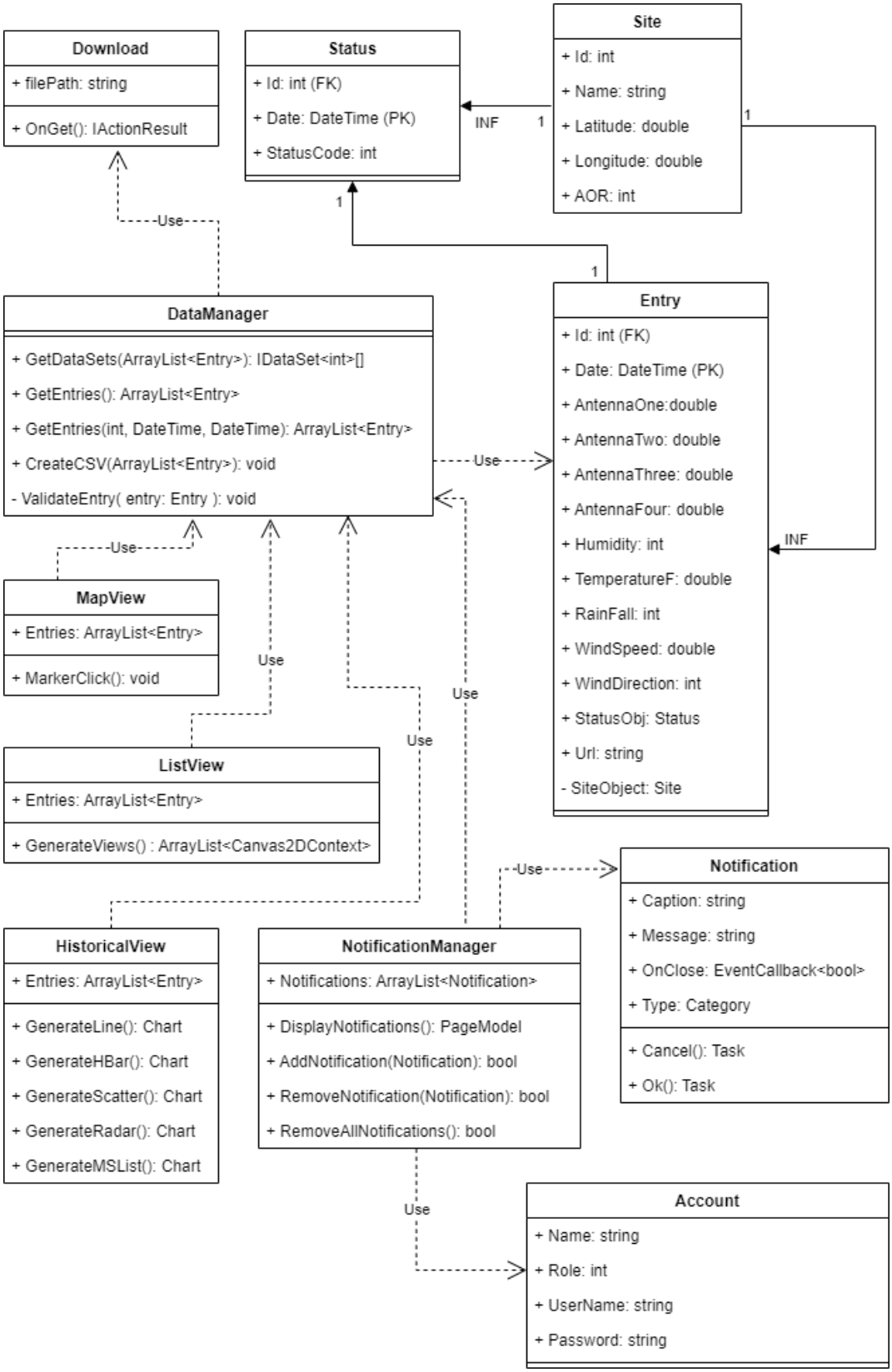

Figure 11: UML class diagram of Website Subsystem

The fourth subsystem of our system is the Website. The Website will be responsible for being the user interface for our client to access all the collected information from the Simulator or SWAPR devices. The Website will provide several views which are called the list view, map view, and historical view. These views will be supported by four backend subsystems called Authentication, DataManager, Notification, and Download. We will first review the functionality of the backend subsystems and then the views.

The backend subsystems are the workhorse of the website because they will provide all the functionality that the views need to work properly. The first subsystem we will discuss is the Authentication subsystem which is responsible for implementing the user role and account functionality. This means that it will deal with registering, sign in or out, and permissions while viewing the Website. The Authentication subsystem is not in just one class which is why it is not shown in Figure 11. The Authentication subsystem is stretched out across the entire website starting in the startup files, showing elements in nav menus, defining access rights to web pages, and providing all the functionality for database storage of account and role information and retrieval of account and role information. The Authentication subsystem is using the C# Identity class which will take care of everything that was mentioned above with very little work from us. The only modification to the Identity functionality is where the database tables are stored. By default, they are stored on a local database connection that is created for the user but will be changed so that the tables are stored on the AWS RDS along with the SWAPR entries. It is planned to also create functionality that will limit the number of SWAPR devices that user roles can view to their area of responsibility (AOR) which is a group of 18 SWAPR devices in a specific geographic region. We chose 18 SWAPR devices because our client told us that this was the average number of RFF sites that employees will be responsible for. The admin users will not have this restriction and will be able to view all SWAPR devices in the network.

The next backend subsystem that we will discuss is the DataManager. The DataManager is responsible for handling all communication between the Website and the Database. The DataManager will be responsible for making one of the two queries to the Database, calculating entry statuses, and creating datasets or CSV files. The DataManager will also call the NotificationManager whenever the status of an entry is anything other than green. The DataManager will have two calls that retrieve data from the Database both called getEntries(). One will take no parameters and will grab the latest entry from every SWAPR device in the network and the other will take in a site ID, start date, and end date and it will grab all the data from that site between the two dates. Both functions will return ArrayLists containing the relevant entries. After the DataManager gets the entries from the Database, the DataManager will loop the entries, find any entry that doesn't have a status, and call the ValidateEntry(Entry) function. The ValidateEntry(Entry) function will then evaluate the SWAPR data and assign a status to the entry. If the status is anything other than green, then the function will create a notification and call the NotificationManager's addNotification function to make sure that an alert is shown to the next admin that signs in or to any admins already signed in. The DataManager will have two data

manipulation methods called GetDataSets() and CreateCSV(). GetDataSets() is used by the historical view and it is responsible for taking in a list of entries and returning a list of datasets that each contains only one column of information. So, for example, it would return one list for the humidity recordings, one list for the temperatures recorded, one list for the wind speeds, etc. The CreateCSV() function will be responsible for taking in a list of entries and it will not return anything. Inside the function, it will use the list of entries to extract the individual members of an entry and put it into a .csv file and it will then be put into the wwwroot/csv folder on the Website. Once the .csv file is created, it can be downloaded by the client and deleted.

Next are the Notification and NotificationManager subsystems. The NotificationManager will be responsible for creating, storing, displaying, and deleting notifications for Admins on the website. Creation of a notification will be done in the DataManager when validating entries. If a Notification needs to be created, then the DataManager will call the NotificationManager and provide the Notification to be shown. Once it is added to the NotificationManager then it will be stored in an array of other Notifications to be shown to the admin users. When an admin logs in, they will be able to check the Notifications and delete them as needed. The NotificationManager will use the AddNotification(Notification) function to add notifications to the list to be displayed. When an admin user wishes to view the Notifications, the DisplayNotifications() function will be called and it will return the page model that is to be displayed. When the admin user wishes to delete notifications, they will call one of two methods. They will call the RemoveNotification(Notification) function when only one Notification is to be removed and they will call the RemoveAllNotifications() function when all of the notifications need to be deleted.

The last backend subsystem discussed is the Download class which in cooperation with the DataManager's CreateCSV() function provides the functionality for creating and downloading csv files that contain the historical data used to create the historical view. Since this will use historical data, the functionality to create and download the csv will be on the historical view page. The Download class will be responsible for being the endpoint by which the user downloads the csv file. The Download class will take in a file path that will be used to show what file to send to the client to download. When the user goes to the download page they will invoke the OnGet() function which will return an IActionResult which will be what prompts the user about the download.

The backend subsystems are going to be the workhorse behind the scenes for the frontend views. The frontend views will be how the client, GDMS, interacts with the website to view the collected SWAPR data in a nice graphical presentation. There will be three views in total: The list view, map view, and historical view. All of these views will be locked in guests and only users and administrators will be able to view these pages.

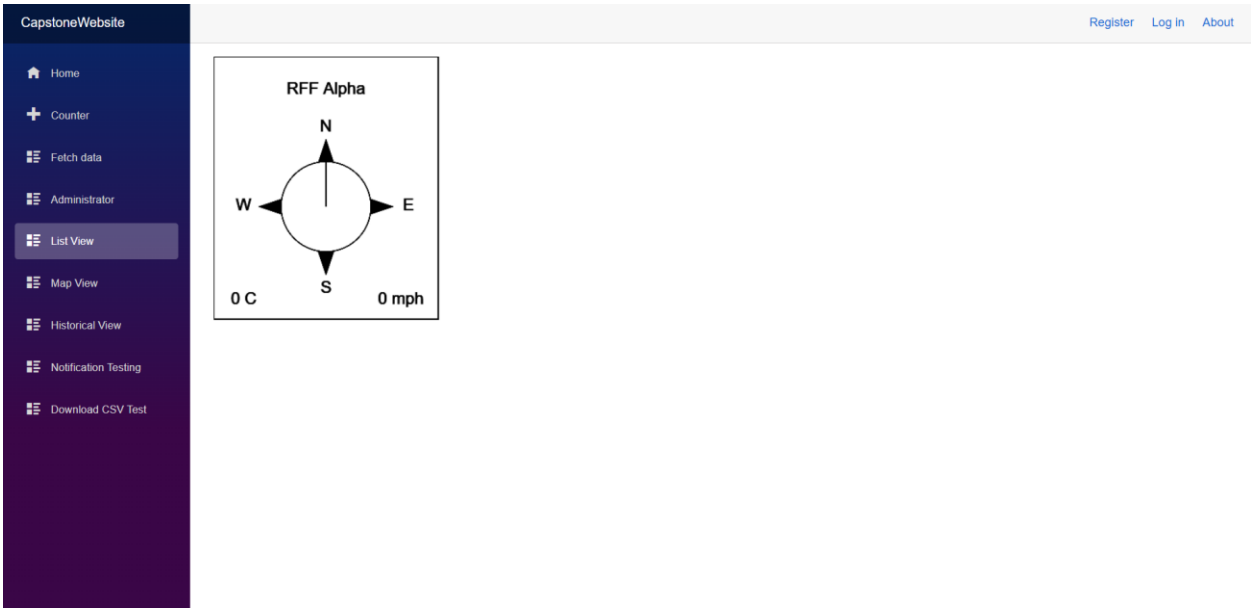

Figure 12: Example of graphical interface for list view on website

To begin, the list view will be responsible for providing a graphical summary of the latest SWAPR entries. This graphical view will have the name of the RFF site, a compass showing the wind direction, the temperature, and the wind speed all in a box. An example of just one SWAPR device is shown in Figure 12. When there are several SWAPR devices connected to the network, there will be multiple boxes just like the one shown in Figure 12 filling the page from right to left. The number of boxes shown should be directly correlated with the number of sites in the database.

The List View will be generated in the ListView class which will call the GenerateViews() function. The GenerateViews function will then call the DataManager getEntries() function and store it in the Entries list. The list will then be used to retrieve each entry and build the view from extracting the necessary information for said entry. The generated image will be of type Canvas2DContext, and it will be stored in a list with the other generated images to be returned when all images are created. Once all the images have been generated, the ArrayList<Canvas2DContext> will be returned to the page view where it will be displayed to the user.

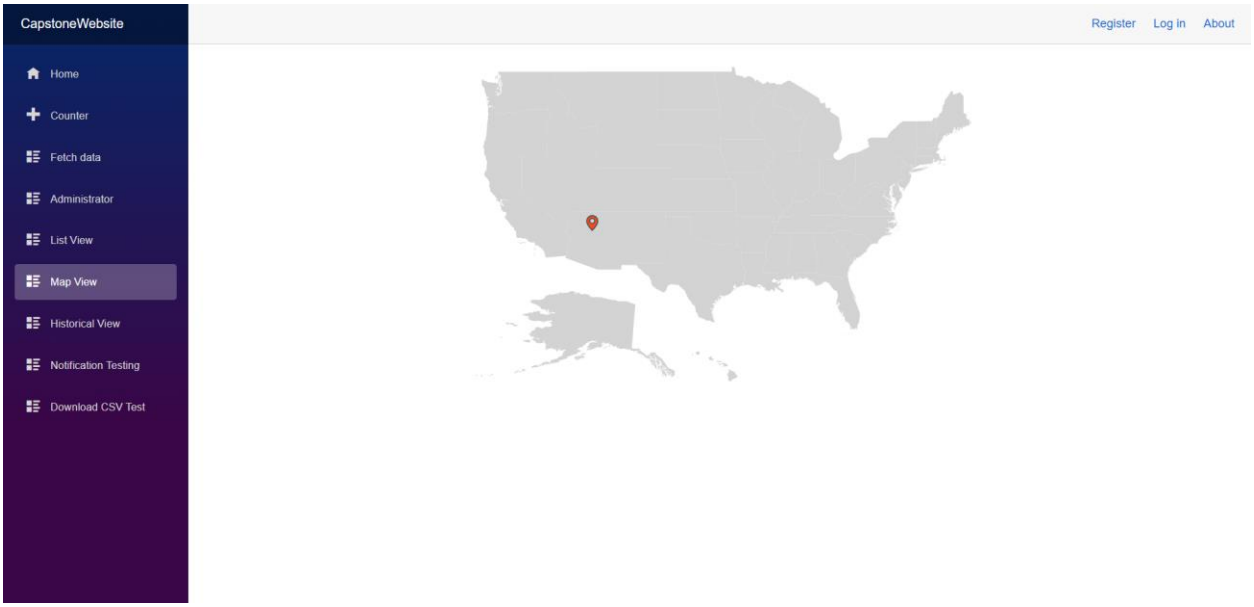

Figure 13: Example of graphical interface for map view on website

The next view discussed is called the Map View. The Map View is responsible for showing the latest entry from each SWAPR device on a map of the continent using color coded markers. For example, in Figure 13 there is a red marker on Flagstaff, AZ. That red marker tells us that the Flagstaff, AZ RFF site is not sending any data. That marker is just an example as there will not be a RFF site in Flagstaff, AZ. Instead, there will be markers lining the coast or large bodies of water. These markers will be the color of the status of the SWAPR so the markers will be Green, Yellow, Orange, or Red. Figure 14 will show exactly what each marker color means. The markers on the map are clickable and will take you to the historical view page for that SWAPR device.

The map view is currently generated using SyncFusion's map functionality. This makes it easy to produce the map but unfortunately it is not a free dependency nor is it cheap. As of right now, the map is produced using a list of states, latitude and longitude pairs that outline states for showing the map, and a list of markers which contain a name, latitude, longitude, a URL to redirect to when clicked, and a status. The markers are the only real operation that we must do when creating the map view. The markers are created by making a call to GetEntries() and storing the entries in the Entries list in the MapView class. The information for the markers can be found in the Site object of the Entry objects and the status can be found in the Status class using the composite key for the Entry object. We will create a list of SiteData objects which will contain only the name, latitude, longitude, url, and status which is then returned to the page view. The page view will then use that list to display markers all over the map in their real-world location and to set up the redirect functionality. When a marker is clicked, the MarkerClick() function is called which will use the C# NavigationManager class to redirect to the url which can be found in the url variable of the SiteData.

We currently like what SyncFusion has functionality wise, but we are not happy with the dependency for the long run because it will not help our client when they set up the project for themselves. For these reasons, we are trying to replicate the functionality of SyncFusion using OpenStreetMaps, OpenLayers, or EPSG.io. Our client is fine with us using SyncFusion however they would be happier with us using a free alternative. The client has stated that they will use their own mapping software if they do bring this project to production. The team would still like to use a free alternative to ensure the longevity of the project. Time permitting, it would be even better to recreate the functionality of the mapping software to really ensure there are no dependency issues in the future.

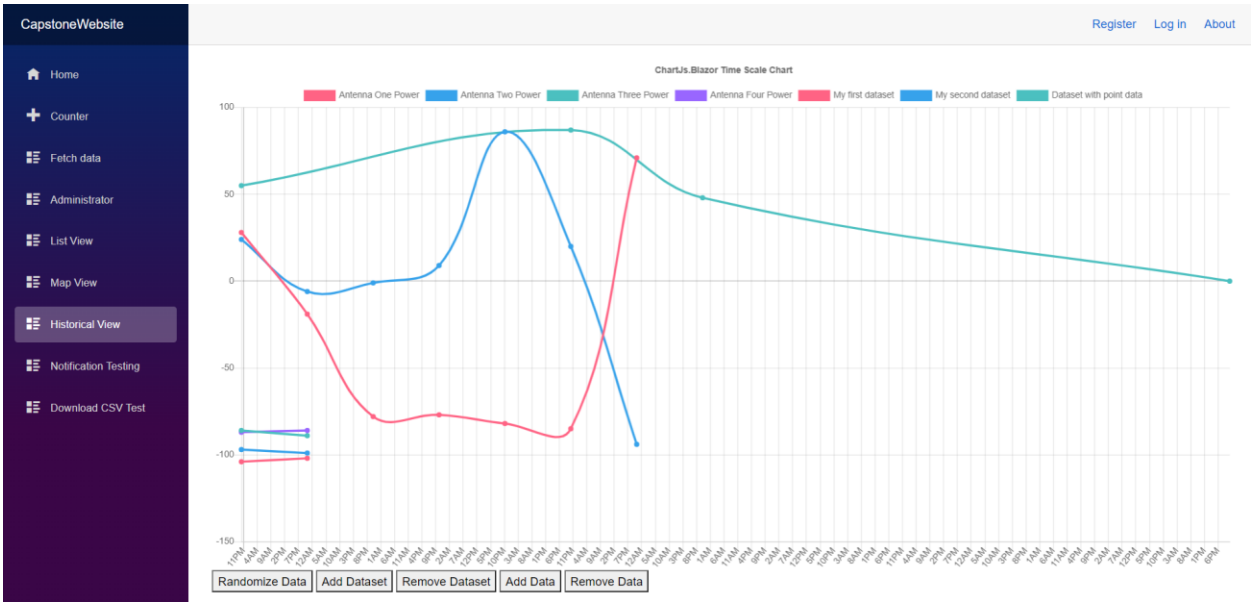

Figure 15: Example of graphical interface for historical view on website

The last view that will be provided on the website is called the historical view. The historical view is used to view the data from one SWAPR device over some time. The data can be shown in one of the following graphs: a line graph, bar graph, scatter plot, and wind rose compass. We show an example of the line graph in Figure 15. To start, the historical view will only show the graphs that make the data look the best. However, we would like to have the historical view change based on what the user wants. For example, the user could enter the start and end date of the data received, the data fields to use, and the graphs to use to display that data. That way the user could completely customize the graphics shown so they can be used in whatever way they prefer. Additionally, the historical view is where the functionality for the downloading of csv files with historical SWAPR data will be. This functionality will show as a button at the bottom of the historical view and when clicked it will generate a csv file and prompt the user with a download. The historical view will not have a refresh rate like the other views because this view is not meant to be an active view, so we only want it to show the data that the user requested.

The historical view will be created on the page load. It will be done by first calling the GetEntries() function and storing the resulting arraylist in the Entries variable in the HistoricalView. After that, we will call the GenerateLine(), GenerateHBar(), GenerateScatter(), GenerateRadar(), and or GenerateMSList() functions which will start by calling the GetDataSets() function passing in the Entries list. The returned DataSets will then be used or not used in the graph's generation depending on what data needs to be shown. The actual generation of graphs is done using a Blazor wrapper class of Chart.js. The configuration and setup for these graphs will be static to start but if time permits, it will be editable by the client.

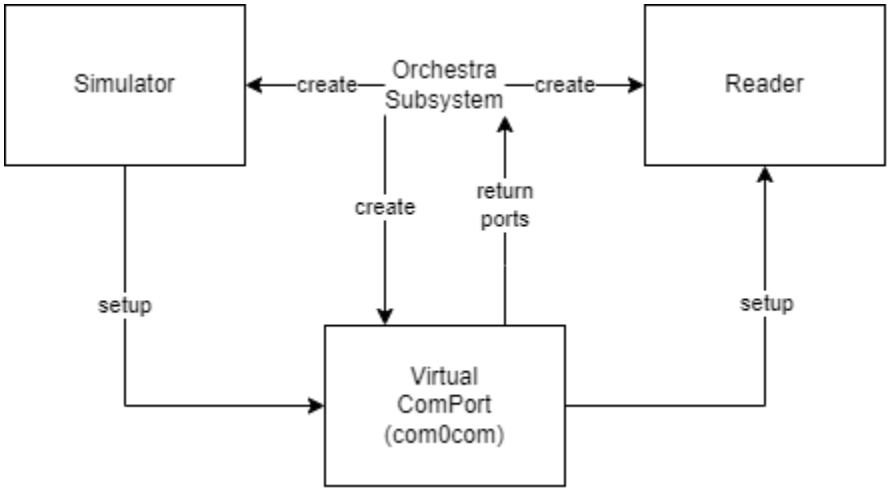

Figure 16: Diagram of Orchestra Subsystem

The final subsystem to our system is the Orchestra Subsystem. This subsystem is responsible for lab stress testing of the entire system. The Orchestra will stress test the system ensuring that there are no bugs or vulnerabilities. The Orchestra will do this by simulating at least 250 SWAPR devices that will be in the final production version of the project. The Orchestra is a stretch goal for the system so focus on creating it will start after a solid Minimal Viable Product of the rest of the system has been achieved.

The Orchestra will work by programmatically creating a virtual com port pair with the Com0Com software. It will then receive the ports to use from the Com0Com software and it will run a Simulator and Reader instance passing the port to communicate over on creation. This will then start the generation and storage of SWAPR data in the Database as described above. The Orchestra will most likely be a script written in C# if possible or Windows bash if not.

5. Implementation Plan

In the last section we described our modules and the interface for our system. Now we will discuss our objectives and timeline for completing them. We have included a comprehensive Gantt chart which lists our tasks and the time windows we plan on completing them which can be seen in Figure 17. Green lines represent tasks currently in progress, while red lines represent bug testing and troubleshooting, we will likely have to do in the coming months. For our purposes, the figure included in this document does not have specific date.

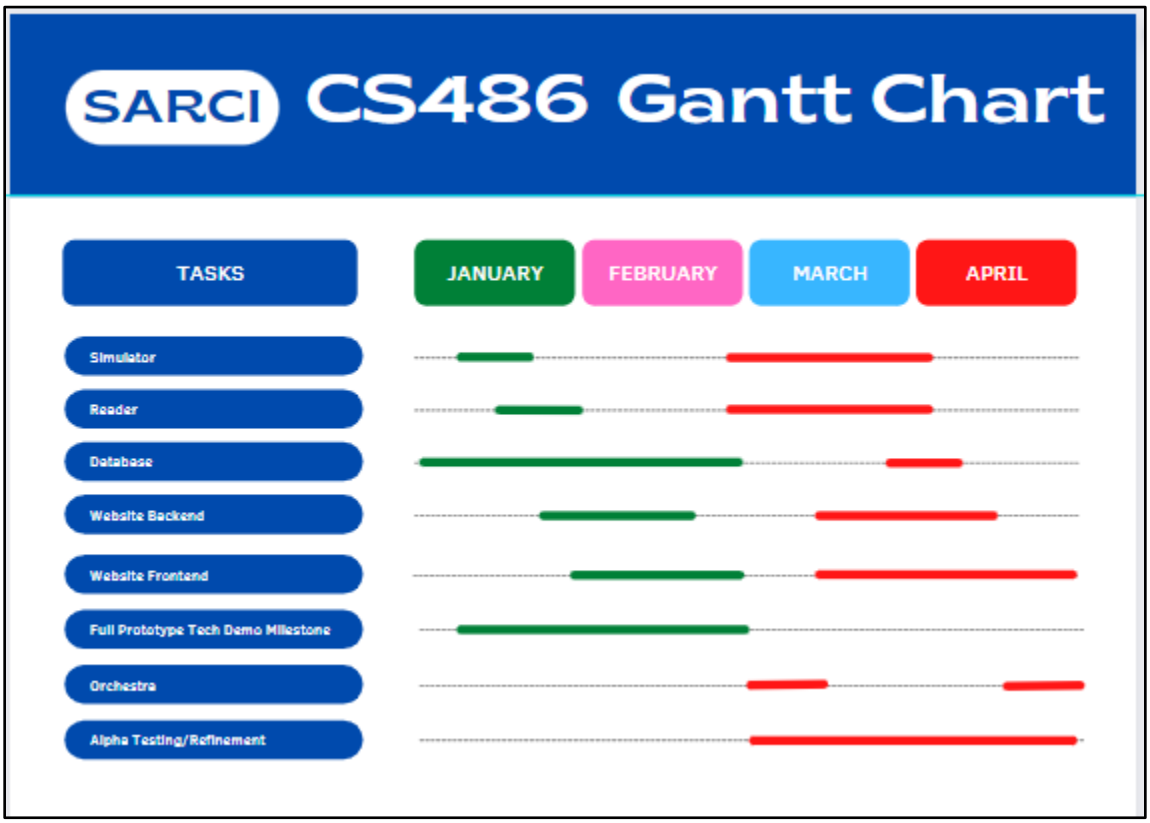

Figure 17: Gantt Chart

To walk through the major development phases in our project, tasks are represented by blue ovals on the left side of the chart. Our team has completed the initial development of the Simulator as indicated on the green line and the red line indicates the team will revisit the Simulator in late February and finish it by early April. We have also completed the initial setup of the Reader task and will similarly revisit this task late February to complete it by April. The Reader and Simulator tasks work directly with one another, which is why their development will need to be done in parallel. Our database environments have been established, and we are working to implement it with the Reader and Simulator subsystems. We plan on having them functioning with one another by the end of February and completing any final fixes or bugs by the end of April. For the Website Frontend and Website Backend, we are still working on creating functionality that

uses custom code or free dependencies. We plan to finish developing the major subsystems of the Website Subsystem by the end of February and revisiting them both through March and April. The next task listed is a Full Prototype Tech Demo which requires that all our subsystems work with one another smoothly and are ready to be shown. The Orchestra Task is a stretch goal for our team, meaning it will only be implemented if our Full Prototype Tech Demo is completed by March. If all goes as planned, the Orchestra will act as a multitude of RFF sites and be a more accurate depiction of how our software works in the Rescue21 system. The last task is Alpha Testing and Refinement which means our team will work to refine any bugs and make sure our subsystems are working smoothly with each other and the system is complete to the best of our abilities. We intend to do weekly code reviews to ensure the entire team is up to date with how our code works, the code follows our standards, and the code doesn't have bugs or vulnerabilities. All team members will be alternating between working on coding and documents, so the workload is reasonably balanced. We don't have a strict assigning of tasks, but it is roughly looking like Jabril will manage the Simulator and some DataManager functionality, Randy will manage the Reader and list view, Vidal will manage the map view and Orchestra, and Dylan will manage the AWS environment, Database, and historical view. Everyone will have miscellaneous tasks that relate to their subsystem. For example, Jabril would have to deal with some of the notification functionality because it is dealt with in the DataManager. These tasks are not permanent and changes to coding responsibility will happen throughout the semester to focus on whatever takes priority which will be determined by the group and client.

6. 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 will be building a secure web application for registering, configuring, and managing, the SWAPR network, and displaying output in a clear graphical interface. We will be implementing this through our five subsystems being the Simulator, Reader, Database, Website, and Orchestra subsystems. The Simulator will emulate the SWAPR device created by last year's capstone project. The Reader will read the generated data and send it to the database message queue. The Database will create entries from its queue and serve data to the Website. The Website will be responsible for providing a secure web environment with accounts and role-based permissions that provides several different graphical views, notifications for equipment malfunctions, and data exporting. The Orchestra will be responsible for emulating any number of RFF sites for stress testing in a lab environment.

Currently, we have each subsystem created with the most basic proof of concept working. We are working on refactoring the code and converting it to connect with the other subsystems. We will then make progress towards making a solid alpha version of the product before the beginning of March. As of right now, with the help of the content in this document, we are on track to provide a beta version product to GDMS by the end of April.

7. Figures

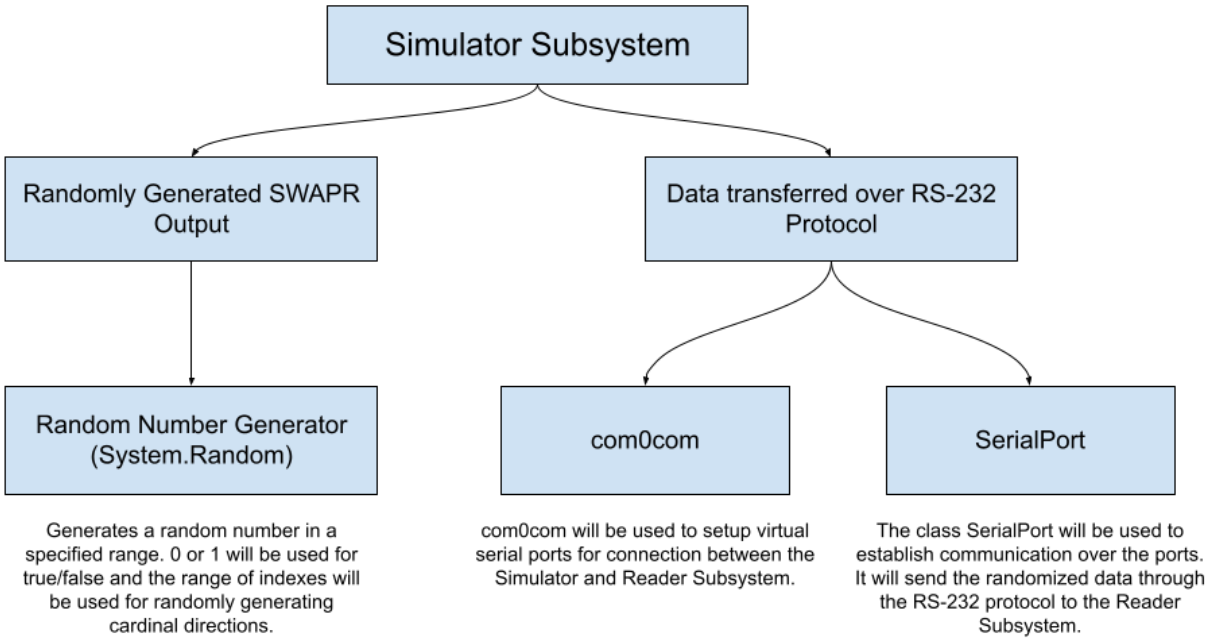

Figure 3: Overview of the Simulator Subsystem features

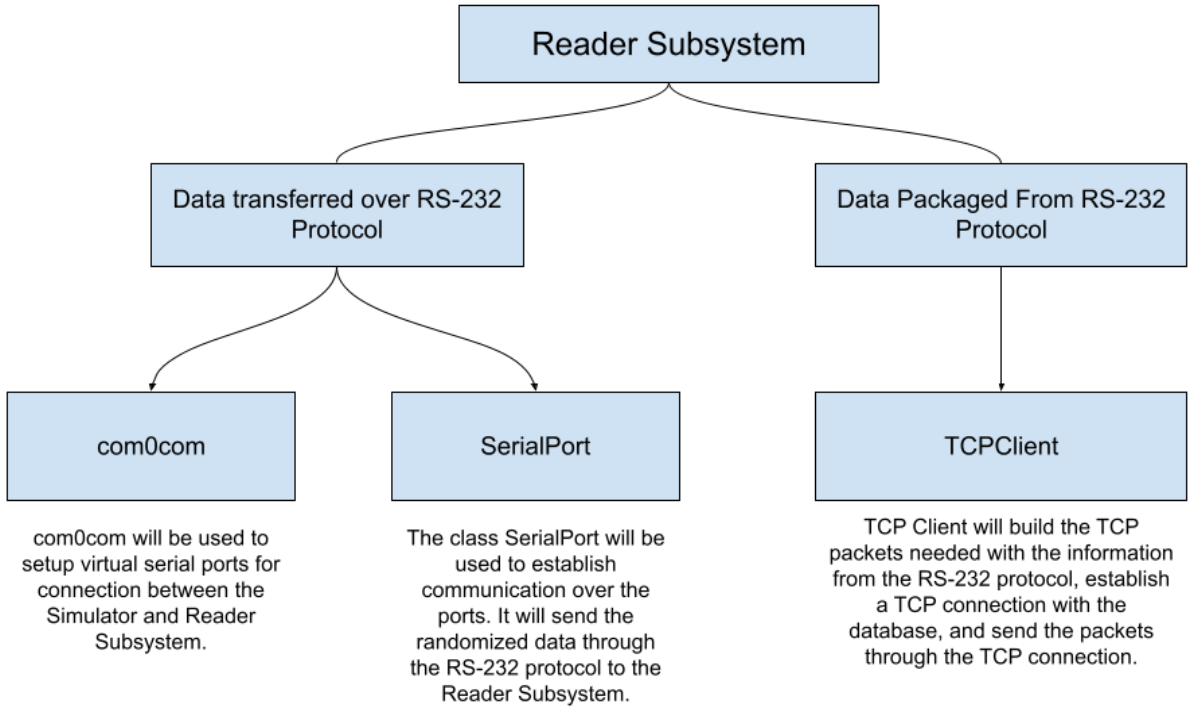

Figure 4: Overview of the Reader Subsystem features

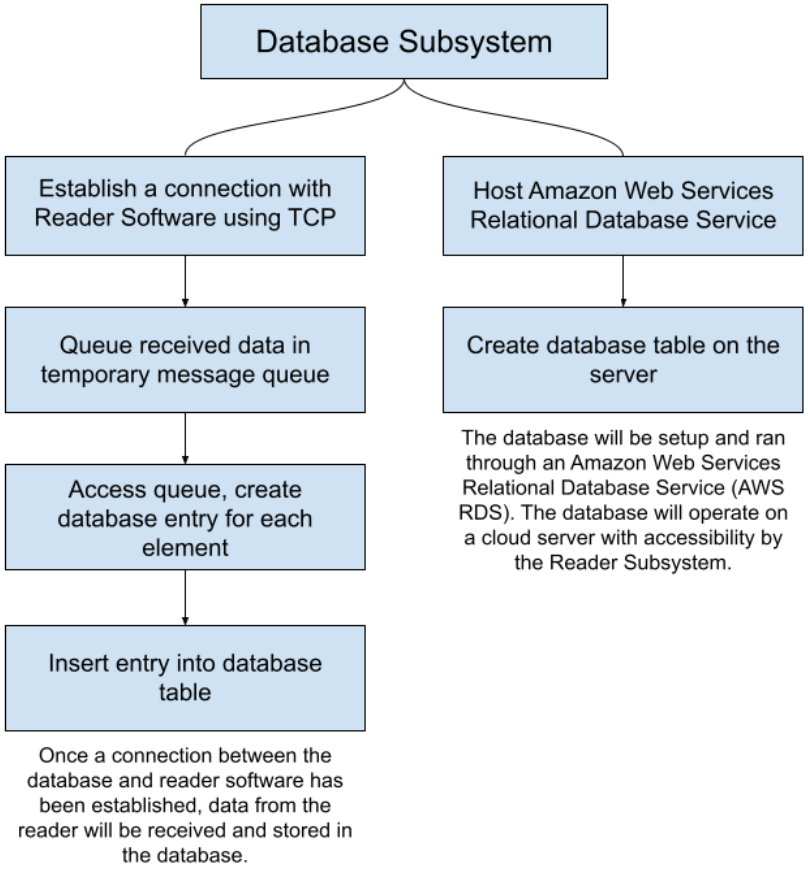

Figure 5: Overview of the Database Subsystem features

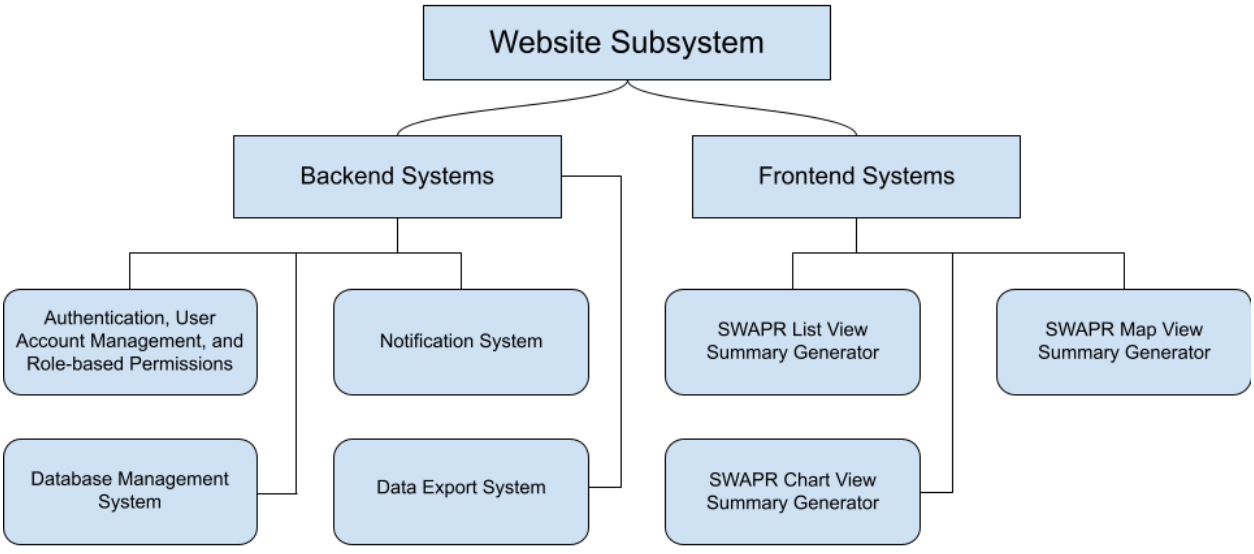

Figure 6: Overview of the Website Subsystem features

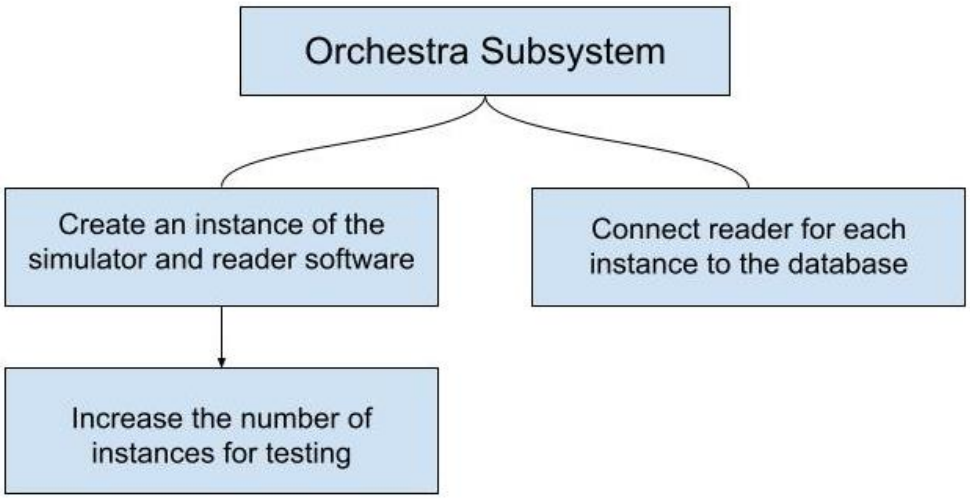

Figure 7: Overview of the Orchestra Subsystem features

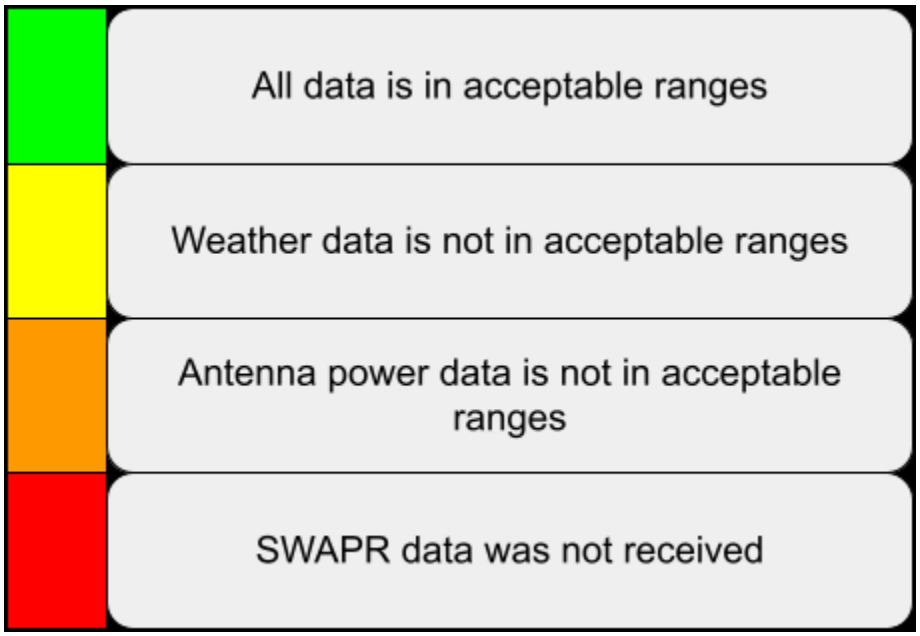

Figure 14: Table showing the meaning of different color statuses