Northern Arizona University

Team Radio Pi

Capstone Team	*Project Sponsor*	*Mentor*
Tyler Plihcik	**General Dynamics**	David Failing
Brandon Click	**Mission Systems**	
Jefferey Williamson	Randy Derr	
Kaelen Carling		
Saurabh Jena		

Final Report

Version 1.0

April 12th, 2021

Table of Contents

Introduction

The world runs on multitudes of IoT (Internet of Things) devices; everything from smart home devices, to wifi extenders, speakers, even cars and kitchen appliances. These devices communicate via radio waves and run on what are called embedded chipsets, which are inherently small, in both footprint and power consumption. As our technology continues to advance, the need for smaller and smaller embedded systems becomes more urgent.

For nearly three decades software defined radios (SDRs) have been revolutionizing the way these IoT devices communicate. Leading the charge into this new era of is our project sponsor, General Dynamics Mission Systems. The need for newer, smaller, more efficient yet more powerful systems is great. Current solutions are often times built on legacy software, and although they still work, are definitively not the best solution given our technological advances since the late twentieth century. Because of this, Software Defined Radios are prone to:

- Failure
- System crashes
- Memory overloads
- General decay

Our envisioned solution is to redesign General Dynamics SDR Command and Control System from the ground up. We will be simulating this solution on a Raspberry Pi, a very small yet powerful, computer. We will establish an interconnected database and web interface in the most efficient language(s) available, including networking protocols and database languages. We will then ensure that the end user has complete and total control, given they have the proper authorization, over the system. From querying the database to collecting, sorting, and storing data to controlling permissions and viewing system diagnostics, our envisioned solution will attempt to right all the wrongs of current command and control interfaces for embedded systems.

The following will serve as the final “as built” report for the Team Radio Pi capstone project. As built in this case meaning the product the team has built over the course of this year, with complete explanations and understanding of the technologies we have used to implement our solution. A full reading of this document by any team will give a clear and complete picture as to how the solution works, what technologies were used in it, and how all the various moving parts and pieces move together to form a cohesive product.

Process Overview

The processes that Team Radio Pi used throughout the lifetime of this project will be contained in the following section. These are listed in this document to give the reader deeper insight into how the product was built and why it was built the way it was.

Methodologies

To begin, the team followed an AGILE development model in which there are six main steps, meeting, planning, design, development, testing, and evaluation. The AGILE model is an iterative one, containing constant reevaluation and refinement, which is why it was chosen for the management of this project.

Team Roles

Throughout the duration of this project the teams role's evolved according to the needs of the project at the time. At the very beginning the team structure stood as follows:

Tyler Plihcik - Team Lead

Brandon Click - Communications Lead, Release Manager

Kaelen Carling - Architect

Saurabh Jena - Coder[1]

Jeffrey Williamson - Recorder

However as the project moved on, Kaelen became not only the architect but our primary engineer, while Jeffrey Williamson kept his responsibilities as the recorder, he also served as the database designer. Saurabh quickly became a "jack of all trades" as it were, he was flexible enough to move around to help in every aspect of the project from documentation to network engineering. It is very important in a project of this size that everyone always be on the same page and able to assist where needed, while still keeping their primary roles based off of prior knowledge and known skill.

---

1 All team members were responsible for some portions of the code base, however it was necessary for the capstone class requirements to have a designated "coder".

Task Management

Task management is an essential part of any successful project. Not only do tasks need to be given based off of personal bandwidth but also towards any team members specialty or knowledge.

For this project we used a free application called Trello. Trello is a variation of a classic kanban board, in which tasks are loaded into a column called a “backlog”, once the time comes for a particular task to be put into development it is then moved into the “doing” column. After the task has been implemented into the code base it is moved into the “testing” column and if it passes all necessary tests then it can be moved into the “done” column as is has been fully implemented and tested successfully within the project.

Version Control

The version control component of our project management tasks were operated by release manager Brandon Click. All code was stored on GitHub in a private repository in which access was only granted to team members.

For each major component or milestone of the development process a separate branch was created so that any work done in that particular branch could not corrupt any other progress made in the development process. After the new features in the new branch had been tested, a pull request was made in the repository and the release manager would then perform a formal code review. If the code passed the code review the pull request was approved and the feature branch was merged into the master branch.

Requirements

The requirements acquisition process began very early on in the Fall 2020 semester. From as early as the first team - client meeting we began discussing General Dynamic's vision for what we would eventually be developing. It is very important for the success of a project that the client and the development team are on the same page from the beginning, it is for this reason that for the first few weeks our discussions were very high level, gathering overall requirements, and not yet dealing with specifics.

As the semester progressed, we had gathered initial requirements. These pertained mostly to the database, languages, and frameworks we would be using, including some others. After this initial gathering we performed a technological feasibility study in an effort to prove that the various choices of technology that we had made based off of early requirements would work well in this project setting, and that we could achieve the goals of the client.

Following the technological feasibility study we continued to have an open line of communication with the client and began both the prototyping and actual requirements gathering phase. The requirements that resulted from this period of development were, in short, as follows; the database shall respond to all queries and provide the necessary data, it shall save radio data provided by multiple software functions, will implement multi-threading for access from multiple processes and or applications, provide the status of the embedded device, in addition to timestamps for information. Additionally it is essential that the database would support storage of multiple datatypes, and provide information as to its own metadata, as well as preserve state in the case of a catastrophic failure and survive unexpected external events pertaining to the functionality and environment of the embedded device itself.

Architecture and Implementation

The diagram pictured below (Figure 1) represents a high-level architectural view of our solution. It contains three distinct major components, each with its own smaller yet still very important components inside. Following this top level overview, a more in-depth exploration into each of these major components is made below.

We will look at this diagram from the bottom up. The first, and most basic component of the solution is the SDR embedded device which itself has 3 smaller components in it, in our case this is a Raspberry Pi 3 B+. The device has its own database, storing all device specific information locally. This communicates with our data manager which receives, formats, and stores the data from the device. The data manager connects to the network we have built for communication, this component connects the simulated SDR devices to the next major component, our Shepherd C2 System.

The Shepherd C2 System is the “go-between” for the SDR devices and our front end user interface (UI). We have dubbed this component the “Shepherd System” because it acts as the shepherd, controlling his flock (our devices) and keeping them all in one place. This major component also contains three smaller components within it. Firstly, we have the Shepherd System itself, which is networked to each SDR device over the network mentioned above. This contains the addresses for our front end, and the SDR device database. It will process, send, store, and retrieve data. This SDR device database is the staging area for all of our embedded devices, where you can see all devices, on and offline, and is the access point to individual device data. Driving this component is our custom built Shepherd API, which receives and parses the requests from our front end to the Shepherd System..

Finally, we have the front end UI. This is the component which allows the user to actually command and control the devices being monitored on our command and control system. In the interface, the user is able to see the status of all devices registered in the SDR device database, and access the information in each one. The user is also able to request specific information by performing certain queries, after which the data is displayed to the user in such a way that is easily digestible.

Our solution as a whole, in short, works as follows: the simulated SDR device stores it’s data locally, and is networked with our Shepherd System, this system contains all SDR devices, and the user is able to access the Shepherd system and the devices it contains via the front end user interface.

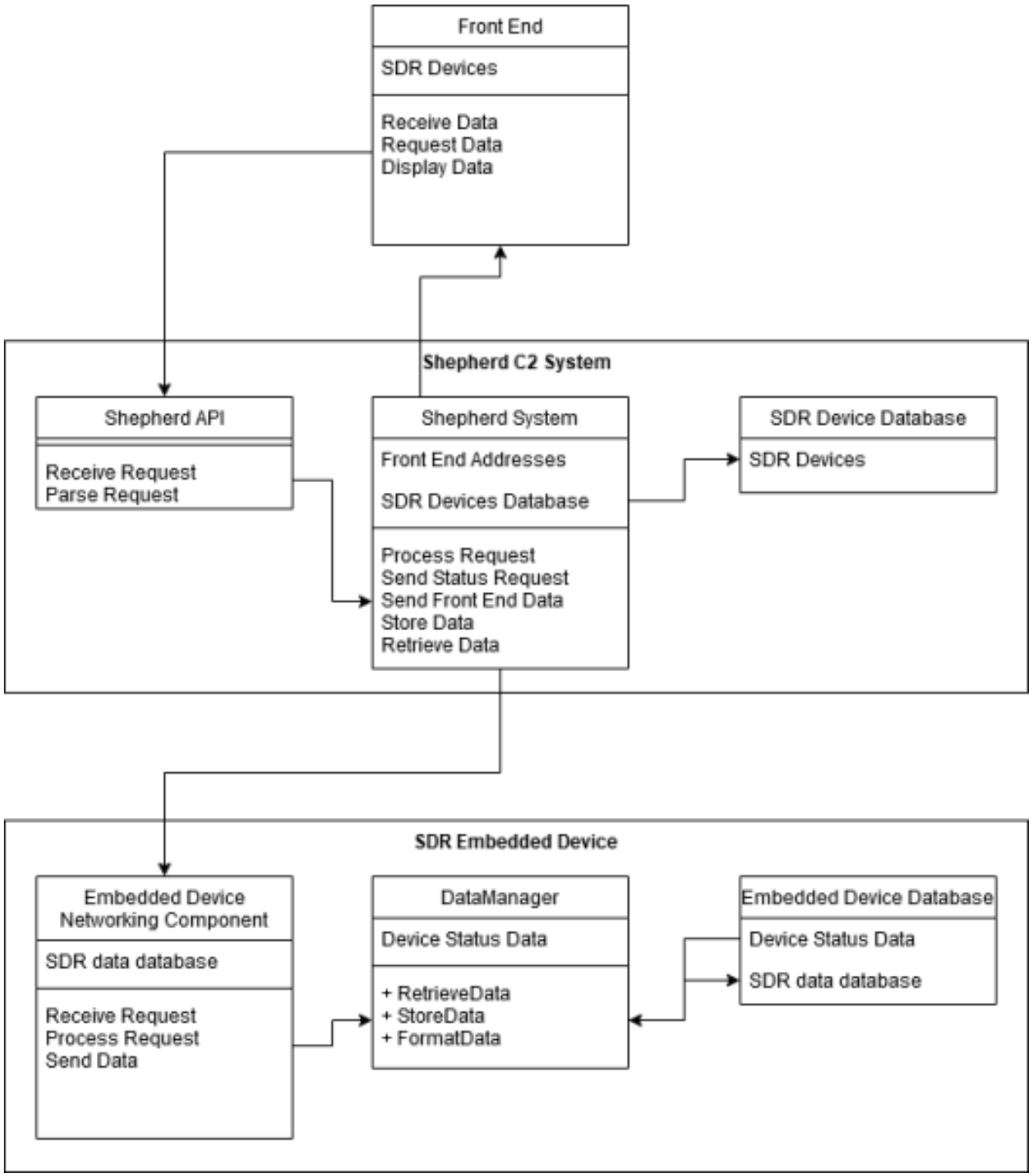


Figure 1. Architectural Diagram of Team Radio Pi C2 System.

Section 1:

SDR Embedded Device

The SDR embedded device is be simulated on a Raspberry Pi 3 B+, because of its low resource consumption and availability it is a perfect substitute for an actual SDR device.

Section 1.1:

Embedded Device Database

The embedded device database is one of the core elements of our solution. It houses all essential device information that is needed for the command and control system.

The database two main tables in it:

- Device Information
- Log Files

The Device information table contains the following:

- Time of check, which will be a formatted string of the exact time that the information being displayed to the user was requested from the database.
- Device ID, which will serve as the primary key, and must not be null.
- Current status, which is a string, either “on”or “off”
- Database size, an string containing the size of the database.
- Temperature, which is real number, representing the temperature of the embedded device.

The Log file table contains the following:

- A Log file detailing information about the embedded device

Section 1.2:

Data Manager

The device data manager interacts with the embedded device database by storing device data, and is able to access and query the device database. This includes retrieving data from the device, formatting that data, and storing it.

Section 1.3:

Embedded Device Networking Component

The embedded device networking component is the data pipeline driving our solution. This will implement TCP, with an easy addition for UDP support if the need arises. TCP is used because of its extreme reliability and extensive documentation in the case that support is needed.

Section 2:

Shepherd C2 System

The Shepherd C2 System is the backbone of our solution. Its purpose is to have all of the SDR devices stored in one place, virtually speaking. This streamlines the management of the devices.

Section 2.1:

SDR Device Database

The SDR device database has been implemented as a field within our overall database to save space, and time in development. It simply contains the SDR device ID, which is used to then access the individual device's data.

Section 2.2:

Shepherd System

The Shepherd system is the most complex part of the overall Shepherd component. It uses the front end address and a connection to the SDR devices to process requests, send status requests, send data to the front end UI, store data in the device database (in the case a new device is registered, or an old device is removed), and to retrieve data from the SDR device database. This system interacts closely with the custom Shepherd API described below.

Section 2.3:

Shepherd API

The Shepherd API is our mediator between the front end user interface, and the Shepherd C2 System. This component receives requests from the UI, and parses those requests out to the Shepherd System, which initiates contact with the database, which obtains information from each individual device over the network, using TCP.

Section 3:

Front End User Interface

The front end user interface is a stand alone component of our system and connects to the Shepherd C2 System. This interface allows the user to view all registered devices, their status access the data stores in their database, query specific information from specific devices, as well as add or remove devices from the system as a whole.

Testing

The following will was the software testing plan for our solution, detailing the testing process and project testing benchmarks. Our solution can be easily broken up into three major parts: front end user interface, our custom shepherd system middleware, and the backend database and networking functionality. More focus will be on individual unit testing than on integration testing because we have so many more units than whole pieces of the project. Additionally, because one of the client's major concerns is how our solution handles under load, a large portion of the unit testing for our backend, namely, our database focuses on stress testing and security testing for our networking functionality as well as our database administration actions.

Unit Testing

Software unit testing is the first stage of testing in the software design process and life cycle. It involves taking apart each of the major portions of the project as a whole and breaking them down into their individual components. For example, a car as a whole can be broken up into major components: engine, chassis, and enclosure, but you can further break those down into the engine block, the timing belt, then fuel injection system, the wheel base, suspension, doors etc… The main point of this unit testing is to ensure that everything functions properly on its own before it is integrated into the system at large. This guarantees no major failures when everything is being put together.

Front End User Interface

Our front end user interface (UI) is the face of our project, what the client sees in every demo, and what they see and interact with every time they use our solution. The front end is built using basic HTML and CSS in combination with Flask. It consists of the actual functionality of the HTML, ie the forms, submit buttons, and responsiveness, and the functionality of our flask code, which will be making the connections to our backend. This is the actual logic that begins as soon as the user presses the submit button on any data entry field.

In order to test the HTML portion of our front end, we ensured the following:

- That all forms were set up properly with the correct method (POST, GET, etc..)
- That all forms were configured to transfer to the correct information to the correct destination, ie the action
- That all submit buttons had the proper value on them
- That all forms took in the proper input type (number, text, etc..)

To test that all forms were set up properly for our UI, (this includes the first three bullet points from above) we performed manual testing on each form with a random set of acceptable values,

and confirmed receipt of those values by observing the actions performed by the front end after their input and submission. Proper set up of the forms that make up our UI will be confirmed if upon submission of our random sets of acceptable values the front end successfully has sent the information entered in the data entry field(s) from the UI, however receipt of these values will be detailed and tested later, for the purpose of this particular unit test, we simply wanted to confirm the send.

To test that all forms took in the proper input type we performed manual testing on each form by inputting a random set of both acceptable and unacceptable values, submitting them, and observing the result. Proper intake was confirmed by seeing there was no error message telling the user that their input was not valid.

In order to test the Flask portion of our front end, we ensured the following:

- That receipt of the values in their proper destination was confirmed
- That when the submit buttons are pressed, the proper database queries were sent
- That when the proper database query has been sent, the proper database response was sent back
- That the front end itself is actually connected to our networking component via sockets

To test that the values are being sent to their destinations, after inputting a random set of acceptable values we observed which actions of the flask code are triggered, i.e. if we entered a lookup for the device information of device identification number three, that the primary key deviceID with the value 3 is sent to the proper flask function which then triggered our next test.

Next, to test that the proper database queries were sent and correct information from the database was received, after successful submission of acceptable values into our data entry fields, we recorded the database query that is sent to our database, and observed the value that were sent back to the UI. If the information sent back to the UI in the use case detailed previously matches the database records for that specific device then the test was marked passed.

Finally, to test that the front end itself was actually connected to our networking component via sockets we simply tried to query any particular piece of information from any data entry field using a random set of acceptable values. Any response containing information from our database confirmed successful connection.

Database

The integrity of our database and its proper function are essential to the success of this solution. In a way, the database is the heart of our project, and its testing is of paramount importance. The database holds vital information such as device identification numbers, core computer data and log files etc…

In order to test the database, we ensured the following:

- That the database continued to perform under load
- That the database size remained small
- That the database connection persisted throughout a user session
- That the database responded to queries accurately and in a timely manner
- That the database was secure and not subject to attacks such as SQL injection attacks

To test that the database performs under load we have performed a great deal of stress testing. This was one of the client's biggest concerns. We performed various tests in an attempt to answer the following questions:

- How much space does each record occupy in the database?
- How fast can the Radio Pi write to the database?

In order to do this, due to the nature of stress testing a database we created our own automated test functions that will run at the Pi's maximum computing speed. We then recorded and documented the results.

To test that the size of the database remains small performed a test in which we ran the Radio Pi nonstop for an arbitrary period of time where it wrote to the database. We ended this test when the database broke due to size, or when we reached a database size exceeding client expectations while the number of writes to the database fell below client expectations.

To test that the connection persists throughout an entire user session we performed a test in which we wrote to the database an arbitrary number of times and ensured that the data sent was received and placed in the database.

To test that the database responds to queries accurately and in a timely manner we wrote a test function that queried the database, and recorded the time that the query was sent. Upon the query response we recorded the time in which we got it and that the received response was the correct response.

To test that the database is not subject to security vulnerabilities such as SQL injection attacks we attempted our own SQL injection attacks against the database, performing our own penetration testing. Success of this depended on whether or not we were able to successfully attack the database. Which we were not.

The following are the results from our database testing, each one starting with the questions we asked ourselves, the procedures, the results, and our conclusions.

Test One - Size

Questions

How much space does each record in the database occupy? How much space is a days worth of data from the status function?

Procedure

1. Code a testing class file that runs the status update function every 5 seconds
2. Run this testing class for 90 minutes.
3. Record database information (size, number of records, time spent running)
4. calculate metrics about size and frequency of entries

Recorded Data

Initial Recorded Data	
Number of records	1024 records
Time spent running	84 minutes
Database size	108000 bytes (108kb)
Update interval	5 seconds

Constants

1. Seconds in a day = 86400 seconds
2. Bytes in a Kilobyte = 1024 bytes

Calculated Data	Math	Result
Status updates in a day	86400/5	17280 status updates
Size of an individual record	108000/1024	105 bytes per record
Database growth in a day	17280 * 105	1814400 bytes (1.814mb)

Conclusions

With size of 105 bytes per record that means that if the RadioPi records status updates every 5 seconds then that means the Database will grow by 1814.4 kilobytes or 1.814 megabytes everyday. It should also be noted this is just the size of the status update table. Any other tables added might have different metrics/sizes.

Test Two - Speed

Questions
How fast can the Radio Pi write to the database?

Procedures
1. Code a testing class file that runs as many status function updates as possible in a two second time interval.
2. Record the number of entires as well as the time spent running
3. Calculate the speed of the entries

Recorded Data

Initial Recorded Data	
Number of records	1022 records
Time spent running	1.82 seconds

Calculated Data	Math	Result
Database updates in second	1022/1.82	561 updates a second

Conclusions
The database is able to have status updates written to it 561 times per second.

Test Three - Environmental Tests (Temperature)

Questions

How does the device handle extreme load? Namely, does is stay within safe operating temperature?

Procedure

1. Code a testing class file that runs the stats update function every five seconds.
2. Retrieve the temperature recorded for each entry.
3. Calculate the average temperature, the lowest temperature, and the highest temperature

Recorded Data

Initial Recorded Data	
Highest temperature recorded	49.388°C
Lowest temperature recorded	45.084°C

Constants:

Raspberry Pi 3 B+ maximum operating temperature of 85 degrees Celsius.

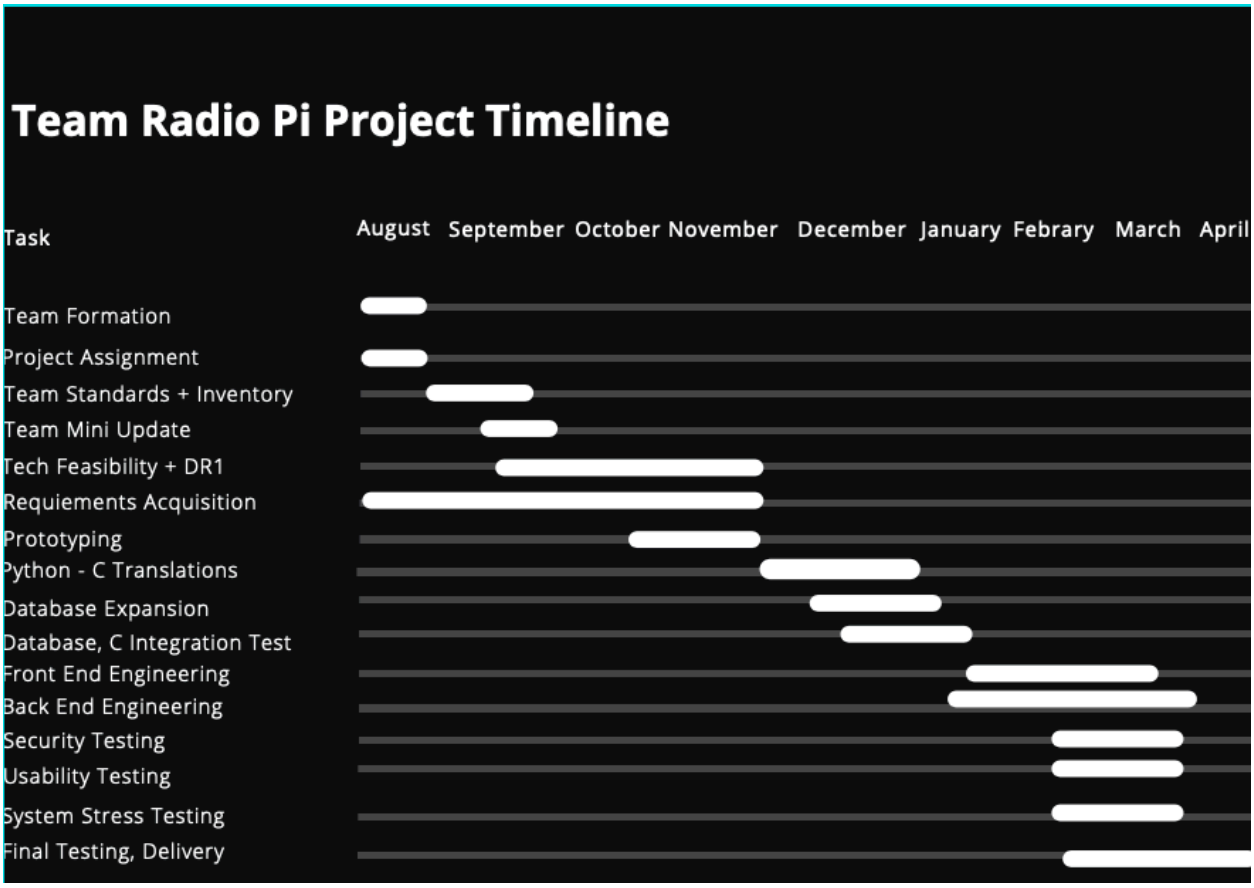
Calculated Data	Math	Result
Average Temperature	(Summed 1024 entries of the current temperature) / 1024	46.16°C

Conclusions

With the standard 5 second status update interval the RadioPi never got above 50°C or below 45°C. Also the average temperature while the status updates were running was 46.16°C. None of these numbers were anywhere close to the maximum temperature the Raspberry Pi can withstand.

Project Timeline

Throughout the lifetime of this project we went through many major milestones. From team formation and project assignment, requirements acquisition and prototyping, implementation to delivery. All of these milestones and each milestone in between can be seen in the Gantt chart below, a timeline detailing from the start of the Fall 2020 semester in August, to the end of the Spring 2021 semester in April. The two semesters served as a break up between thew two major parts of the project, requirements acquisition and implementation. Team Radio Pi in this case elected to use the time “off” over winter break 2020 to continue development in order to ensure that we were able to implement all major aspects that the client had detailed in their overall vision for this project. Leading into the Spring 2021 semester we had a great head start into development, and hit a few roadblocks along the way, including even a bug in the Raspian Operating System we are using, but all of those were cleared with time to spare.



Future Work

Future development and additions to this project are crucial, as mentioned earlier IoT devices and the need for more efficient C2 systems are constantly evolving. Future work in the team's vision includes but is not limited to:

- A more robust networking solution
- Additional expansion of the database to include more tables and fields
- Testing with actual SDR devices.

We wished to add a more robust networking solution for our product, but the project timeline did not allow for this if we were to get everything else done that the client had specified in the initial requirements stage.

In the very beginning of this project, the client outlined many different data types, fields, and tables that their current solution contains and has support for, however for the sake of the project and the known life time of it, we elected to implement only a few of those. Future work should absolutely include a vast expansion of the current database implementation to cover all of the previously supported types of the client's solution.

Due to the nature of General Dynamics Business, we were unable to test with their actual devices, we used Raspberry Pi 3 B+'s because of their small footprint, low power consumption, and limited resource availability. However close the Pi's were able to get us in our implementation to real SDR behaviors we will not know, but future work should include testing by GDMS with our solution on actual SDR devices.

Conclusion

At the beginning of this capstone experience, we were assigned GDMS as our client. Throughout the last nine months we have been interfacing on a regular basis with Randy Derr, and various members of his team, in collaboration with our team mentor, Dr. David Failing. We were initially presented with a need for a redesigned command and control system for embedded products, we quickly came to the realization that our solution, upon delivery, would serve General Dynamics as a proof of concept for a completely redesigned C2 system, and they would then use the research, development, and work that we have done to take a very serious look at the solution that they currently have, and explore the feasibility of implementing some, if not all, of our solution into theirs.

The importance of continuous development of these systems is paramount because we live in a world today that is dominated by IoT devices, which run on embedded systems. The better the systems are that control these IoT devices, the more efficient they will be and the more effective they will prove in the lives of the people that use them every day. What we have done for GDMS is saved them the time on researching the feasibility of a new system, and provided initial development of such a system for them to do what they see fit with it. As General Dynamics is an industry leader in IoT devices, anything they do and or release sets new standards across the field of technology to further advance society.

In closing, the performance of the members of Team Radio Pi, Architect and Back End Engineer Kaelen Carling, Coder and Network Engineer Saurabh Jena, Communications Lead and Release Manager Brandon Click, Recorder and Database Designer Jeffrey Williamson, and Team Lead Tyler Plihcik were admirable and up to par of the real life industry development teams. The mentorship of Team Mentor Dr. David Failing was invaluable, and the oversight of our Client Randy Derr, and General Dynamics Mission Systems was second to none. The project had some roadblocks, from non disclosure agreements, to operating system bugs, but none of them were by any means show stoppers. What our team was able to complete in nine months of development during this capstone class, the lessons learned, hours spent, and experience gained was incredible, and will prove to be a fantastic experience as the members of Team Radio Pi prepare to enter the field of technology in whatever capacity they happen to fall into.

Glossary

IoT - Internet of Things devices - such as wifi routers, smart home devices, etc…

HTML - Hyper Text Markup Language - the markup language used in web page development

CSS - Cascading Style Sheets - a way of styling HTML pages

GDMS - General Dynamics Mission Systems

SDR - Software Defined Radio

UI - User Interface

Appendix A

The following will serve as an overview of any and all technologies used that are essential to running the product and performing future development on it.

Hardware

The team developed on a variety of different machines, Linux, MacOS, and windows, however all code was run and tested in a Linux environment, namely, Ubuntu, and on a physical Raspberry Pi 3 B+ running the latest version of Raspian OS. The Pi ran off of the following core technology:

- SoC: Broadcom BCM2837B0 quad-core A53 (ARMv8) 64-bit @ 1.4GHz
- GPU: Broadcom Videocore-IV
- RAM: 1GB LPDDR2 SDRAM
- Networking: Gigabit Ethernet (via USB channel), 2.4GHz and 5GHz 802.11b/g/n/ac Wi-Fi
- Bluetooth: Bluetooth 4.2, Bluetooth Low Energy (BLE)
- Storage: Micro-SD, 32 GB
- GPIO: 40-pin GPIO header, populated
- Ports: HDMI, 3.5mm analogue audio-video jack, 4x USB 2.0, Ethernet, Camera Serial Interface (CSI), Display Serial Interface (DSI)

There are no “minimum requirements” for a machine used in development, as long as said machine was built in the last ten years, it will more than suffice.

Toolchain

During the development of our product, our primary development environment for the front end UI was PyCharm, professional edition. The primary development for environment for the back end was CLion, profession edition. SQLite3 was the database used in development and implementation. Sockets were used to build our networking component. To build our front end we used the Python micro-framework Flask. Flask was needed for this project as a way to turn our web page into a web app. All on device development was done using a Raspberry Pi 3 B+ running Raspian OS, updated to the latest version. Testing was done on the C2 dashboard with all major internet browsers, Firefox, Safari, Internet Explorer etc..

Setup

For setup on the simulated SDR device, the server.py file must be run on the device to initialize database actions. Outside of downloading and initial personal set up of PyCharm and CLion, no special configurations are needed other than configuring PyCharm to run the most updated Python 3.x interpreter. To be able to run the Flask portion of the solution, Flask, and flask-forms must be installed via pip.

Production Cycle

For production, the front end must be run off a Flask production server, and the backend must be run off of the Raspberry Pi 3 B+, which simulates the actual SDR device. The make file included in our repository is all that is needed to compile and deploy the backend of the product written in C.