Northern Arizona University

Team Radio Pi

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

Technological Feasibility Document

Version 3.0

January 19th, 2021

Contents

Introduction 3
Technological Challenges 4
Technological Analysis: Multi-Processing vs Single-Processing 5
Technological Analysis: Database Management System 7
Technological Analysis: Database and Application Language 10
Technological Analysis: Networking Protocols 12
Technological Analysis: API and Web Application Interface 14
Technological Integration 17
Frontend - Device Communication 18
Conclusion 19

Introduction

The world runs on multitudes of various IoT (Internet of Things) devices. These devices are anything from smart home devices, to wifi extenders, speakers, even cars and kitchen appliances. All 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 prevalent each hour.

For nearly three decades software defined radios, or SDRs have been revolutionizing the way these IoT devices communicate. Leading the charge into this new era of devices and communication platforms 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 for this rising issue is to completely redesign from the ground up, using modern technology in both hardware and software the way we interface with SDRs today. We will be simulating this solution on a Raspberry Pi, a very small yet powerful, computer. We will establish a database and a web interface, connect the two, implementing these 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, collecting, sorting, and storing data, 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.

Now that we have outlined our project's scope, and gone into brief detail of our envisioned solution, we will describe the technological challenges we feel will be most adverse in the development of our solution.

Technological Challenges

Our solution must save data coming from the radio by multiple functions within the radio, respond to queries of the database, define querying protocols, and define thread safe access from multiple processes and or applications. It must also provide status of data transactions, contain metadata, and be completely reconfigurable in the case of data deletion. Additionally, the database must support multiple data formats, save state upon unexpected or scheduled system shut down, and operate on limited computing resources.

From this list of requirements, we have derived the following five major areas to research as we see them to be the pillars of success on which our project will be built on.

- Process Threading
- Database Management System (DBMS)
- Database and Application Language
- Networking Protocols
- API, Web Application and Database Interfacing

In the following sections, we will go into detail on these five areas. Describing the issue in depth, providing desired characteristics of a potential solution. Then we will survey the viable options, and make a recommendation based on the information given, and set forth a plan of development and testing for the solution that we selected.

Technological Analysis: Multi-Process Vs Single-Process

Intro

In order to achieve the directives of this project, that state we want fast, clean, and efficient code we need to decide between two clear set options. Do we choose to only have one process running at any point in time, or do we utilize threading in an attempt to speed up our total run times by running multiple processes at the same time.

Desired Characteristics

We are looking for the option here that not only provides us with a high amount of stability, but one that offers the fastest route possible for processing and accessing our data. We are looking for something that provides a safe environment to run our code in, avoiding any potential for system locks or false sharing of data.

Alternatives

As it stands, there are only two options available to us. We can either utilize Multi-Processing, or stick with Single-Processing. Each holds their own advantages and disadvantages. Multi-Processing is the act of utilizing multiple cores on a processor to run multiple programs at the same time. Single-Processing only allows for one program to be running at any time, halting processing whenever a new program must be run.

Analysis

Multi-Processing offers us a quick and efficient way to speed up our run times. By fully using the cores provided to us by our hardware we are able to run multiple tasks simultaneously. This, when properly utilized with longer running tasks, cuts a significant amount of runtime off of our program. However with running multiple programs at the same time we run the risk of encountering issues with thread locking, false sharing, and data races.

Single-Processing is when we simply run our code in a linear way, jumping from one task to another, pausing the current task whenever something else must be handled. This provides a lot of safety as it never runs the risks that Multi-Processing can incur. However it can be extremely difficult to cut down run times with Single-Processing, relying on efficient coding tactics to make up for the speed increase.

Chosen Approach

We have decided to go with Multi-Processing. With its possibilities to handle multiple actions at the same time we deem it is worth the risks that it carries with it. To mitigate these risks we can partake in various coding standards and safety procedures when designing our code in order to minimize these risks.

Proven Feasibility

In order to demonstrate and assure we have selected the appropriate choice, we can employ a simple testing strategy that will be able to show that we are achieving our desired characteristics as stated previously. Firstly we can test the speed of which our programs are running. By simply coding in a simple timer, we can track the speed of which each thread completes its job, by calculating the time needed to switch threads, we can then compare the total run time of our program to one that would have been run in a Single-Process environment. If the speed difference is greater than 10% between these two, then we can safely say that Multi-Processing was a good choice.

Technology Analysis: Database Management System

Intro

In order to support a software defined radio, there needs to be a system in place to track how it is performing and log any and all changes. One obvious solution to accomplish this is to implement a database. A database to handle this needs to be small to accommodate the limited resources available in an embedded system. For our purposes we are developing for a Raspberry Pi 3B+, and this limitation will need to be kept in mind. Our solution should be general enough to also apply to a range of physical devices.

Desired Characteristics

The most basic characteristic is that the database will save radio data provided to it from multiple other applications and functions. This data needs to be stored even when the device is powered off. The database also needs to safely survive unexpected shutdowns. Any data that has already been stored should be accessible after power is restored.

A database that cannot be accessed is useless. Any database management system chosen will need to respond to requests as well as provide data to other software. Aspects of this are covered elsewhere in this document, including the network protocols, API, and web application. The database will need to support a variety of data types including binary, hexadecimal, strings and links to files. Another data type that needs to be supported is some form of date or time to be used to timestamp operations. It is possible that a database management system does not include every single necessary data type explicitly. If this is the case, then some kind of acceptable substitute for these data types will need to exist.

In regards to the environment the database will exist, it needs to operate in a Linux operating system. The exact Linux environment is not vital to this particular aspect of the project, but it may affect other factors in development. Ideally the database management system will use the fewest resources possible to allow other software on the embedded system device to perform optimally in their operation. This means that any database will need to run with constraints to the memory usage, the storage space, and the CPU utilization. Managing CPU utilization was discussed in the previous section about threading, but the database management system will need to handle multiple processes accessing the data simultaneously. This may or may not include simultaneous writing to the database.

Alternatives

A popular database management system is SQLite. SQLite is commonly used for embedded systems and mobile devices which aligns with our target range of devices. PostgreSQL is a popular extensible database that can handle large numbers of simultaneous access and writes. Both are open source projects that are freely available for use. PostgreSQL began development back in 1986 making it the older of the two compared to SQLite which began in 2000. A third alternative is developing a custom database management system for all of the desired characteristics.

Analysis

SQLite and PostgreSQL are both powerful and popular choices as database management systems. Both provide the same service, but with different levels of complexity and resource usage. SQLite is designed to require very little storage on the device. It is designed to not be a client server approach to databases, instead writing directly to the disk. SQLite offers a smaller variety of specialized data types compared to PostgreSQL. This means that more general types would be used instead. SQLite can allow multiple queries from multiple processes at once, but only permits one to make changes at a time. PostgreSQL on the other hand is more of a classical client server database that excels at data integrity and handling larger numbers of queries and changes simultaneously. This comes at the cost of needing more resources, in particular memory to handle multiple clients at once. PostgreSQL offers many ways of adding functionality, but this also makes setup more complicated. This information was gathered from multiple independent sources as well as limited testing to verify common differences and comparisons. In regards to developing our own database management system, this would be the most challenging option. In theory this would be the most flexible solution and grant a level of freedom to solve any unforeseen complications. This advantage is less important when proper planning is done before. It is impossible to quantify this theoretical solution that does not exist yet, and it could not match solutions that have decades of development behind them.

Chosen Approach

To review, SQLite takes up fewer resources and offers a comparatively simple setup. PostgreSQL has the capacity to be more powerful and allow more connections to the database at once, but requires more resources to accomplish this. A custom solution would be the most difficult to develop, but support any possible data types.

Desired Trait	SQLite	PostgreSQL
Memory Usage	5	3
Supports Multiple Points of Access	3	5
Data Types Supported	3	4
Data Integrity in the Event of Error	4	5
Ease of Implementation	5	3

*Score out of 5

The above table quantifies how the three alternatives perform in different categories. SQLite and PostgreSQL are really close overall and often had qualities that each chose to prioritize and sacrifice. The most important factor is the memory usage and SQLite is the better option in this

category. Any shortcomings that it has like a comparatively limited set of data types can be solved by using more general data types. The flexibility that a custom solution could provide is outweighed by the amount of work needed to develop and optimize it compared to both of the other solutions. Currently SQLite appears to be the better choice with one caveat. SQLite does not support as many connections as PostgreSQL, which could be enough to change what database system we use. Currently we do not think it will be a limitation for the device, but it is worth remembering.

Proven Feasibility

Testing is necessary to verify that SQLite is the correct choice. This will take the process of prototyping a database to hold the required fields and simulating the environment. A test to see the limits of the number of changes and requests made will need to be done. For any data types not implemented, we will have to design the database to work with more general data types as needed. To develop a minimum viable product, benchmarks of how it performs on the Raspberry Pi as well as integrating it with other parts of the project will be done.

Technology Analysis: Database System and Language

Intro

Our database system will need an efficient programming language. Database systems run on various different programming languages such as C, Java , Python, Perl, Javascript, and etc. Most database systems can adapt to most programming languages, however some of them have more efficient libraries to work with and are much more feasible to create modular code. Some programming languages are more efficient than others and are more portable between multiple operating systems.

Desired Characteristics

The project we are working on will be using a Linux based system and we will be using threading in our program as well. We want to use a language that most of our team is familiar with and have had lots of experience in as well to come up with the most optimal solution. With that said C and C++ are the most commonly used languages in most modern database systems people have nowadays. Using C could benefit us in the long run since we are using Linux and are integrating multiple threads within our project. We also feel as if we can write much more modular code for our database by using this programming language over using other languages.

Alternatives

Some great alternatives other than C would be C++, Java, or maybe React. Depending on how C works out for us will determine this but from our research C and C++ would be the most optimal programming languages to use for this project. Overall there are several alternatives we could use though if we think the first language we go with is not optimal. C++ is another viable option because we can use another language to wrap around the C interface.

Analysis

Overall there are several pros and cons for each language and how it can benefit being used in a database system. Overall the ones that mostly stood out during research were C and C++ being used in database systems. Below is a brief pros and cons list comparing C, and C++.

C		C++	
Pros	**Cons**	**Pros**	**Cons**
Free Entities that C++ lacks	Insufficient error handling	Object oriented	Uses manual memory management
Team has more practical experience	Procedural	Uses namespaces	
Effective libraries available		Sufficient error handling	
Easy to compile with Linux			

Chosen Approach

We've decided to go with C seeing that it is the most compatible with our project and our whole team has experience in it. Also it would be more efficient to go with the threading that will be implemented using POSIX threads.

Proven Feasibility

Overall this programming language we have decided to go with may be the most compatible with the chosen database we have chosen which is SQLite. We may end up working on the project and realizing that we may need to change the chosen programming language but for now we are going with the C programming language for our database language. The one way may end up deciding this is by how we implement the libraries that the languages have and utilize them when we are coding.

Technology Analysis: Network Protocols

Intro

In order to actually command and control multiple embedded chips, information will need to be transported and in order to do that we will need to use a transport layer protocol. We will be using one of the two most commonly used transport layer protocols TCP and UDP.

Desired Characteristics

For our solution to be feasible we must have the fastest most efficient transfer of data between components. This quick transfer rate will allow for many commands to be send one after another without degradation of performance in our system. Additionally, and perhaps for the same reason, we must have low congestion upon transfer, as well as modularity within that data transfer.

Alternatives

In Transport Layer Protocol there are really only two viable options.
TCP, or Transmission Control Protocol which is the more commonly used for most network traffic today. It has proven reliable and has stood the test in the history of the internet, however brief that may be.
UDP, or User Datagram Protocol is the less commonly used of our two explorations, and was also developed nearly ten years after TCP.

Analysis

TCP could be seen as the correct option in many cases, it ensures delivery of packets and if one or many have been lost somewhere along the line, it will attempt to resend. It accomplishes this via a “handshake”. When the two systems of a network wish to connect and exchange data, this handshake must be exchanged in order for any networking to occur.

However UDP, does not share this handshaking trait and is as a result, faster as far as data transfers goes. Although it is faster there is potential for packet loss under unwelcome circumstances since there is no verification between sender and receiver.

Suggested Usages

TCP	UDP
Remote configuration changes	Embedded device status updates
Requests to embdedded devices	Information updates to front end
Data Transmission to the front end	

TCP vs. UDP

Pros	Cons	Pros	Cons
Reliable Data Transfer	Slower Data Transfer	Quick Data Transfer	More Potential For Lost Packets
Less Packet Loss	Lots of Overhead	Less Congestion Issues	Less data Verification
	Possible Congestion Problem	Connectionless	
	Not as Scalable		

Chosen Approach

UDP for transfer of data to and from embedded chips will allow us to request and receive updates from the device without worry about congestion and connection maintenance. The occasional lost packet will not affect the embedded chips enough to justify the weight of TCP.

Proven Feasibility

Our command and control system will function similarly to DNS with one system receiving and requesting data from possibly hundreds of other systems. While TCP may seem like the obvious choice here due to the fact that speed is not a requirement it doesn't actually provide us with any more functionality than UDP. Like DNS our system can simply wait for a received response from a request made over UDP and then resend the request if no response is received. A TCP request would guarantee that a response is given but it would take more than double the time of a UDP request which could simply re-request if an issue was found.

Having established our use of UDP, we now dive into the research of integration of front end and back end, between database and GUI, and define how that relationship will function.

Technology Analysis: API and Web Application - Database Interface

Intro

Aside from the database we have chosen for our project, the next most important thing we must consider is how said database will interact with our particular web application, the glue that ties everything together, database, language, network protocols, and finally our application, which as a whole makes up the Command and Control System. This web application interface must be dynamic, working on a variety of different devices, be fast, scalable, and able to communicate with the database in the most elegant way.

Desired Characteristics

For this project, we must create a fast, scalable, intuitive web application interface to communicate with our database 24/7. It must automate basic operations, such as storage and transfers, control basic input-output operations, contain a remote control interface for these operations to communicate commands to the database allow for complete control over all operations of the database.

Alternatives

In the team's research into this topic, we found only a handful of outside approaches that could potentially be molded to be optimally used in our project.

Firstly, the Eclipse Hono developed Command and Control API. We found this particular option just by searching around for any API that might fit our needs for this project. This product is used for some Command and Control systems, albeit seemingly less complex than our desired product, and is mainly used to send and receive queries and responses, and to control the state of actuators in certain applications.

Next, we found the SmartSDR Command Line API, developed by FlexRadio. We found this one while looking up general information about the underpinnings of SDR's and were immediately intrigued. This as we found, is used solely for FlexRadio's SmartSDR engine.

Lastly, as a default directive we also considered the idea of building our own tool to support these actions. Being developed from scratch in house by the team would provide the easiest way to truly customize every single part of the web application-database interface.

Analysis

The Eclipse Hono Command and Control API was the most relevant of the “out of the box” options, and was promising upon first discovery and a cursory reading on eclipse’s website. It had all the capabilities we would potentially need, from command sending, to automated configuration, sending request / respond commands, and even came nicely prepackaged, ready to go upon download or, “out of the box” as it were.

The option that we came across during this research period was the SmartSDR Command Line API. At first this seemed too good to be true, an API specifically for software defined radios, however, it was indeed too good to be true. The true nature of this specific API is really for debugging FlexRadio’s SmartSDR engine software, which we would not be using, so however insightful it was to see an API dedicated specifically to SDR’s we discarded this one rather quickly.

After these two we began to really dig into the requirements we had been given by General Dynamics, and started to explore the idea of building everything we needed for this portion of the project from scratch, in house. This idea seemed appealing to us as we could customize it every which way we wanted, and truly make it fit to the exact specifications from General Dynamics.

Chosen Approach

After looking at all of our options, (only the ones listed above were worth discussing in this document, however there were other alternatives that we came across, however briefly) we came to the conclusion that while there are certain API’s that we could potentially mold and shape to what we need, in the long run it would take too many resources, for the perhaps subpar result they would deliver. Because of this the decision has been made unanimously by the members of the team that developing our own tool, although not necessarily an API, to accomplish our tasks would give the best result based on the scope of our project and the initial requirements set forth by the General Dynamics Mission Systems team.

Desired Trait	Command and Control API	SmartSDR Command Line API	In-House Solution
Customization	2	1	5
Ease of Use	3	3	4
Speed	3	3	4
Database Connection	3	2	5

*Score out of 5

As you can see from the table above, comparing the traits we thought important to the overall success of this project, in this particular area of development, an in house solution wins across the board. If we take advantage of the ability to start from scratch we can control exactly what is in this solution, we can completely customize our GUI, and custom select network protocols for various tasks based off of the inherent performance needs of our project, ensuring the most efficient connection and interfacing between the application and our database making for the fastest, most user friendly option.

Proven Feasibility

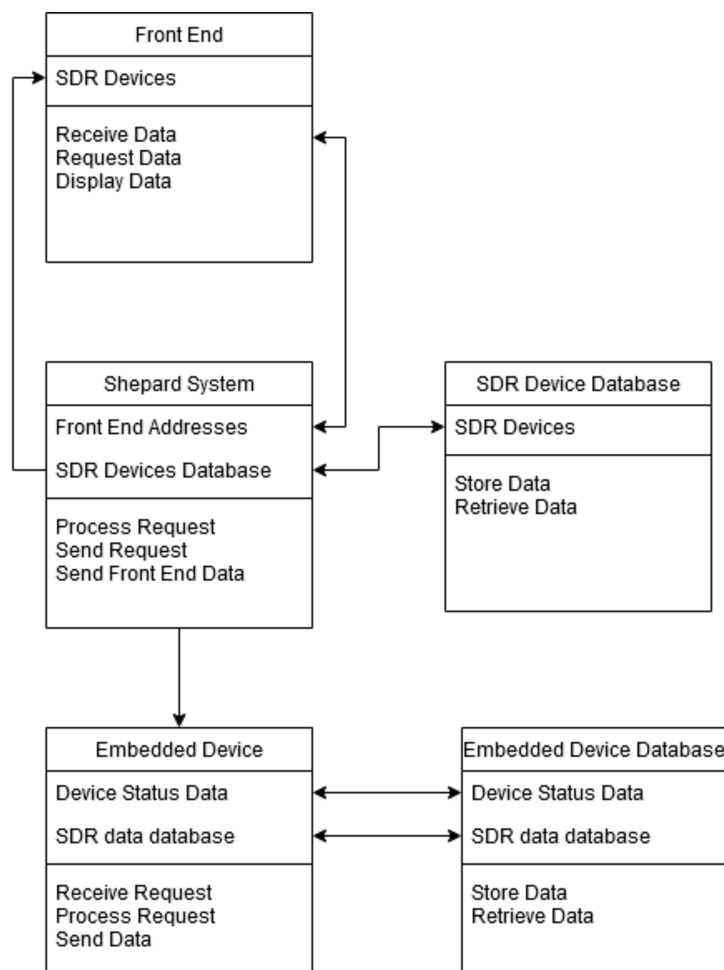
In the coming weeks, as we gather more and expand upon previously set forth requirements, we plan to begin building this solution, and as soon as possible, getting it connected and interfacing with our chosen database. From there we can perform tests sending information from the database to the application, and commands from the application to the database, and confirm our hypothesis that building this tool in house was and is the best option for our situation.

For any Command and Control solution, all parts must integrate into each other, and function both independently to do individual tasks, and as a whole, do accomplish an over all goal. The following section will detail how we intend to make the five major areas you've just read about above work together in such a way.

Technological Integration

As stated above, for a solution like this to work, as complex as it is, all parts must seamlessly integrate with one another, and be able to function independently and together as needed by even the most basic functionality of a Command and Control System.

It should be clear at this point that that the entire success of our solution is predicated upon the efficient execution of database commands on our database, and the ability of our application to effectively and remotely control the SDR. The SDR will need to communicate with our database effectively and in turn the database will need to communicate with our web application quickly and efficiently, which will require the careful selection of networking protocols based off of a given task that must be accomplished. Once the data reaches the application we will build it will need to be simultaneously displayed, sorted, and stored in various different file formats depending on the data types. The application will need to be built to run highly efficiently, and create an intuitive way to interact with our system. By careful use of threading, and proper choice of programming language, working in parallel with an efficient database and a well built application we believe that our proposed solution will succeed on every level and fulfill every requirement that we have been given by our client. As you can see in the diagram accompanying this page, we have created a document that details how the interworking of our envisioned solution will work based off of the information we have been given and the research we have done up to this point.



Frontend - Device Communication

We selected JSON as the markup language to transmit data between the frontend/command and control system and the simulated SDR device on the Raspberry Pi.

One of the key issues with the current implementation listed in the provided project proposal was that XML is cumbersome and heavily formatted which has slowed down data transfer in the past. Other markup languages like YAML were considered but JSON was still picked over them since JSON was the slimmest choice. YAML had features that we would not make use of and would only serve to bloat the transfer of data. Additionally not as many people (including our team) are familiar with YAML as they're are with JSON which would make it more difficult to support in the long term.

Conclusion

As technology continues to evolve, and the needs of everyday embedded systems continue to become more complex, especially as SDRs become more prevalent we must strive to make our embedded system run on less power consumption, require less memory, and efficiently and effectively execute its desired operations. Throughout the duration of this document we have covered what we consider to be the five major areas that the success of this project will rely on. We believe that the use of POSIX threading, working to maximize functionality to and from our SQLite Database, in combination with our use of UDP for successful data transfer to our web application, we will provide a highly efficient, optimally configured, nominally functioning whole Command and Control system to provide proper control over a software defined radio.

Going forward we intend to continue to discuss requirements with our client, General Dynamics, to further establish realistic and necessary requirements that will shape this project in the coming months. We will begin to build a demonstration of our envisioned solution implementing all of the technologies we have chosen earlier in this document, and through this building, we will perform extensive testing, working in parallel with General Dynamics to verify our testing results and gain further insight to build the best product we can with the time and resources we have been given.