Applying Blockchain to Energy Delivery Systems

FINAL REPORT

Team: sdmay20-12
Client: Grant Johnson
Adviser: Manimaran Govindarasu

Team Members:
Anthony Cosimo - *Test Engineer*
Jacob Dawson - *Project Manager*
Keegan Bloedel - *API Architect*
Katherine Ringgenberg - *UI Architect*
Steven Rein - *Blockchain Architect*
Dakota Moore - *Cybersecurity Manager*

sdmay20-12@iastate.edu
https://sdmay20-12.sd.ece.iastate.edu

Revised: 26 April 2020

Executive Summary

Engineering Standards and Design Practices

- Agile Software Development
- Test-driven Development
- Continuous Integration and Development
- Peer Reviews
- NASPInet Standard

Summary of Requirements

- <u>API</u>
    - API shall expose Smart Contract functions to Web Requests from the User Interface
    - API shall still be functional when the Blockchain network is down
- <u>Blockchain Network</u>
    - Blockchain Network shall have a minimum of five nodes
    - Blockchain Network shall have multiple orderer nodes
- <u>Smart Contract Layer</u>
    - Functions within the Smart Contract Layer shall be able to create and query data stored on the ledger
- <u>User Interface</u>
    - Allow user to display Blockchain performance data
    - Allow user to query data stored in the Blockchain network
    - Allow user to issue commands to the Publishers to start and stop the stream of data to the Blockchain Network.
    - Shall remain functional when the API is down
- <u>Publishers</u>
    - Publish PDC synchro-phasor data to the blockchain by parsing information out of TCP packets from a pcap file.
    - Must have the ability to start and stop publishing through messages received.
- <u>System-level</u>
    - Each system shall be fault tolerant to other systems.
        - For example, if the Blockchain Network is down, the API shall return an appropriate error code to the UI, indicating that the Blockchain Network is down. The UI will then display a message notifying the user of the Blockchain Network being down.
    - Each component shall be deployed on Linux virtual machines made available through PowerCyber resources.

Applicable Courses from Iowa State University Curriculum

- COM S 309: Software Development Practices
- COM S 319: Construction of User Interfaces
- S E 329: Software Project Management
- S E 339: Software Architecture and Design

New Skills/Knowledge acquired that was not taught in courses

- Knowledge of HyperLedger Fabric and other HyperLedger technologies
- Knowledge of Blockchains concepts applicable to the problem domain
    - Orderers
    - Self-Signing Certificates
    - Peer interactions
    - Raft Consensus
- Understanding and implementation of Docker tooling for containerized applications

Table of Contents

1	Introduction	6
1.1	Acknowledgement	6
1.2	Problem and Project Statement	6
1.3	Operational Environment	7
1.4	Functional Requirements	7
1.5	Non-functional Requirements	8
1.6	Intended Users and Uses	9
1.7	Assumptions and Limitations	10
1.8	Expected End Product and Deliverables	11
2	Specifications and Analysis	12
2.1	Proposed Design	12
2.2	Development Process	15
2.3	Design Plan	15
3	Statement of Work	16
3.1	Technology Considerations	17
3.2	Project Tracking Procedures	18
3.3	Project Timeline	19
4	Testing and Implementation	21
4.1	Process of Testing	22
4.2	Interface Specifications	22
4.3	Hardware and Software	23
4.4	Functional Testing	23
4.5	Non-Functional Testing	24
4.6	Results	25

5 Closing Material 28
5.1 Conclusion 28
5.2 References 28
5.3 Resources 28
6 Appendices 29
6.1 Operation Manual 29
6.2 Previous Design Versions 31
6.3 Project Code 31

List of Figures

Figure 1: Use Case Diagram
Figure 2: Architecture Diagram
Figure 3: Level 0 Diagram
Figure 4: API Level 1 Diagram
Figure 5: Blockchain Level 1: Diagram
Figure 6: UI Level 1: Diagram
Figure 7: Gitlab Issues example
Figure 8: Gantt Chart
Figure 9: Process of testing
Figure 10: NASPInet Conceptual Architecture Diagram
Figure 11: A Conceptual Architecture Diagram For Our Blockchain
Figure 12: Bar Chart Comparing Block Timeout Configurations
Figure 13: Bar Chart Displaying Publishing Improvements

1. Introduction

1.1. Acknowledgement

Our team would like to thank and acknowledge PowerCyber Labs for allowing us to use their compute resources, Manimaran Govindarasu for advising our project, and Grant Johnson for being the client to our project.

1.2. Problem and Project Statement

1.2.1. Problem Statement

Energy Delivery Systems are deployed in an environment that is geographically distributed utilizing public internet infrastructure for communications. The integrity of measurements, commands, and authenticity of control devices performing communication are critical for trusted operations.

1.2.2. Proposed Solution

The purpose of this project is to develop a blockchain Energy Delivery System solution to solve the problem statement. The project is driven by the need described in the problem statement above, to create a secure network for a geographically diverse Energy Delivery System. To help mitigate security concerns, the use of blockchain technology would provide a secure network for managing Energy Delivery Systems by removing the dependency on a central service. Additionally, a permissioned blockchain system ensures a higher level of security when concerned with cyber attacks similar to a fifty-one percent attack. With this project we hope to deploy a permissioned blockchain system for

interaction between devices on an Energy Delivery System and users managing those devices.

1.3. Operational Environment

The operating environment for this project is servers that will store the project and other data we may use, such as simulation data. Therefore, it should not be subject to any harsh weather conditions, and the servers should be properly maintained. These servers will be linux-based and deployed on a network that will allow for secure permissioned communication between all nodes.

1.4. Functional Requirements

1.4.1. API

- The API shall expose the Smart Contract functions to Web Requests from the User Interface and/or Devices.
- The API calls shall be authenticated to the Blockchain Node System.
- When the Blockchain network is down the API shall return an appropriate error code indicating the loss of the Blockchain Network.

1.4.2. Blockchain Network

- The Blockchain Network shall consist of at least five organizations with two nodes each for transaction consensus.
- The Blockchain Network shall consist of multiple nodes to act as orderers.
- The Blockchain Network shall use Raft consensus for the ordering service and the deterministic consensus algorithm.

1.4.3. Smart Contract Layer

- The Smart Contracts shall implement the blockchain functions to read, update, delete, and query data stored on the ledger.
- Users that are assigned to a channel are the only users allowed to utilize the Smart Contracts functions made available to that channel.
- Each Smart Contract shall define more than one endorsing node for consensus.

1.4.4. User Interface

- The User Interface shall be rendered in a modern web browser.
- Given the API is down the User Interface shall remain functional.
- A user shall be able to view the results of metrics and measurements they had previously requested.
- A user shall be able to start and stop the flow of data to the Blockchain Network.

1.4.5. Publishers

- Three Publishers shall run, each publishing to a single blockchain organization.
- The Publishers shall send synchro-phasor data to the blockchain. The synchro-phasor data should be retrieved from a PCAP file.
- The Publishers shall have the ability to be stopped and started from the User Interface.

1.4.6. Operational Environment

- The blockchain network shall run on a linux-based virtual machine provided by PowerCyber resources.
- The API shall run on a linux-based virtual machine provided by PowerCyber resources.
- The User Interface shall run on a linux-based virtual machine provided by PowerCyber resources.
  - The User Interface should be interacted with using a modern web browser, Chrome will be the supported browser.

1.5. Non-functional Requirements

1.5.1. API

- The API shall produce query responses in under 10 seconds.
- The API shall have continuous integration and automated deployment.

1.5.2. Blockchain Network

- Each node on the network shall be run using Docker containers to allow for reliability when run on different types of machines.
- The Blockchain Network shall be deployed to PowerCyber resources by implementing a CI/CD pipeline.
- The Blockchain Network should be able to run in a Linux-based system via the services provided by Amazon Web Services, however this is not a direct stipulation by the client, just a potential for additional complexity if needed.

1.5.3. Smart Contract Layer

- The Smart Contract Layer shall have automated deployment and automated testing.

- The Smart Contract Layer shall make updates to the Blockchain Network.
- The Smart Contract Layer controllers and models shall contain documentation that describe their purpose and intended use.
- The Smart Contract Layer shall have unit tests for all controllers.

1.5.4. User Interface

- If the blockchain network is not available, the User Interface shall display loss of communication to the user.

1.5.5. Maintainability

- The documentation made available through the project's wiki is descriptive enough for the client to understand how to modify the project properly when needed.
- When the client wishes to modify the project, the project shall not be difficult to modify.
- Continuous Integration and Continuous Deployment can run all tests and deploy to the necessary environments as needed by the client.

1.6. Intended Users and Uses

Our Energy Delivery System has two main types of users: human users and devices. The use cases of these users consists of the following:

1.6.1. Querying metrics and measurements from the blockchain system. This is done by the human users.

1.6.2. Periodically post updated measurements. This is done by the devices, although there is potential for the human users to make changes to the domain specific data, in cases of errors with metrics or measurements.

These use cases and users may have overlapping interactions with the system as mentioned, but generally the interactions consist of the primary user in each use case, as mentioned above.

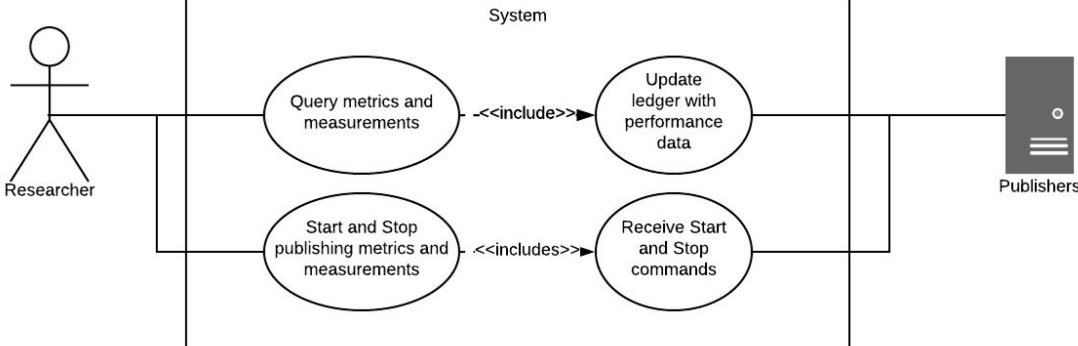

Figure 1: Use Case Diagram

1.7. Assumptions and Limitations

1.7.1. Assumptions

- Server hardware and operating system environment are made available through PowerCyber, and these resources are sufficient for the development.
- We are provided access to devices that use PowerCyber.

1.7.2. Limitations

- There is no budget for the project, thus, we are constrained to using PowerCyber resources.
- If the PowerCyber resources are found to be insufficient, the sponsor will either arrange for different resources or modify the scope to work with the existing resources.
- Hyperledger Fabric must be used as the permission-based distributed ledger framework.
- All Hyperledger Fabric related code must be written in JavaScript.

1.8. Expected End Product and Deliverables

1.8.1. Utility for Publishing Data To Blockchain Network (Publisher)

There will be a utility for publishing data to the Blockchain Network. The Publisher will be publishing syncho-phasor data to three organizations within the Blockchain.

1.8.2. A Fully Functional Blockchain Network

There will be a Blockchain Network that is running within Virtual Machines within the PowerCyber network. The nodes in the Blockchain Network will help to endorse information updates to the ledger, provide immutability of the data within the ledger, and be fault tolerant to losses of nodes within the network. Communication with these nodes will be done by a multi-node ordering service. Unlike other blockchain systems, the orderer will allow our data to be deterministic instead of probabilistic.

1.8.3. Authenticated Call from API to the Blockchain Network

There will be an API that can utilize the Smart Contracts in order to create, read, update, delete, and query data available on the Blockchain Network. All calls to the Smart Contracts from the API shall be authenticated and verified by the allowed participants in each channel. The Smart contracts will be able to receive data from the nodes along with updating the data on the nodes.

1.8.4. Web-based User Interface

The web-based user interface will contain web pages that will allow users to query and view metrics, view blockchain frequency statistics, and view statistics based on the metrics stored in the blockchain. The user interface will also give the user the ability to register blockchain users and start and stop blockchain publishers for the different organizations.

1.8.5. Project Documentation

There will be an “operation manual” that lives within the project repository that describes how to deploy updates to the Smart Contracts, Blockchain Network, API, and Web-based User Interface. This manual will also have documentation regarding the development environment setup and required dependencies. This is close to the same manual that can be seen in Appendix I.

1.8.6. Continuous Integration and Deployment Infrastructure

There will be a CI/CD solution in place for the project. Continuous Integration will run on all merge requests and will run on the master branch weekly to ensure that the master branch remains clean. Continuous deployment will be available for the blockchain network, API, and web-based user interface.

2. Specifications and Analysis

2.1. Proposed Design

The software solution consists of the following components: a User Interface, an API, and a Blockchain Network. The API receives requests from the User Interface to view performance data and issue operator commands to PowerCyber devices. The API will authenticate those requests with the Membership Service Provider. After a request has been authenticated, the API will make a request to run a smart contract on the Blockchain Network that either queries performance measurements or updates operator commands for the PowerCyber devices.

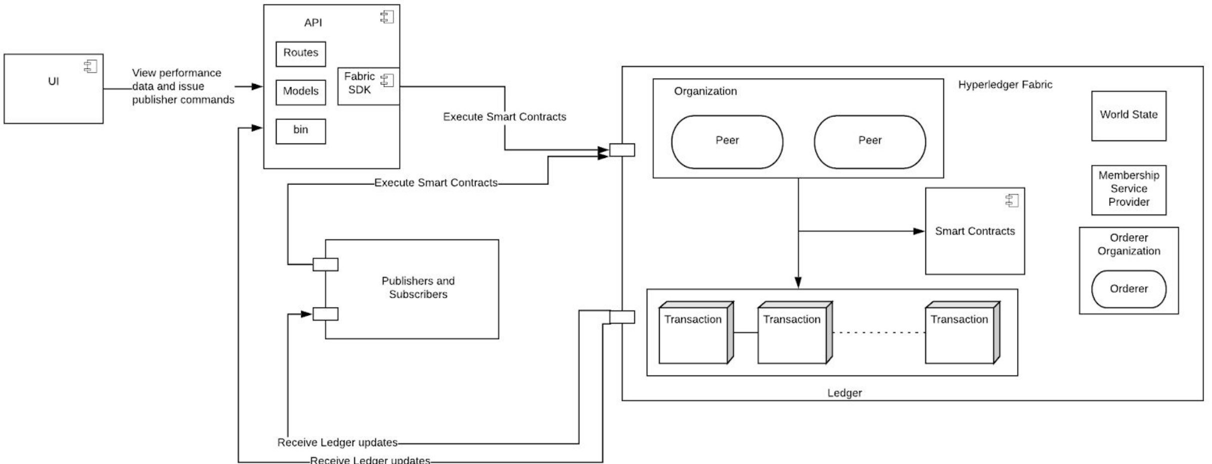

Figure 2: Architecture Diagram (above)

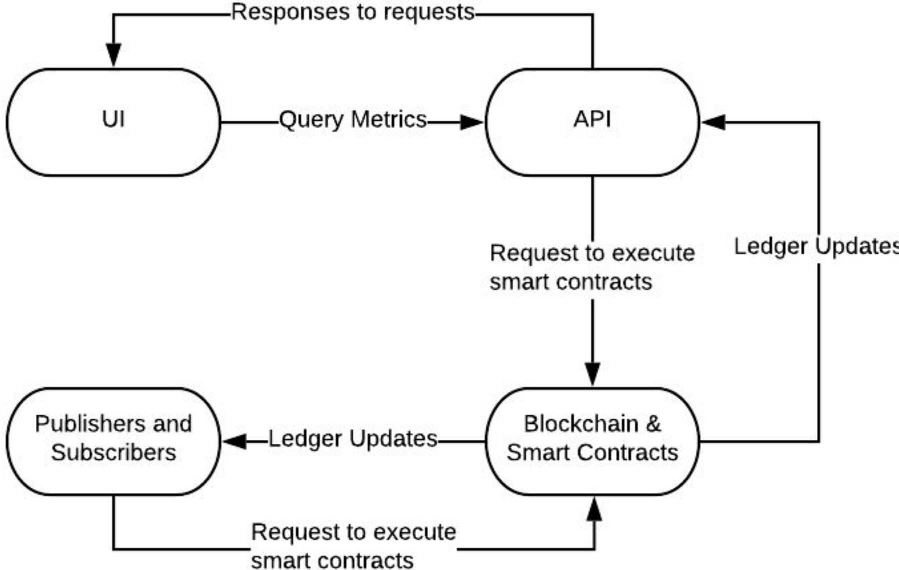

Figure 3: Level 0 Diagram (above)

2.1.1. API

- The API will run smart contracts when new performance data has been received by a PowerCyber Device.
- The API will run smart contract when an authenticated user requests to view performance data.

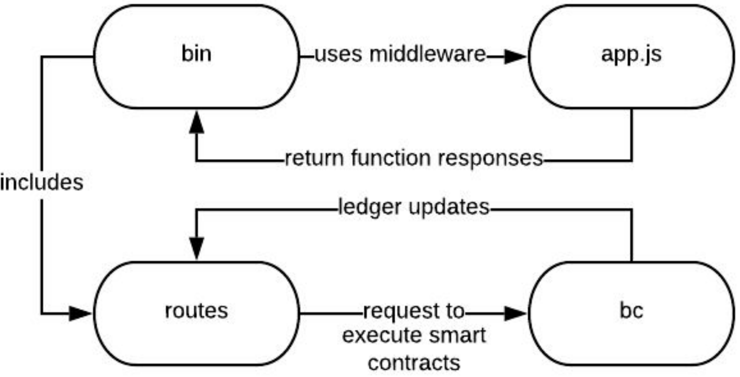

Figure 4: API Level 1 Diagram

2.1.2. Blockchain Network

- The Blockchain Network will report ledger updates to the API.
- The Blockchain Network will receive requests to run smart contracts from the API.

- Given that the entity who has requested to run a Smart Contract is authorized for that requested data, the Blockchain Network will run that Smart Contract in order to update the ledger.

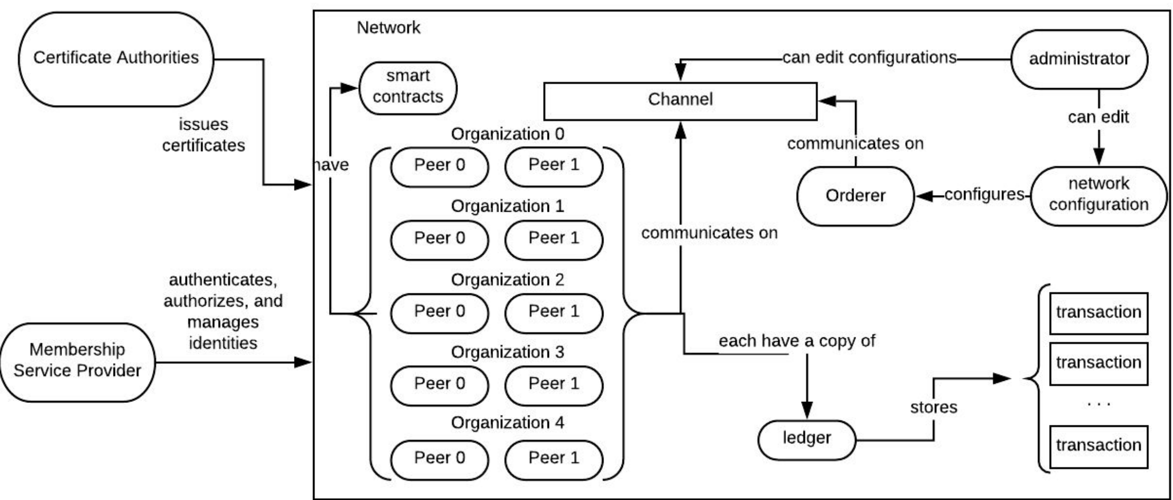

Figure 5: Blockchain Network Level 1 Diagram

2.1.3. Publisher

- Given that the Publisher is authenticated with the Blockchain Network and has proper permissions, the Publisher will be able to post data to the Blockchain Network.
- Given that the Publisher is running, the Publisher will be able to be stopped.
- Given that the Publisher is stopped, the Publisher will be able to be started.

2.1.4. Smart Contract Layer

- Given that the entity who is requesting to query entries via a Smart Contract is authorized to do so, the Smart Contract Layer will be able to query entries in the Blockchain Network.
- Given that the entity who is requesting to create an entry via a Smart Contract is authorized to do so, the Smart Contract Layer will be able to create an entry in the Blockchain Network.
- Given that the entity who is requesting to update to an entry via a Smart Contract is authorized to do so, the Smart Contract Layer will be able to create an update to an entry in the Blockchain Network.

2.1.5. User Interface

- The User Interface will allow a user to request PowerCyber Device metrics.

- The User Interface shall allow the user to create a new blockchain user.

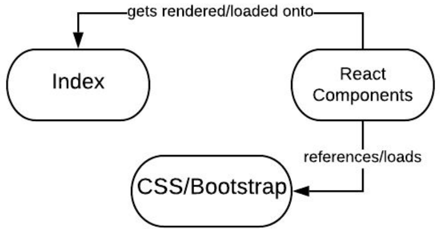

Figure 6: UI Level 1 Diagram

2.2. Development Process

Our team used an agile approach to the project. We used two-week sprints with a retro and demo at the end of each sprint as necessary. For all merge requests, there were tests that accompanied all new work introduced into the master branch. We followed a Test-Driven Development process for the project that helped us accompany all new work in merge requests with tests. All merge requests were required to pass all component and system-level tests, and be reviewed and approved by at least one other member on the project.

2.3. Design Plan

In order to determine if utilizing a blockchain network is a feasible solution for maintaining the integrity of communication between control devices, we first had to determine the optimal configuration for the blockchain network. Our blockchain network needed to handle at least 60 frames/second since PDCs produce measurements at a rate of 60hz. Eventually, we arrived at a blockchain configuration with a Batch Timeout value of 10ms. The Batch Timeout value is the amount of time the ordering service will wait before validating a batch of transactions. With this low value, we decreased latency. Additionally, we designed the chaincode to accept a list of measurements. By doing so, devices publishing to the network could concatenate results through some form of middleware service between the devices and the blockchain network. A deeper explanation of how the configuration and design decisions were made can be found in the 4.6 Results section.

Then, we needed a way to show the PMU data and blockchain network performance to the user. We decided to implement a user interface that shows graphs depicting the frames / second being published over time, a section for displaying the recent phasor values for specific PMUs, and any errors with published data. In order to display this information to the user, we decided to implement the API and UI. The API processes requests from the UI and makes

requests to Smart Contracts using the Hyperledger Fabric Node SDK when necessary.

At this point in the project, we needed to simulate a PDC publishing measurements to the blockchain network. To do this, we decided to implement the Publisher component. This component decodes synchrophasor requests found in a PCAP file. These requests include data frames and configuration frames that we use to properly decode PMU data. Then, we aggregate 60 frames of PMU data into one request per second to the blockchain network.

After implementing the Publisher, we needed to have some notion of acting as a subscribed entity to this PDC data. We decided to implement a Subscriber component. Before a Publisher begins to publish PDC data to the blockchain network, the Publisher establishes a web socket connection with the Subscriber component. Each time a request is made to the blockchain network, the Publisher sends a message over the web socket connection containing the ID of the PDC data just published. The subscriber can then query the blockchain network using that new ID. Thus, allowing a subscriber to receive new records as soon as they are published to the blockchain network.

3. Statement of Work

3.1. Technology Considerations

HyperLedger Composer, a tool used for building a blockchain network and implementing smart contracts using HyperLedger Fabric, has recently been deprecated. To work around this, we looked into an alternative called Convector, supported by Hyperledger Labs. Convector provides similar functionality that Composer provides with some additional features such as an API Server Generator, configuration file generation, Smart Contract boilerplate code, etc. Also, convector allows the user to make configurations to a Definitions JSON file for exposing Smart Contract functionality to an API. After further exploration of HyperLedger Convector, we determined that our use case should be fully implemented without this technology. By not using Convector, we had more autonomy in our design and was able to define the network, smart contracts, and API on our terms.

HyperLedger Fabric was used for configuring and deploying our blockchain network. HyperLedger Fabric utilizes YAML configuration files for structuring and setting up the network. HyperLedger Fabric also uses Docker Composer for defining, creating, and deploying the containerized nodes to ensure all environment requirements are met. HyperLedger Fabric works using nodes that exist as the central communication for the network. These nodes ensure consistency is maintained with the state of the Ledger. HyperLedger Fabric has a

key value database called CouchDB to deal with transaction logs and ledgers (NoSQL document store). CouchDB is the default database for HyperLedger Fabric, therefore the best choice when working with Fabric.

For development of our request API and Publisher there are many technologies on the market to consider. With that in mind, there are three web frameworks the team has looked into. Django, written in Python, provides rapid development and a plethora of tools for development for teams. Flask, also written in Python, provides a simple and flexible developer experience. Comparing the two, Flask is most likely preferred if the focus is gaining experience and learning. Django would be preferred if the focus is on the final product and maintainability, being the older of the two. The final web framework we have considered is Node.js. The benefit of Node.js for our team is that we are already doing some programming in JavaScript through other components of the project. Therefore, the team has decided to move forward using Node.js for the API and Publisher.

For the web user interface we used ReactJS. The team decided upon this to give ourselves the opportunity to work with a framework that is growing within the developer community. Additionally, ReactJS is primarily used for building single page applications, which fits our expected solution for the web user interface.

3.2. Project Tracking Procedure

The team used a Gantt Chart to track progress. This Gantt chart was updated at the end of every sprint, and was primarily for the visibility of those outside of the team. We used the Gantt chart for a high-level view on project tasks, relationships between tasks, and milestones. We also used Gitlab Issues to break up tasks on the Gantt chart into smaller tasks. We chose to use Gitlab Issues due to its ability to produce burndown charts and allows us to use custom formats for issue tracking. We used these issues to track individual and team progress throughout the project. These issues include a title, description on what will be worked on, due dates, labels, and assigned members. When an issue is completed, it is marked as so and archived, leaving a detailed list of what was worked on for the project. Looking at the example below, we can see that Gitlab provides the ability to not only explain what the specific task will implement, but also displays a record of changes to the issue.

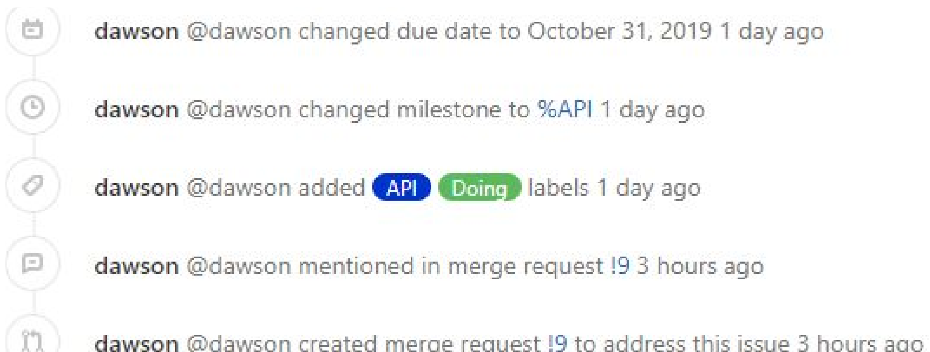

Figure 7: Gitlab Issues example

3.3. Project Timeline

For the majority of section 3.3, when stating that an implementation of a feature will be happening, this suggests that the implementation will include the necessary source code, along with the implementation of any appropriate tests e.g. unit, integration, smoke.

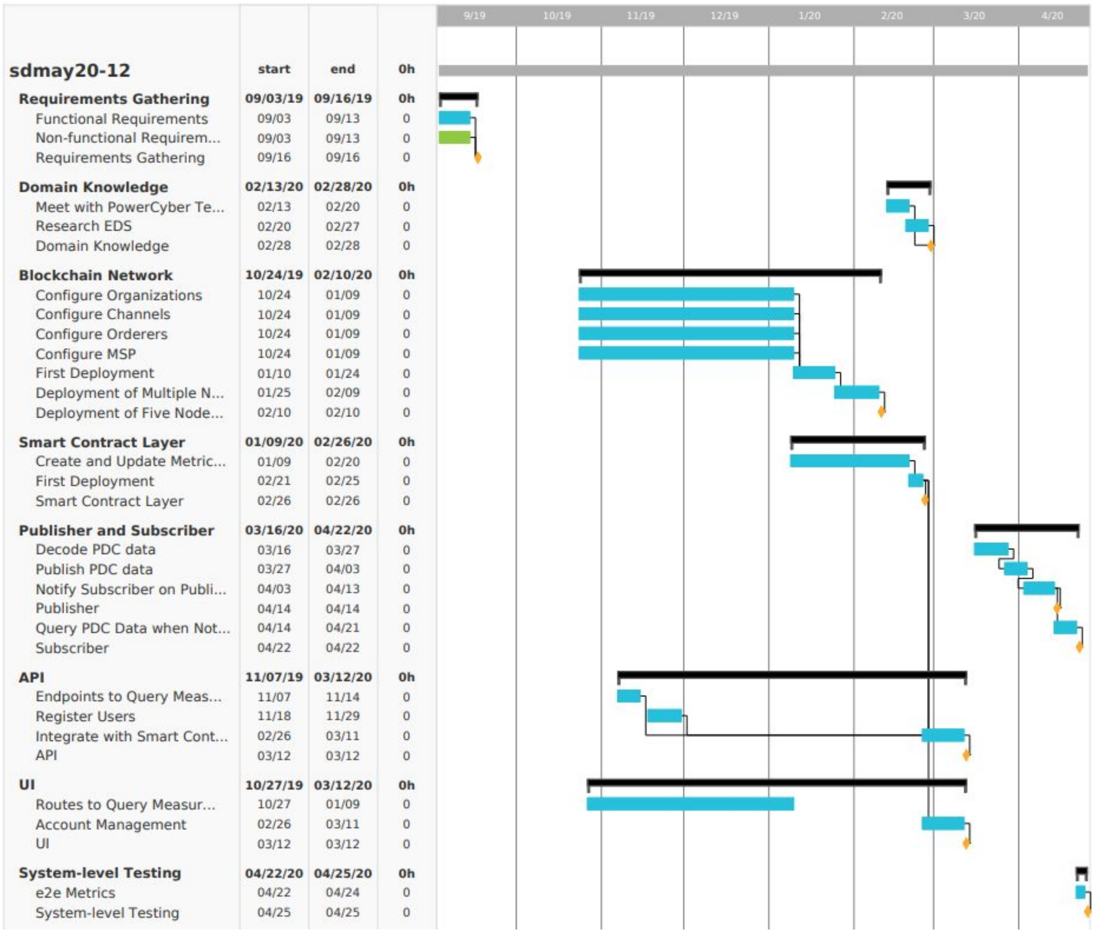

Figure 8: Gantt Chart

Implementing a Blockchain Network took the longest compared to other groups of tasks due to the team's inexperience with the technology and gaining access to the resources for hosting the Blockchain Network.

4. Testing and Implementation

4.1. Process of testing

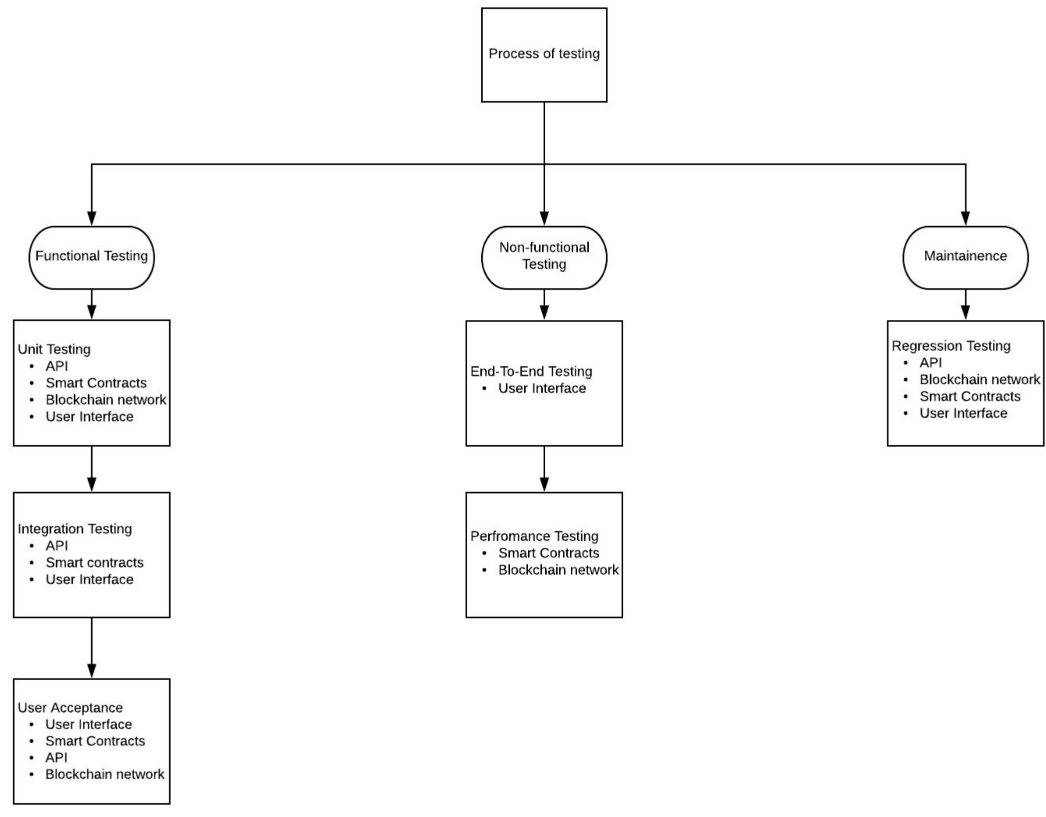

Figure 9: Process of testing

Figure 9 displays an overview of the process of testing. We followed each flow as we progressed forward in the development of our project. For each flow, we implemented the testing in the first block before moving on to the next and so on. This way we were able to break down our testing into concise parts to know what needs to be accomplished for our testing.

4.2. Interface Specifications

All components within the software suite will have to agree on consistent models to represent performance data, types of devices to interact with, and operator commands. We base our model off a use case of NASPInet, a standard put forth by the DOE and NERC to standardize the infrastructure of synchro-phasor communication networks. We can look at the NASPInet Conceptual Architecture in Figure 10 and compare it with our blockchain architecture to get a better concept of the interface.

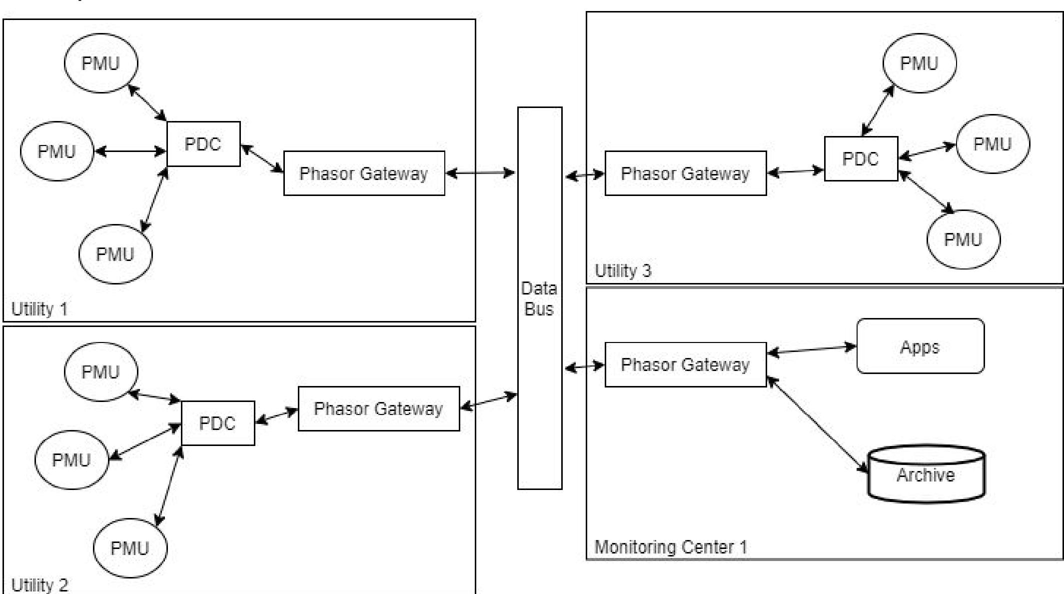

Figure 10: NASPInet Conceptual Architecture Diagram

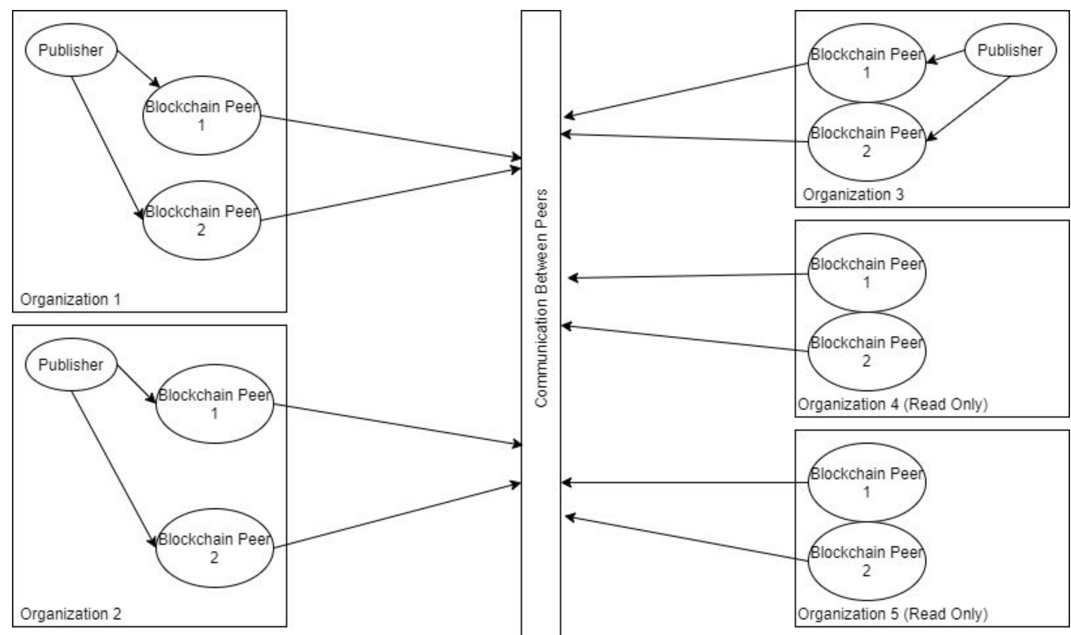

Figure 11: A Conceptual Architecture Diagram For Our Blockchain

The usage of blockchain technology itself represents most of the components that would be expected in the Phasor Gateway and Data Bus representation on the NASPInet, things such as authentication and data validation. In our network, the Publisher represents the PMU or PDC that would send data to the Historian, or the ledger within each peer in the team's project. By making Organization 4 and 5 read-only on the network, we can represent a Monitoring Center in the NASPInet that would not be publishing data to the network.

4.3. Hardware and Software

4.3.1. API

- Unit tests and integration tests for the API use the Jest Javascript testing framework. With this framework we wrote automated tests to make HTTP requests to our endpoints and unit test classes and functions.

4.3.2. Blockchain Network

- All new chaincode was added to a chaincode subdirectory and is tested when relaunching the Blockchain Network.
- Performance testing was done on the Blockchain Network using a tool created by the team. The tool follows a publisher subscriber pattern. The publisher will publish to the blockchain network, after which it will notify the subscriber that it can now make a query for the published data. If the query is successful, the subscriber will record the latency information in a CSV file. The results can then be analysed to check for performance increases from blockchain

configuration modifications. For this tool we used the JavaScript WebSockets library, for open communication between the publisher and the subscriber.

4.3.3. Smart Contract Layer

- We used the Jest Javascript testing framework for testing smart contracts.

4.3.4. User Interface

- For writing unit tests, acceptance tests, and end-to-end tests, the Jest Javascript test runner was used along with the React Testing Library. The Jest test runner allowed us to access the DOM via jsdom for testing React components and to use mocks. The React testing library allowed us to test React components without relying on their implementation details.

4.4. Functional Testing

4.4.1. Unit Testing

All individual testable tasks had their testing consisting of one or more tests for each component of the task. Each task that involves configuring or creating a part of the blockchain, smart contracts, API and UI had individual unit tests to confirm that the task is fully accomplished and that component can start to be safely integrated. The creation and integration tasks for the blockchain were tested to ensure the organizations, channels, orderers, and MSP are all configured correctly. This was done by testing the contents of each one individually on the running blockchain network. Smart Contracts were tested to ensure that the metrics data are tracked and updated properly throughout all contracts, as well as tested to make sure that the deployed smart contracts are interacting properly with the blockchain network. Each one of the API's endpoints were tested to ensure that they can be used to query the specified data.

4.4.2. Integration Testing

Integration testing of the blockchain network consisted of deploying the nodes for the network and testing that the blockchain network is running and it's able to accurately execute the smart contracts as well as update all nodes on the network with a new contract. The integration of the blockchain with the API was tested by running commands through the API to query, log in, and create commands/data that were tested against known data. The integration of the API with the smart contracts was tested by trying to execute a contract through the API and checking that it was properly executed. The Integration between the API and UI was

tested by using the UI to execute commands in the API and comparing what returns with known data.

4.4.3. System (end-to-end) Testing

The system testing consisted of testing the entire software from end-to-end. A test was done to ensure that a user can access the UI, query some data by having the UI use the API to execute said queryon the blockchain, and then comparing the result with what should've been accomplished.

4.5. Non-functional Testing

4.5.1. API

We used unit tests for query response times appropriate for each endpoint. For example, the query response time making a GET request to "/api/metrics/" will be much greater than the query response time for "/api/metrics/1". The expected query response time was different for each endpoint.

4.5.2. Blockchain Network

Testing the blockchain network can be a difficult task since most of what makes up the network is just configuration files. To do some minor integration testing with the blockchain network, we used a HyperLedger Fabric CLI (Command Line Interface) docker image. After the blockchain has been started and the chaincode has been installed and instantiated, we used the CLI to make some basic queries to test data in the chaincode, to ensure that a connection can be established with the network.
Performance testing was also done on the Blockchain Network, to find the correct configuration to match our designed latency and throughput.

4.5.3. Smart Contract Layer

We implemented an eslint rule to ensure that there is JSDoc for all functions and classes.

4.5.4. User Interface

We used automated acceptance tests put in place to simulate loss of communication between the User Interface and the API, and loss of communication between the API and Blockchain Network.

Unit tests for making unauthenticated requests via the API and to the Smart Contract layer were done to ensure that all requests must be authenticated.

4.6. Results

After research and investigation of the types of devices that would be publishing to our blockchain network, we found that a reasonable frequency of data being published would be 60 results per second.

After all components of the project (Blockchain Network, API, UI, and Publishers) were created, we recognized some serious performance issues. The Blockchain originally was only able to create transactions at a rate of about 0.4 Transactions Per Second.

To gather data on a solution, we developed a tool using a publisher-subscriber pattern for performance testing. This tool creates and sends a blockchain transaction, after this transaction is completed it sends a message to the subscriber that there is new data available, the subscriber will then query the blockchain network to retrieve that data. This allows us to test both publishing latency and up-down latency.

After running tests and adjusting blockchain configurations, we were able to increase blockchain performance from 0.4 Transactions Per Second, to almost 3 Transactions Per Second. Our discoveries with different configurations can be see below:

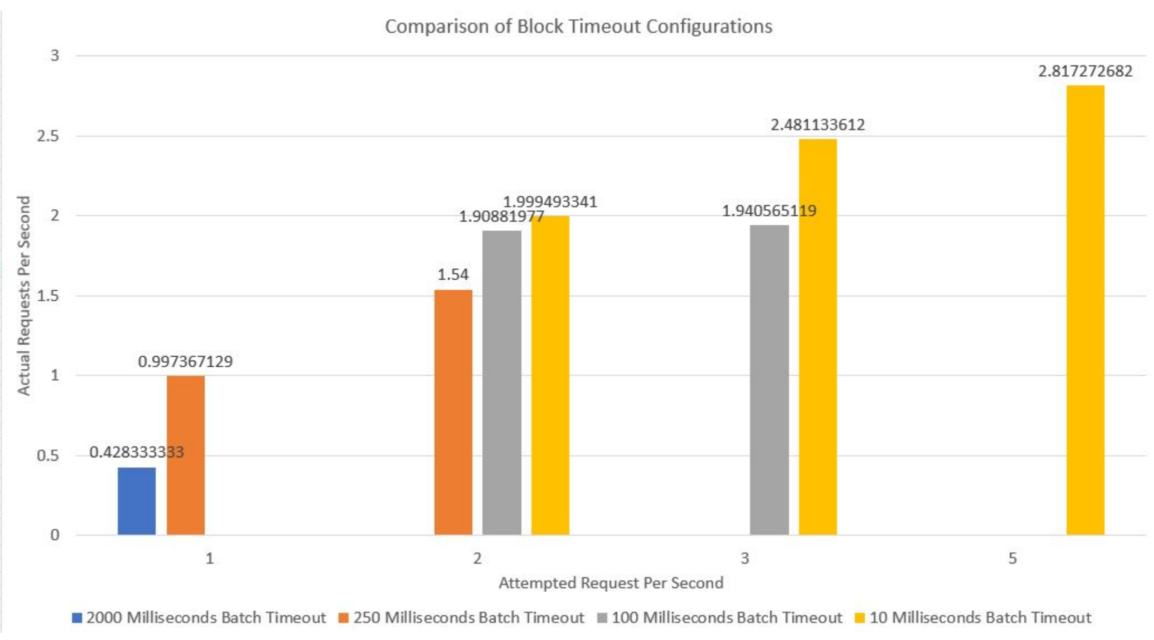

Figure 12: Bar Chart Comparing Block Timeout Configurations

Our goal was to get our publishing performance up to 60 results per second and currently our implementation of the Blockchain Network and the Publishers only allows for 3 results per second. To meet our goal, modifications were made to the testing tool to

allow multiple results to be sent in the same transaction, which required modification of the chaincode as well. Our discoveries with these changes can be seen in the graph below:

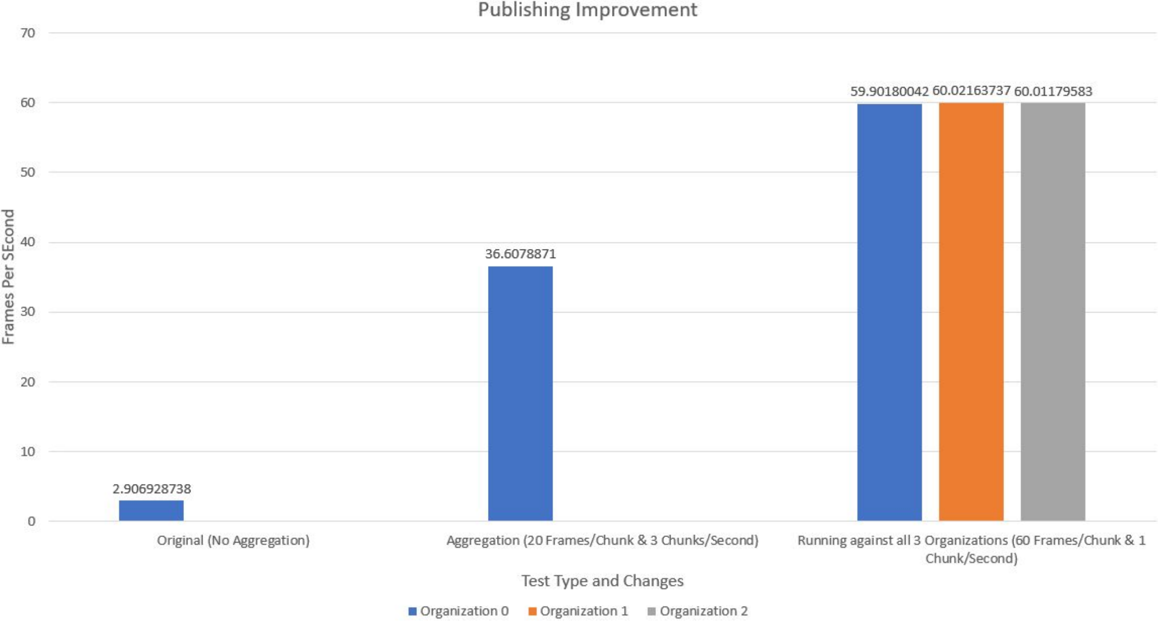

Figure 13: Bar Chart Displaying Publishing Improvements

The Original implementation of the tool for testing performance was able to publish just under 3 results per second without aggregating the data. After modifying the tool, a test was run sending 20 results with each transaction, and still performing 3 transactions per second. The results were much better, and we found that publishing to the database within the chaincode would be a much better solution than making multiple transaction requests.

In the final stage of testing, we modified the tool to create three publishing entities, with each publishing to a different organization. In this test we sent 60 results with each transaction and only performed 1 transaction per second. We expected running three different publishing entities to affect performance, but the results above were exactly the performance we had hoped for. All three organizations were able to keep up with the publishing rate of 60 results per second, with the only issue being that our virtual machines ran out of space.

5. Closing Material

5.1. Conclusion

Reflecting upon our implementation of blockchain in an energy delivery system, we are now able to answer some of the questions that our client initially had, such as, is blockchain a viable option in this situation? Our final product shows that blockchain can handle consensus, and validate that integrity of the information in the blockchain is maintained. By modeling our system after the NASPInet standard, see Figure 10, and aiming for a goal of 60 frames per second being published, NASPInet defines that our latency must be less than 100 milliseconds per frame. The best latency our system could achieve was approximately 300 milliseconds. To compensate for this unsatisfactory latency result, the team decided to concatenate frames together. The team was able to achieve 60 frames concatenated into a single create request, then publish this request once per second, resulting in a hypothetical 60 frames per second. Although these results do not perfectly reflect the NASPInet standard, they do meet the goal of publishing 60 frames per second, with some slight modifications to our expectations. Our blockchain network still improved upon typical energy delivery systems' security by using an authentication system for operations on the network and removed the single points of failure from the system.

5.2. Resources

CouchDB: http://docs.couchdb.org/en/stable/
Express: https://expressjs.com/en/4x/api.html
HyperLedger Fabric: https://hyperledger-fabric.readthedocs.io/en/release-1.4/
HyperLedger Convector: https://docs.covalentx.com/article/71-getting-started
Moesif Orign & CORS Changer :Chrome Store Link
PowerCyber Labs: http://powercybersec.ece.iastate.edu/powercyber/welcome.php
Raft: https://raft.github.io/
React.JS: https://reactjs.org/docs/getting-started.html
NodeJS WebSockets: https://github.com/websockets/ws
NodeJS PCAP Parser: https://www.npmjs.com/package/pcap-parser

6. Appendices

6.1. Operation Manual

Optimally, the project will be deployed through a CI script, there are many complex issues that could occur without a CI script to ensure conformity in deploying the project. There is a CI script in the root directory of the project repository named *.gitlab-ci.yml*.

6.1.1. Blockchain Manual

- Navigate to bc/network and run the generate_script.sh. (Keep the results of this)
- HyperLedger Fabric is doing communication between virtual machines by way of Docker Swarm, here are the steps for starting and running the swarm.
  - Navigate to bc/swarm
  - Node 1
    - Run start_manager.sh
    - Run hacky_script.sh
  - Nodes 2-5
    - Run `sed -i ‘$ s/$/ --advertise-addr <Node IP Address> join_command’` (join_command was created originally by the hacky_script in the previous step)
    - Run `sh join_command`
    - Run hacky_script.sh
    - Note: You may need to adjust the variables in these files to fit your machine.
- Start HyperLedger Fabric
  - Nodes 1-5 (represented by X)
    - Navigate to bc/network/start-scripts/org${X-1}
    - Run node-${X}.sh
  - Note: You will need to change the hostname of each of the node-${X}.sh scripts to the hostname of the machine you are using for this Organization.
- Install Chaincode
  - Navigate to bc/network
  - Execute command `./scripts/start-cli.sh`
  - Navigate to bc
  - Execute command `./startFabric.sh`
  - To Run Tests on the Chaincode navigate to bc/chaincode, then either monitoring/javascript or utility/javascript.
    - npm install (install dependencies)

        - npm run coverage (run tests and check code coverage)
- **NOTE: HyperLedger Fabric Network should not be ran locally as it was developed with remote use in mind.**

6.1.2. API Manual

- Change directory into *api/*
- Install dependencies by running *npm i*
- Running Tests
    - Use *npm test* to run all tests. To run a specific test run *npm test path/to/test.js*.
- Serving Locally
    - Use *npm run dev* to start serving the API locally on port 8080.
- Compiling a Build
    - Use *npm run build* to compile a build. The compiled files will be stored in *api/dist/*.
- On our virtual machines, the API is containerized and served using several commands involving Docker. The commands are located in the gitlab-ci.yml file found in the root directory of the git repository.

6.1.3. UI Manual

- When in the cloned git repository, change into the ‘ui’ directory.
- In a terminal, run ‘npm i’ to install dependencies.
- To run tests, run ‘npm run test’. Test results will be printed in the terminal.
- To run a local server, run ‘npm run start’.
- On our virtual machines, the React app is containerized and served using several commands involving docker. The commands are located in the gitlab-ci.yml file found in the root directory of the git repository.

6.1.4. Publisher Manual

- The publisher will be deployed in a state where it will publish metrics of a specific Organization. To publish metrics to all 3 Organizations concurrently, you will need to do this three times and modify the Org environment argument in docker run command, replace ${ORG} in all commands with 0, 1, or 2.
- Execute `mv bc/network/connection-org${ORG}.json publisher/src/bc/connection-org${ORG}.json`.
- Execute `cd publisher/`.
- Execute `docker build -t publisher .`.
- Execute `docker run --env ORG=${ORG} --net host --name=publisher_container -d publisher`.

- You will need to ensure that the docker container named publisher_container is not running when executing Step 4.
- Run tests against the library with `npm run coverage`.

6.1.5. Publisher-Subscriber Performance Tool Manual

- The PubSub Testing Tool directions are the exact same as the Publisher deployment directions with some minor adjustments.
- The directories to navigate to are cd pubsub-recorder/subscriber and pubsub-recorder/publisher for Subscriber and Publisher respectively.
- Additionally, the subscriber should be deployed with an additional environment variable argument on the docker run command. This should be IPADDR=${The Subscriber IP Address}

6.2. Previous Design Versions

6.2.1. The software we wanted to use initially for developing the smart contracts, HyperLedger Composer, was depreciated.recently causing our team to explore other options, leading to us using Convector instead.

6.2.2. We realized that Convector wasn't providing us with the autonomy necessary for moving forward in our implementation and decided to not use it as well.

6.2.3. Our initial blockchain was able to only process 0.4 executions per second after it's first successful latency test, which was a lot lower than needed.

6.3. Project Code

6.3.1. Our projects code can be found on our team's git page, found here:

- http://sdmay20-12.sd.ece.iastate.edu