Applying and Evaluating Blockchain in Energy Delivery Systems

FINAL REPORT

Team:	Sdmay21-01
Client:	Grant Johnson
Faculty Advisor:	Gelli Ravikumar
Team Members/Roles:	Joshua Edwards - API Developer
	Dylan McCormick - Lab Integration Lead
	Thai Pham - Blockchain Developer
	Owen Snyder - UI/Frontend Developer
	Emileo Xiao - Caliper Evaluation Lead
Team Email:	sddec21-01@iastate.edu
Team Website:	https://sddec21-01.sd.ece.iastate.edu/

Table of Contents

List of Figures:

1. **Introduction**

1.1. Problem Statement

Energy Delivery Systems are deployed in an environment that is geographically distributed through public internet infrastructure. The integrity of measurements, commands, and authenticity of control devices performing communication are critical for trusted operations. Energy Delivery Systems are increasingly encountering security threats in the digital domain, and although current systems work with the developed security features such as encryption, firewalls, and authentication techniques, our client is curious about the quality of a blockchain application in securing critical information and ways to measure its effectiveness.

1.2. Solution Statement

The purpose of this project is to develop a HyperLedger Fabric blockchain network to be distributed within the PowerCyber Labs of Iowa State University. Blockchain systems are innately very secure, with the reasoning behind that discussed shortly, and attacks against them are near impossible until quantum computing becomes mainstream.

Hyperledger Fabric's ledger is immutable, meaning that it is unable to be changed by someone without the proper authority to do so. Our project will piggyback off of prior teams in order to achieve integration with an Opal-RT module within PowerCyber Labs which generates various electrical measurements in real time. This integration will allow for a full pipeline of data from the lab to a distributed blockchain network. Furthermore, we utilized HyperLedger Caliper which is a module developed by the HyperLedger foundation with the aim to provide performance metrics surrounding the deployed network.

This Caliper module is what allows us to determine if this solution is valid, as we are ultimately performing R&D to see if blockchain is a solid replacement for legacy systems in place currently.

1.3. Design Evolution

In 491, our initial plan was to continue development of a prior senior design team's blockchain network, with the end goal being to add on the real-world integration mentioned above retroactively. This, however, proved to be a plan that was scrapped due to the fact that as a team we would not gain much knowledge of the underlying technology of blockchain, which was the main reason most team members chose this project. We then shifted gears to develop our own blockchain network.

Plans were initially to have a 5 node/organization network each with 2 peers to participate in the network. Since then the final implementation goal was scaled back to 2 nodes with 2 peers each. This would provide us with a strong base to determine if this is a viable solution long term. Short term success with this miniscule network would allow our client to look to the future a bit, as it is likely that this network would be the quickest in terms of latency and transactions per second achievable which is imperative with sixty records of data being produced each second.

Another aspect that we were tasked with implementing is adding evaluation tools being HyperLedger Caliper. This is a testing framework that was developed by the HyperLedger foundation with the goal of providing flexible, powerful, digestible testing capabilities to all things fabric. This proved to be the main driving force behind the project as our client was most interested in learning more about Caliper, what its limitations are, and what it does extremely well. Our team had a dedicated member working with Caliper in order to fulfill that desire for our client.

1.4. Functional Requirements

<u>Blockchain Network:</u>

The Blockchain Network will consist of at least five organizations.

The Blockchain Network will have one orderer for the entire network.

<u>Hyperledger Caliper:</u>

The Blockchain will be evaluated using Hyperledger Caliper.

Hyperledger Caliper will test every Smart Contract method.

<u>Smart Contract Layer:</u>

Smart Contracts will be functionally responsible for read, update, delete and query data stored on the ledger.

Smart Contract must update and run quick enough to avoid timeouts leading to data loss.

User Interface:

The User Interface will be developed with commonly used web development tools.

The User Interface will distribute specified metrics and/or measurements to users.

The User Interface will issue authorized user specific operator commands to modify/process the ledger.

API:

The API shall solicit the Smart Contract functions to the requests made by the web application.

All API calls to the Blockchain Node System will be authenticated.

Any calls not meeting specification shall have corresponding errors with response status codes.

1.5. Non-Functional Requirements

Blockchain Network:

The Blockchain Network will be implemented in docker containers allowing for flexibility when running a network.

The Blockchain Network must run in a Linux system environment.

Hyperledger Caliper:

Hyperledger Caliper will execute its tests with its own specified wallet.

Hyperledger Caliper will execute tests on every organization.

Smart Contract Layer:

The Smart Contract Layer will push updates to the Blockchain Network.

The Smart Contract Layer controllers and models will have detailed documentation that describes functionality and intended use.

The Smart Contract Layer will be activated through an application which receives server data.

User Interface:

The User Interface will display the current status of the blockchain network.

API:

The API shall have Unit testing to verify validity of endpoints.

The API shall execute smart contract requests which do not exceed 10 seconds.

The API shall handle connection to the network on behalf of the user and data source.

The API shall provide documentation to the relevant parties via Swagger.IO documentation.

Maintainability:

There will be freely and readily available documentation via the project wiki containing a detailed description of the project should the client have the need to make modifications.

The project will be developed in such a way that making changes would not compromise the integrity of the project.

1.6. Relevant Standards

- 1028-2008 - IEEE Standard for Software Reviews and Audits
  - De Facto standard - Used to measure progress and keep track of work being done, as well as, quality of code. Common to follow similar procedures when working in a group even if it didn't have representation behind this standard.

1.7. Security Concerns and Countermeasures

Security concerns are the main driving factor behind this project, in the wake of the cyber attack on the Colonial Pipeline, it became more evident that Energy Systems are and will increasingly become targets of digital attacks. Whether it be domestic or from abroad, a shutdown of Energy Delivery systems in terms of electricity or traditional fuel can prove catastrophic to communities large and small. Moving into our implementation, as mentioned security is innately developed into blockchain as well as HyperLedger Fabric's version of it. This is not to say that blockchain is an unbreakable lock, as there are absolutely means to cause issues within a blockchain network.

HyperLedger Fabric does not follow other blockchain technologies in the sense of block verification. Fabric uses Proof of Authority, which differs greatly from both Proof of Work and Proof of Stake (used in other applications such as Bitcoin and Cardano respectively). Rather than spending large amounts of computing power which costs large amounts of money over time and leads slow transaction speed (Proof of Work), or staking expensive cryptocurrencies also having slow transaction times (Proof of Stake), Proof of Authority uses the reputation of the block validators as the collateral value to deter malicious actors.

Another concern that is mitigated through the raw use of blockchain in this application is data tampering. This problem is solved by a hashing process in which the header of each block of transactions contains a SHA256 hash of itself, along with a SHA256 hash of the preceding block. This is where the “chain” aspect of blockchain comes to play as each block is cryptographically chained to the data block that was verified prior. In the case of an unauthorized change to data, the compromised node would be unable to post any more transactions as its chain of hashes would differ from the others, resulting in a rollback to an agreed upon chain in order to again reach consensus among all nodes.

One concern that we must discuss is external access, all of the work done implementing a blockchain network will not mean anything if external access is not managed properly. Given that the network is distributed, machines must be able to communicate with each other adding the requirement that some ports will be exposed to external communication. The majority of components are completely internal, thus ports not necessary for essential blockchain functionality should be closed to avoid flooding attacks among others that could take down nodes on the network. The only components that need external exposure would be the peers, certificate authorities, and an orderer to communicate with the client SDK.

Administrative spoofing is a concern of high priority, as a malicious actor which the network views as an administrator could prove devastating to the network. A mitigation tactic possible of preventing such an intrusion would be to implement TLS client authentication on the previously mentioned exposed components of the network. TLS utilizes certificates exchanged ideally by both parties, but only vital to verify the client before allowing a connection, this exchange will disconnect in the case of an invalid certificate or no certificate adding another layer of security between attackers and the network.

Lastly, another concern we’re aware of would be a local host intrusion. There is not much that can be done, as there have to be certain machines that are completely trusted, but things that could make data just that much more secure would be a secondary form of authentication to enter the lab (biometric, key card, ect), a timeout which logs out users after a certain time of inactivity, and to encrypt at the database level with our case being the CouchDB.

2. Implementation Details

2.1. HyperLedger Fabric Blockchain

Our HyperLedger Fabric network is built upon the culmination of several things. First, the docker-compose files that create the containers and volumes for the various organizations and orders within the network. Secondly, the configtx files that configure all the peers and orderers, and the various shell scripts that deploy these nodes, the channels, and the chain-code that runs the smart contracts the nodes will be using.

All of the individual organization configuration files are stored in the “organization” directory under Network-Testing, the docker-compose files are stored under “docker”, and a majority of the scripts used under ./network.sh (shell script to deploy and tear down the network) are stored under the scripts directory.

The way things are set up the network will be running locally and connects to the IBM-HyperLedger VSCode extension that we have that serves as middle-ware to deploy queries and transactions for our network and to connect to our backend API as well as the UI.

2.2. Hyperledger Caliper

Hyperledger Caliper is a blockchain based testing software that works with many blockchain technologies such as: Hyperledger Besu, Hyperledger Burrow, Ethereum, Hyperledger Fabric, FISCO BCOS, Hyperledger Iroha, and Hyperledger Sawtooth. Furthermore, Hyperledger Caliper has many performance metrics. This includes: success rate (how many successful or failed transaction in a test cycle), transaction and read throughput (the flow rate of all transactions through the system per second in a test cycle), transaction and read latency (the time between a transaction being issued and completed in a test cycle), and resource consumption (maximum and minimum memory/CPU usage and I/O traffic in a test cycle).

Hyperledger Caliper has three main files for testing: Workload Module, Benchmark Configuration, and Network Configuration. From the figure below, it can be seen that Benchmark Artifacts is missing from the list of main files.

Benchmark Artifacts represent any additional resources needed for Hyperledger Caliper to monitor the blockchain network. In this project, there were no Benchmark Artifacts needed. Thus, it is not listed as a main file.

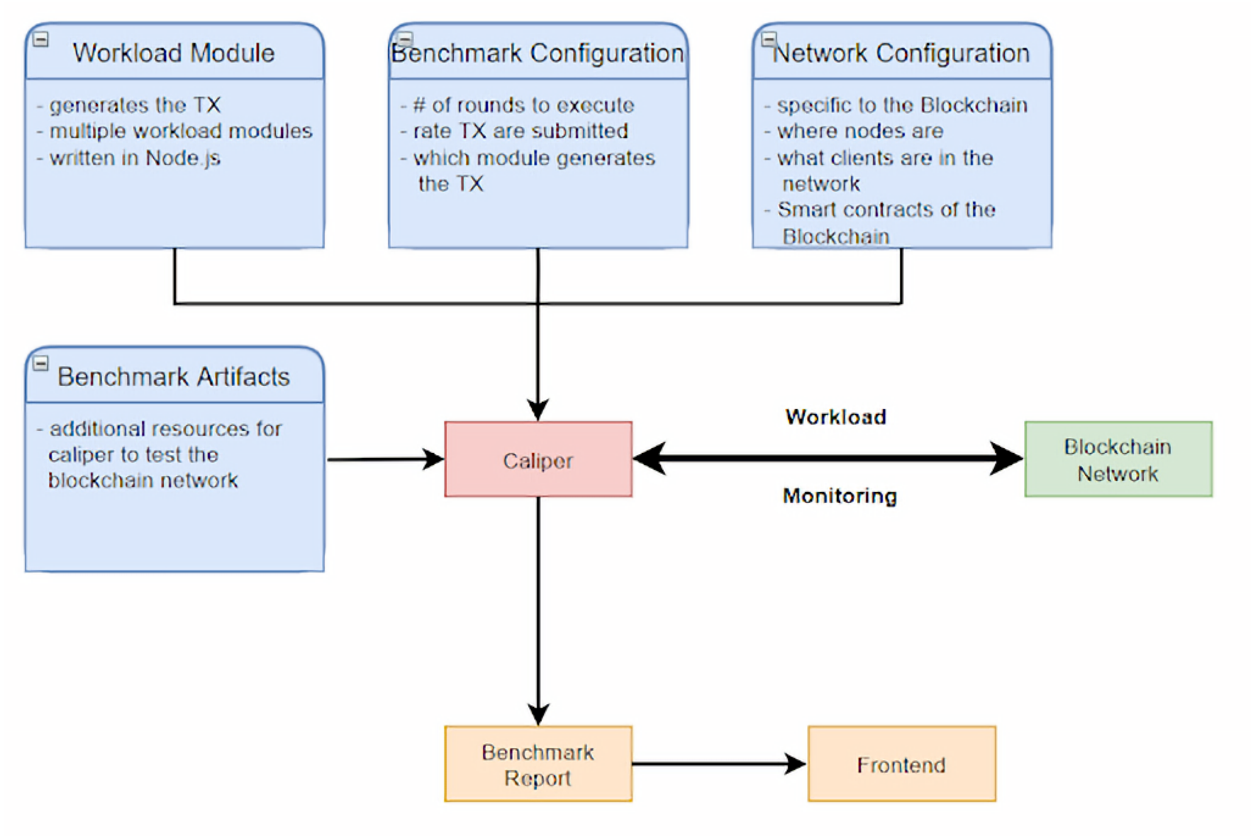


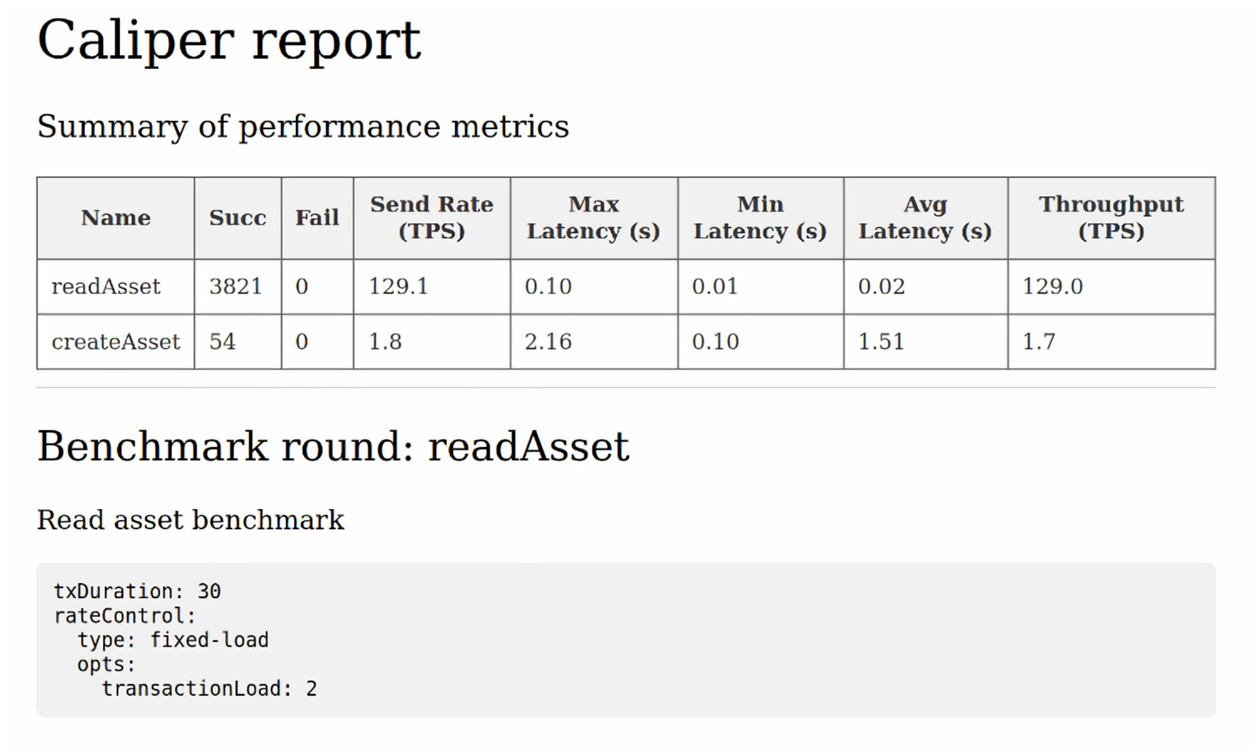
Figure 1: HyperLedger Caliper Overview

Looking at the Workload Module, it contains multiple Node.js files. Each Node.js file corresponds to a method in the Smart Contract. Since the Smart Contract has six methods, there are six Node.js files. Each file can be split up into three methods: initializeWorkloadModule, submitTransaction, and cleanupWorkloadModule. InitializeWorkloadModule is where you can initialize items or objects that will be needed to run the tests. SubmitTransaction is where the code to submit a transaction is written and monitored. CleanupWorkloadModule is invoked after the completion of the benchmark, and it is used to cleanup dummy or test objects used during benchmarking. InitializeWorkloadModule and CleanupWorkloadModule are not required, but submitTrasnaction is required for every test.

Looking at the Benchmark Configuration, it is the file that controls how we run the tests that are written in the Workload Module. It specifies the following: number of test workers to use when generating the load, the number of test rounds, the duration of each round, the rate control applied to the transaction load during each round, and options relating to monitors. All of these options can be customized based on the user’s needs. What this means is that Hyperledger Caliper allows the creation of complex test scenarios based on a complex environment.

Looking at the Network Configuration, it is the file that connects Hyperledger Caliper to the blockchain network that needs to be monitored. In this project, Hyperledger Caliper is connected to the custom Hyperledger Fabric blockchain network. The Network Configuration file specifies the path to the connection profile (exported from the blockchain network), user specific certificate authorities, and user specific private keys. This allows Hyperledger Caliper to have permissions to invoke Smart Contract transactions on the blockchain network.

After Hyperledger Caliper runs all of its tests, it will return the results in the console and an html file.



Caliper report

Summary of performance metrics

Name	Succ	Fail	Send Rate (TPS)	Max Latency (s)	Min Latency (s)	Avg Latency (s)	Throughput (TPS)
readAsset	3821	0	129.1	0.10	0.01	0.02	129.0
createAsset	54	0	1.8	2.16	0.10	1.51	1.7

Benchmark round: readAsset

Read asset benchmark

```
txDuration: 30
rateControl:
  type: fixed-load
  opts:
    transactionLoad: 2
```

Figure 2: HyperLedger Caliper Example Report

Looking at the sample benchmark report above, it displays all the information to the user in a readable format. Looking closely at the table, each test that ran becomes a row in the table. Likewise, each test also has its own specified Benchmark round section which links to the corresponding section in the Benchmark Module. This is helpful for the user to interpret the results or debug if the results are not satisfactory.

2.3. Chaincode

Our smart contract was implemented in javascript and has some differences compared to the previous team's implementation. One of these differences is in how the contracts are called. The previous team went with an implementation where they would just call the contracts without using an api to connect to their server with http requests. We initially started along that route as well, but as a team we came to the decision that having everything using the same framework for requests would make more sense for unifying the different moving pieces we had to connect and that meant more calls to our backend/api instead of using the frontloaded hyperledger fabric calls available.

Another difference is in the available commands. We added the ability to update records, delete specific records, and to delete all records. Great for real world applications involving testing or instances where data has been mishandled and needs to be righted. Currently we have an application that just deals with connecting to a websocket and creating records based on the input it receives, since that was the main idea behind what this technology would be used for. The ability to expand the application would not be too difficult and there are other ways to execute those other commands for now.

2.4. User Interface

The user interface was developed as a ReactJS web application using the Material UI component library to cut down on development time. The UI was designed to consist of a main dashboard page with 2 tabs that correspond to different information that a user would like to be displayed. The first tab contains phasor metrics from the smart contract ledger and includes functionality to view these metrics as well as the performance of the system in the form of a line graph. The second tab includes information from the caliper reports that have been run on the system. The UI interacts with the API using Axios, a promise-based library for making HTTP requests. The API sends data from every phasor from each organization in the blockchain which is then parsed for relevant information which is then displayed to the user.

2.5. Backend/API

As for the API, not much changed along the way. It was expected that it be done in JavaScript to be able to utilize the robust yet lightweight framework that the *Express* library has to offer for backend developers. The majority of the time was spent by the backend developer gaining a solid understanding of the shape that an API will take, given that he had not built a RESTful API prior to this project.

First, let's run through the file structure the API takes on in an aid to explain the modularization that went into development. The "bin" folder contains the sole executable source code that represents the backend, it sets up an Express app using a dependency to the app.js file in the root directory which outlines the routes and error handling that is implemented.

The next folder to mention is the "fabric" folder which is used to store identification information for each organization, being the organization itself, and the representative

Certificate Authority. These identification jsons allow for verification of transactions before execution.

Lastly, there is the "routes" folder. This is the meat and potatoes of what is an express application, the "routes" describe each group of endpoints that are available to the Opal-RT module within the lab and the client. The network file, as the name suggests, handles all interactions with the network, with those being to create, update, delete, and query an asset. Initially, the organization specific json file is parsed to get the information necessary to interact with the ledger on behalf of that organization. This is used to have the organization user name, gateway, wallet, channel, and contract name all specified, which is later used within each of the network actions outlined above to create a connection profile, a gateway (a communication link between the blockchain and the backend) using said connection profile, and finally a connection to the specified channel on the network with the created gateway. Once this is done, the asset can be created, updated, queried, or deleted as seen fit.

Finally, within the routes package, there is the index.js file which outlines the URLs usable to interact with the blockchain. It describes HTML requests with asset specifications passed as header values allowing the utilization of the network functions that are implemented in the network.js file.

Finally, it should be mentioned that Swagger API documentation has been generated to allow users to see the endpoints and how they are constructed. Swagger is a very high quality documentation source that also allows for interaction as well.

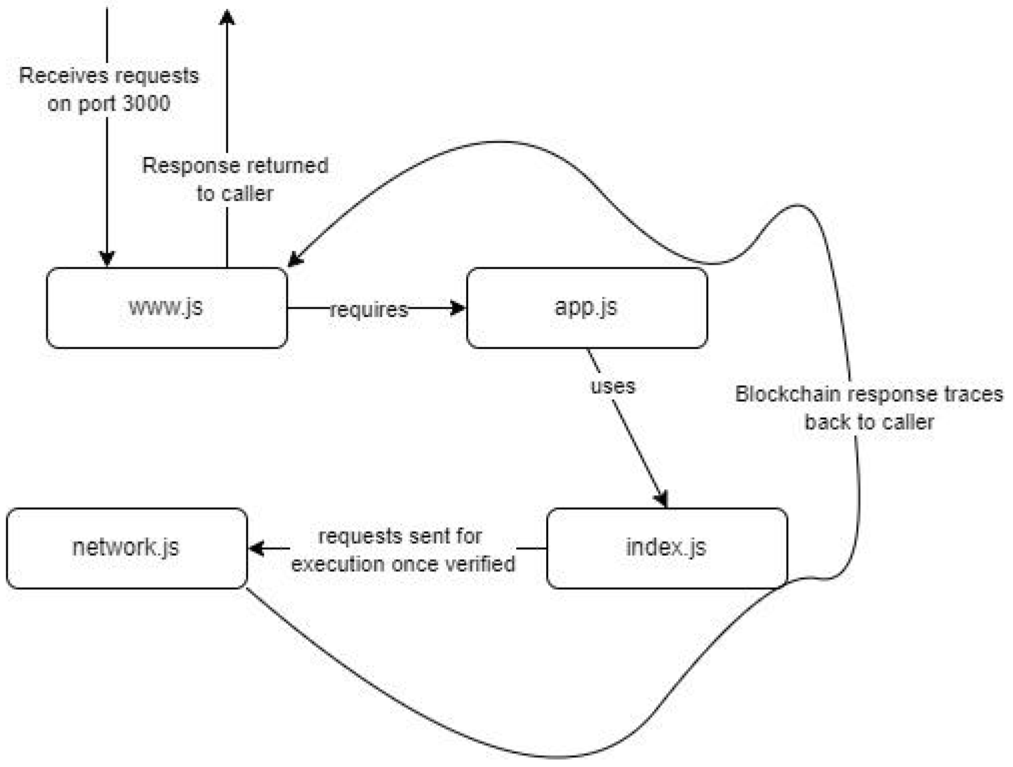
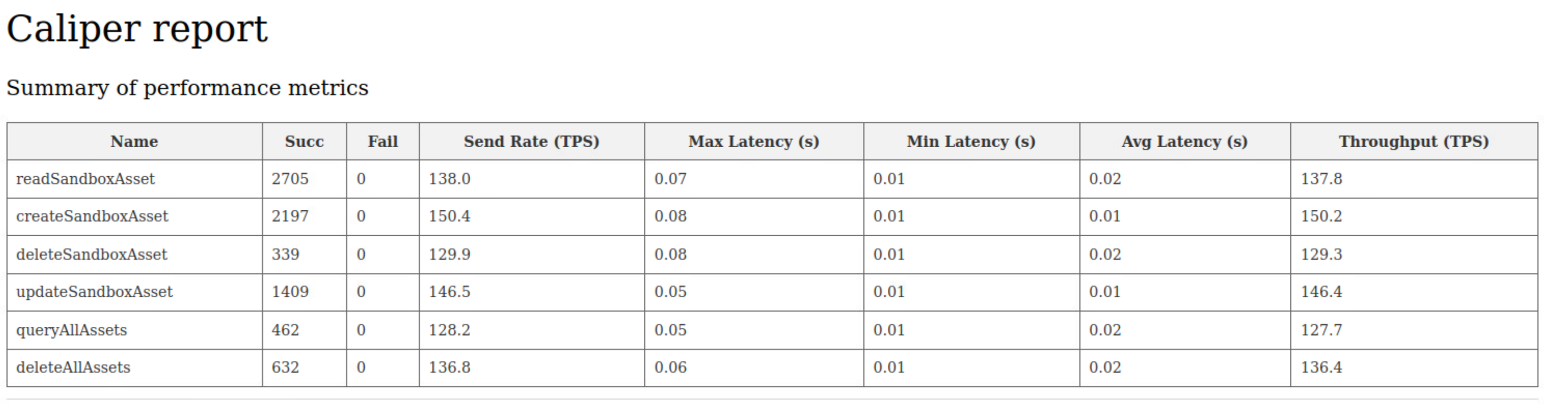

Receives requests on port 3000
Response returned to caller
www.js
requires
app.js
uses
Blockchain response traces back to caller
network.js
requests sent for execution once verified
index.js

Figure 3: API Data Flow Diagram

3. Testing Process and Testing Results

3.1. HyperLedger Caliper

In this project, Hyperledger Caliper will be used to monitor the Smart Contract transactions on the custom blockchain network. The Smart Contract includes: createSandboxAsset, readSandboxAsset, updateSandboxAsset, deleteSandboxAsset, queryAllAssets, and deleteAllAssets. After running these tests, Hyperledger Caliper will return a html file for our client to see and interpret for themselves.



Caliper report

Summary of performance metrics

Name	Succ	Fail	Send Rate (TPS)	Max Latency (s)	Min Latency (s)	Avg Latency (s)	Throughput (TPS)
readSandboxAsset	2705	0	138.0	0.07	0.01	0.02	137.8
createSandboxAsset	2197	0	150.4	0.08	0.01	0.01	150.2
deleteSandboxAsset	339	0	129.9	0.08	0.01	0.02	129.3
updateSandboxAsset	1409	0	146.5	0.05	0.01	0.01	146.4
queryAllAssets	462	0	128.2	0.05	0.01	0.02	127.7
deleteAllAssets	632	0	136.8	0.06	0.01	0.02	136.4

Figure 4: Final Caliper Report

The caliper report shown above is after running one instance of all the tests. While it cannot be interpreted by the group members, it is an important piece of data for our client. The team at PowerCyberLabs can use this data to help debug or refactor the blockchain to meet their standards. Without the caliper report, there would be no possible way for them to test the latency or throughput of the blockchain network. Based on these results, our client can then plan for their next possible steps of going along with the blockchain or looking for another solution.

3.2. Chaincode (in reference to our ‘APP’, majority of chaincode testing occurs in ‘3.5 Backend/API’)

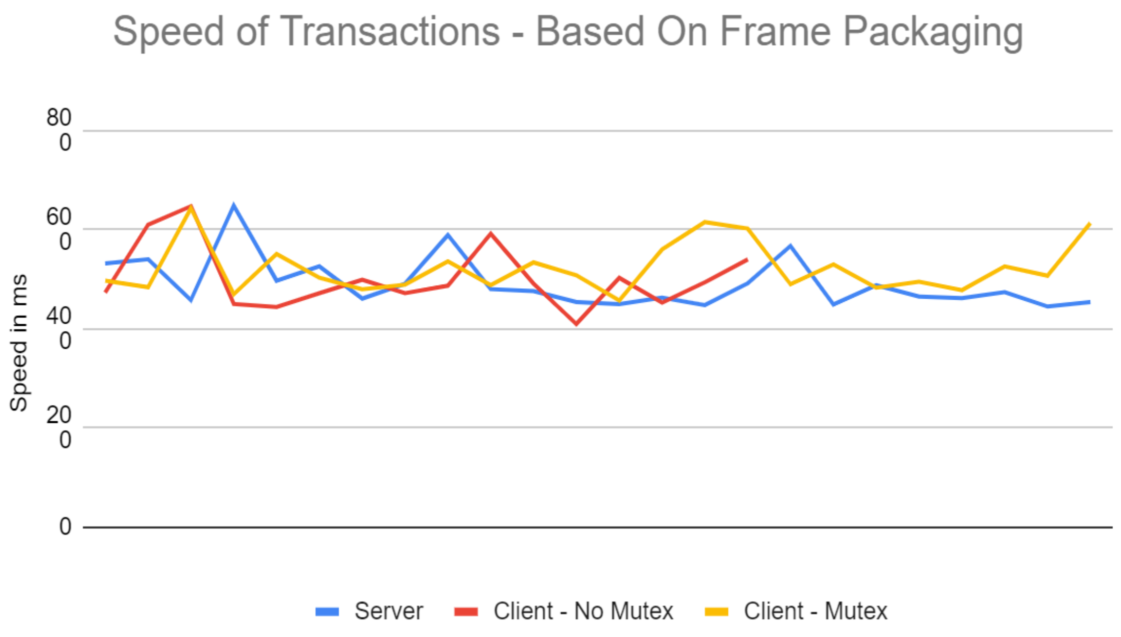


Figure 5: Chaincode Transaction Speed Comparison

This is a graph showcasing what happened when we sent data from our imitation data server to our application in different formats. The process involved putting in the speeds measured from our transactions into a spreadsheet and comparing time. ‘Server’ was when our server would package 60 frames together before sending it to our application. It sped up transaction times by an average of 20 ms. ‘Client-No Mutex’ was when our client would receive a single frame from the server at a time and had to package them together. It had data loss issues which will be presented in our next chart and seemed to have the most inconsistent data values. ‘Client-Mutex’ is our stable implementation which best fits what we and our client expected from our application. It was the slowest but was stable and fits our needs perfectly.

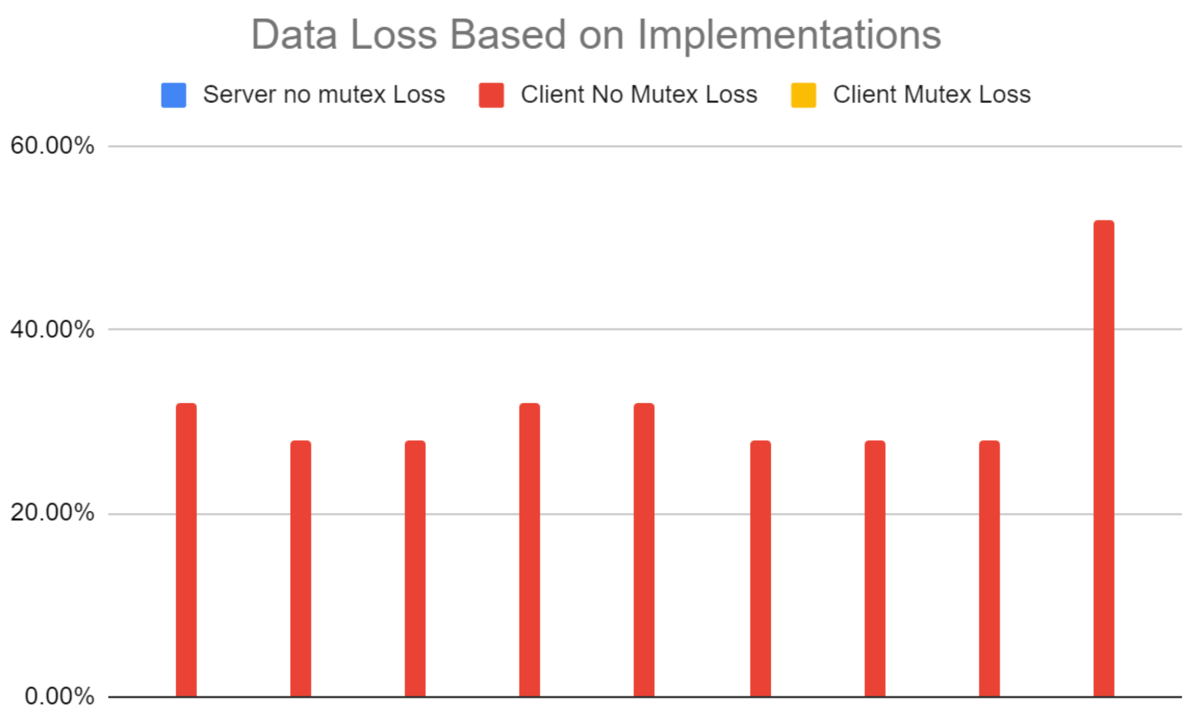


Figure 6: Chaincode Data Loss Comparison

Same format as above for the legend and how data was sent. As shown here, when the server sends out the data or the client receives and packages it with a mutex, there is no data loss. When the client has to package the data received from the server without a mutex, there is massive data loss. This is because of the async nature of the websocket calls messing up when the counter would trigger the array to push through our smart contract API. The process involved emptying our database and repeatedly letting 1500 entries try to fill up the database under different implementations.

3.3. User Interface

The UI components that interact with the API to retrieve data from the blockchain network were Unit tested using the Jest framework along with the React Testing Library. This allows us to see that all of the components are rendering correctly and receiving data in the format that is expected by the UI so that it can be displayed properly.

Each component that was tested has a corresponding test file that runs unit tests on the specific functions of that component.

Jest works very well in tandem with ReactJS, both of which are developed by Facebook/Meta. It provides the framework for creating, formatting, organizing, and running tests on Javascript elements.

3.4. Backend/API

Thoroughly tested API endpoints are the core to a solid working application, allowing all components interacting with said API to have confidence that any issue encountered would be their own. This aids in development and helps to shape an application moving forward as change is inevitable in software systems. Testing provides a known benchmark as to what "working" code is, and informs developers about small bugs that normally would not be caught by simply making a single change and testing the change in a production sense. Overall, a well tested API is used to gain confidence in performance. Depending on the framework used, nearly any facet of an API can be tested within defined metrics, and these metrics show with certainty that the software is acting as intended.

Moving into the specific testing utilized in the API for our application, the backend developer decided on using a combination of 'Chai' and 'Mocha'. Mocha is the framework, and as the name suggests it provides a 'framework' for the tests to run. This includes outlining guidelines and rules used for creating and designing test cases within a specific application. It also automates the testing process, and in our case, allows for a simple "npm test" command to apply all tests to the currently deployed code without any further intervention from the developer. Next, we have Chai. This is a javascript TDD assertion library for Node.js that pairs with any javascript testing framework, AKA mocha. This library has extensive documentation, is highly compatible, and has countless assertion operations making it highly flexible in the testing that can be carried out with it. Finally, under test, feedback within the console is digestible, with Mocha providing exception mapping to show which test case failed under which conditions along with what exception was thrown.

Chai also supports three different interfaces of their testing options: 'should','expect', and 'assert'. The backend developer decided on the should interface as that is the most human readable, while still offering the wide range of flexibility offered to thoroughly test the endpoints. In terms of the specific tests, cases were obviously written for each of the endpoints. These cases, in turn, contained multiple different manners of testing the same endpoints each with their own respective 'should' statements. Each of the statements, asserted what the value truly is, and what it should be depending upon the expected values both received from the caller, and received from the blockchain network. These cases included the clear need to have a properly formatted request resulting in a proper response, where values within the response are asserted to be as they should be. Also included would be an improperly formatted request, where in the case of queries, updates, or deletes did not have the necessary parameters to execute the request on the blockchain.

4. Context

4.1. Related Products/Literature

One solid application that is actually quite similar in topic to what we did is the Siemens/LO3 energy "Brooklyn Microgrid Project". Microgrid refers to the relatively new approach of reversing the traditional unilateral direction of power and introducing more of an interconnected network type of framework. This is done with their microgrid modules that offer a wide range of benefits including algorithmic energy usage optimization, 24/7 load balancing and blackout protection, and supporting sustainable energy generation techniques.

This is an incredibly useful module in and of itself, but Siemens went above and beyond to prove the networkability of their technology by partnering with LO3 Energy's blockchain platform allowing for transactions of electricity to be executed exchanging energy in a decentralized manner with full transparency and security, knowledge of which can be view in the literature cited in our works cited section. This small example in Brooklyn, however is not the end goal, as the technology could prove particularly promising in more remote regions where the power grids are not as robust and blackouts are more common. A small group of sustainable energy generation 'nodes' could, in these settings, act as utility companies supplying reliable electricity to those who need it at a fraction of the cost. Once a network is scaled, then the potential can really be observed. Battery stores are distributed, many nodes are generating and contributing electricity, users are consuming the electricity paying in a peer-to-peer manner lowering cost substantially, and everyone wins, including the Earth.

5. Appendices

5.1. Appendix I - "Operation Manual"

Our project has multiple moving parts that need to be working in order for it to operate correctly. This manual will be split into sections which are ordered as one should follow to set up the system. It will also include errors that we ran into or thought about and possible solutions.

Stage 1: General (If errors at any step - check the bottom of this section for troubleshooting issues we ran into)

Go to our website to find a link to our git repo. Clone our project onto your machine. Make sure VScode is installed and use its extension toolbar to download Pylance(v2021.12.0) and IBM Blockchain Platform(v2.0.5). Then in the top level folder make sure to run npm install to grab dependencies.

There are now a few steps to go through to make sure everything will be set up correctly to run. One is to make sure the VScode interpreter has Python 3.8.10 64-bit running on it.   VScode has a status bar that runs along the bottom of its window, which has a button to allow this swap to the correct python path (/usr/bin/python3   In our case).

Stage 2: Server

Now we have a couple of things to configure. Go into the folder serverTests and open the fakeServer.py file. Scroll down to line 31 and if you know the port is available then make no changes, keep in mind that if you run into a port issue later - you will want to make sure the listed port is available for our application.

Stage 3: fabric-sandbox-app

Inside the fabric-sandbox-app folder and inside of the src directory is our application named create.js. Open this up and on line 6 you can change the orgID associated with the key of all the data going through this application. On line 20 you can change the websocket server port that is being created. If necessary - keep in mind that you will have to go back and change the port number inside of fakeServer.py if you do this so that they match.

(Note: If integrated with OpalRT - that websocket connection info must be changed and the create.js websocket connections should be more similar to the fakeServer.py connection, since the APP would no long CREATE the server and just be connecting to OpalRT)

Stage 4: fabric-sandbox-api

Fabric-sandbox-api has a folder named 'fabric'. You will need to export a connection.json file and a folder with the wallet info into this folder for our API to work. To do this go to the IBM extension interface (this is done by clicking the square on the left side of the VScode interface). Once there, inside of the Fabric Environments interface click on '1 Org Local Fabric' and then in the Fabric Gateways interface click on 'Org1 Gateway'. Click the ellipses next to Fabric Gateway, and export the connection file to the fabric folder as mentioned before (connection.json is the name it needs). Then inside of the Fabric Wallets interface, right click Org1 (NOT Org1 Admin) and export the wallet to the fabric folder as well (Org1 is the name it needs).

Stage 5: fabric-sandbox-sc

In visual studio code, go to file - new window. Open up the FOLDER for fabric-sandbox-sc, it must be the top level folder opened up in order to continue. Once that is done, go into the IBM extension and connect to the fabric environment/gateway like in the previous stage (just clicking to connect - no need for dealing with exporting). Click on the ellipses next to the smart contracts interface and "package open project" (keep note of the version number). When it finishes, inside of the fabric environments interface, click the drop drop for 'mychannel' and hit deploy smart contract. A new tab will pop open and you need to select the correct version you just packaged and hit the blue deploy button multiple times, no need to worry about settings. The smart contract should now be deployed.

Stage 6: Running
(Assumes stage above have happened. If not, then make sure to start the fabric environment and gateway through the IBM extension)

In order:
Open a terminal and go into the fabric-sandbox-api directory. Type npm start.
Open another terminal and go into the fabric-sandbox-app directory. Type npm start.
Go into serverTests directory and in the top right corner of VScode should be a 'play' arrow to start the python file.

The project should begin working together and sending data into the ledger.

(If the ledger is already full, the transactions will repeatedly return error messages that alert you that the Key already exists for each in the ledger - stage 7 will explain an option to reset the ledger)

Stage 7:Testing

In the IBM extension inside of 'fabric gateways' use the dropdown for channels, mychannels, and the smart contracts. Clicking the sandbox button will show different smart contract commands and clicking any of them will open up a window where it's possible to run those commands. This is a great way to run the 'deleteall' transaction if you want to repopulate the ledger with the same values again to test. You simply pick a transaction name, enter the arguments if required, and choose whether to evaluate or submit. Evaluate will not have an effect on the ledger, submit will lead to changes.

Also in the fabric-sandbox-api directory you can run npm test to see the results of a bunch of tests we created for the API/SC backend. The file itself is in fabric-sandbox-api/test/routesTest.js.

When testing it is important to use control-c on the terminals mentioned in stage 6 and then restart them in the correct order (probably after using the deleteall transaction).

Clicking on the ellipses in the IBM extension under fabric environment interface will let you open couchDB (admin , adminpw) is the default local info and will allow you to see all the records if you click on ‘org1_mychannel-fabric-sandbox’.

Stage 8: Errors

1. websockets not working - python interpreter needs to be changed
2. Can’t access docker
   a. sudo usermod -aG docker $USER
   b. sudo chmod 777 /var/run/docker.sock
3. Failed to start 1 Org Local Fabric: Error: Environment failed to become available -
   a. Must edit /etc/systemd/resolved.conf
      i. [Resolve]
      ii. DNS=8.8.8.8
      iii. Domains=~nip.io.
   b. sudo service systemd-resolved restart
4. Timeout on connecting to orderer/user issue -
   a. Export connection profile again to fabric-sandbox-api-/fabric folder
   b. Export Wallet again to fabric-sandbox-api-/fabric folder
5. If port is stuck in use - close port or killall npm to free it

Stage 9: Alternate Network

- First navigate to the Network-Testing directory in the repository and use the command “./network.sh up” to create the containers and nodes for all the organizations.
- Second use the command ./network.sh createChannel to deploy the channel which is defaulted to “mychannel”
  - Alternatively can use ./network.sh createChannel -c “channelname”, otherwise the default channel will be “mychannel” as stated above
    - “Channelname” must be all lowercase and can include numbers
- Lastly, to deploy chaincode onto the network use the command ./network.sh deployCC -ccn “chaincodename” -ccp “chaincodepath” -c “channelname” -ccl “chaincode language”
  - Chaincodename can be anything you would like it to be, it will simply serve as the name of the .tar.gz file that installs onto the organization peers.
  - Chaincodepath for our purposes will be ../fabric-sandbox-sc/
  - -c “channelname” is only necessary if there is a channel that is not the default “mychannel”, otherwise the chaincode will default to installing on mychannel

Stage 10: HyperLedger Caliper

Open a terminal in “caliper-workspace”. Run “npm install -g --only=prod @hyperledger/caliper-cli@0.4.2”. Next, run “caliper bind --caliper-bind-sut fabric:1.4.8 --caliper-bind-args=-g”. The initial setup is finished. To start the tests, run “caliper launch master --caliper-benchconfig benchmarks/myAssetBenchmark.yaml --caliper-networkconfig networks/network_config.json --caliper-workspace ./ --caliper-flow-only-test

--caliper-fabric-gateway-usegateway --caliper-fabric-gateway-discovery". After running the tests, a html file called "report.html" in "caliper-workspace" will be created. This report contains all the results and can be viewed by the user.

Stage 11: UI

In the ui directory, you can follow the path: ui/src/components/. There you can find 2 files called FabricDash.js and CaliperDash.js, both of which must be manipulated in order to communicate with the API properly. In both files, a baseURL variable is declared on line 9. This must be edited to correspond with the IP address of whichever machine the API is currently running on. In the case of the PowerCyber Labs VMs, this IP address will be 10.1.200.51. Once that has been done, the application can be started by opening a console, navigating into the /ui directory and running the "npm start" command. Alternatively, npm start can be run via opening the directory in VSCode and running the start script.

5.2. Appendix II - Alternate Versions of the Design

Version 1.0: This version of the project is essentially what was initially planned when we had a six-man team last semester. Our hope was to get our own blockchain network completely off the ground and have a UI, smart contracts, API, and blockchain testing framework all put together to gather data from PowerCyber labs.

Version 2.0: This is the version our project narrowed down when it became apparent that getting the network up was harder than we thought initially and we were making sure at the very least that our individual modules worked. It involved working heavily with the IBM blockchain extension to act as a network for us while still trying to get our own network functioning.

Version 3.0: This is what we would like to call our *almost* final version and the one we are presenting in this report. After many learning processes, we currently have a system with multiple modules in communication together (still using the IBM extension for the network). Unfortunately as we got down to the wire and wanted to connect with the physical system, PowerCyber labs had technical difficulties that prevented us from doing so. Which is another lesson learned, plan for the worst case scenario.

Version 3.5: This is the version that we will be working on between this report due date and our actual presentations. It is in constant flux and we are hopeful that PowerCyber labs will resolve their issues so we can connect for real data, as well as that we hope to make progress in deploying our own network in the few days we have left.

5.3. Appendix III - HyperLedger Fabric Terminology

Certificate Authority: Component whose job is to issue X.509 certificates to administrators and nodes within the network. These certificates serve the purpose of identifying components in terms of what organization they belong to. This identification is necessary to ensure that organizations are not modifying ledgers belonging to channels of which they don't, among other operations requiring verification of identity.

Channel: Private communication mechanism populated by the relevant organizations involved and defined in its respective consortium (Configuration file outlining details of the channel)

Organization: A term for the part of the system that can assign identities for participants to allow for clear transactions and which has access to channels.

Peer: Allows you to interact with the network and helps endorse and verify transactions.

Ledger: This is the technical term for the database in a blockchain application. They have many properties determining functionality depending on the overall application, but the main difference between a "ledger" and a traditional database is that they are unchangeable without the execution of a smart contract from an administrative figure.

Orderer: Serves as an administrative point for the network, where the initiator and any specified administrators can make changes to the system. Also serves, as the name suggests, as an ordering service with the role of determining which order transactions are placed into blocks for distribution.

Transaction: An invoke result or an instantiate result that is submitted for ordering, validation, and committing.

6. Works Cited

Breuer, Hubertus. "Smart Grids and Energy Storage: A Microgrid Grows in Brooklyn." *Siemens.com Global Website*, Siemens, 16 Feb. 2018, https://new.siemens.com/global/en/company/stories/research-technologies/energytransition/a-microgrid-grows-in-brooklyn.html.

Ganion, Jana. "Amid Rolling Blackouts in California, a Microgrid Proves Its Resilience: Siemens Stories about Smart Infrastructure." *Siemens USA*, Siemens, https://new.siemens.com/us/en/company/press/siemens-stories/smart-infrastructure/amid-rolling-blackouts-in-california-a-microgrid-proves-its-resilience.html.