UNIVERSIDADE FEDERAL DE SANTA CATARINA
CENTRO TECNOLÓGICO
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

Johann Westphall

Blockchain privacy and scalability in a decentralized validated energy trading context with Hyperledger Fabric

Florianópolis
2021

Johann Westphall

Blockchain privacy and scalability in a decentralized validated energy trading context with Hyperledger Fabric

Dissertação submetida ao Programa de Pós-Graduação em Ciência da Computação da Universidade Federal de Santa Catarina para a obtenção do título de mestre em Ciências da Computação.
Supervisor: Prof. Jean Everson Martina, Dr.

Florianópolis
2021

Ficha de identificação da obra elaborada pelo autor,
através do Programa de Geração Automática da Biblioteca Universitária da UFSC.

Westphall, Johann
Blockchain privacy and scalability in a decentralized validated energy trading context with Hyperledger Fabric / Johann Westphall ; orientador, Jean Everson Martina, 2021.
142 p.

Dissertação (mestrado) - Universidade Federal de Santa Catarina, Centro Tecnológico, Programa de Pós-Graduação em Ciência da Computação, Florianópolis, 2021.

Inclui referências.

1. Ciência da Computação. 2. Energia. 3. Blockchain. 4. Performance. 5. Escalabilidade. I. Martina, Jean Everson. II. Universidade Federal de Santa Catarina. Programa de Pós Graduação em Ciência da Computação. III. Título.

Johann Westphall

Blockchain privacy and scalability in a decentralized validated energy trading context with Hyperledger Fabric

O presente trabalho em nível de mestrado foi avaliado e aprovado por banca examinadora composta pelos seguintes membros:

Prof. Antônio Augusto Medeiros Fröhlich, Dr.
Universidade Federal De Santa Catarina

Prof. Luis Carlos Erpen De Bona, Dr.
Universidade Federal do Paraná

Prof. Wilson De Souza Melo Junior, Dr.
Instituto Nacional de Metrologia, Qualidade e Tecnologia

Certificamos que esta é a **versão original e final** do trabalho de conclusão que foi julgado adequado para obtenção do título de mestre em Ciências da Computação.

ICP-Edu
Documento assinado digitalmente
Vania Bogorny
Data: 16/10/2021 06:44:03-0300
CPF: 684.586.570-15
Verifique as assinaturas em https://v.ufsc.br

Vania Bogorny
Coordenação do Programa de
Pós-Graduação

ICP-Brasil
Documento assinado digitalmente
JEAN EVERSON MARTINA
Data: 15/10/2021 17:24:48-0300
CPF: 003.355.939-25
Verifique as assinaturas em https://v.ufsc.br

Prof. Jean Everson Martina, Dr.
Supervisor:

Florianópolis, 2021.

This work is dedicated to my parents, Carlos and Carla,
and anyone who positively impacted my life.

ACKNOWLEDGEMENTS

I thank my father, Carlos, and my mother, Carla, for the constant support in my life and during my Master's period. I also thank Professor Jean Martina for advising this work and bringing relevant discussions to improve this work. Every member of LabSEC's blockchain research group also contributed in some way to our accomplishments, and I appreciate their help. I thank the contributions of Professor Prof. Antônio Augusto Medeiros Fröhlich, Prof. Luis Carlos Erpen De Bona, and Prof. Wilson De Souza Melo Junior while integrating my dissertation committee. Lastly, I thank CAPES for financially supporting this research.

RESUMO

O uso de energia renovável vem aumentando devido à preocupação com preservação do ambiente ameaçada por fontes energéticas como carvão e petróleo. Apersar do alto custo de fontes renováveis em relação às fontes sujas, essa diferença tem diminuido. Com preços mais baixos, pessoas instalam painéis solares para reduzir o custo da conta de eletricidade ou até vender o excesso produzido à empresa transmissora. Quando pessoas vendem energia à rede elétrica, elas são classificadas como *prosumers*. Geralmente, *prosumers* podem vender a energia gerada exclusivamente à companhia de eletricidade, que determina o preço de compra. Mercados de energia decentralizados podem aumentar tanto a competitividade quanto a adoção de fontes energéticas limpas. Ao mesmo tempo, mercados centralizados apresentam vulnerabildades de segurança e carecem de resiliência. Neste contexto, *blockchain* é estudada como uma tecnologia para possiblitar a decentralização de mercados de energia, principalmente por ser um banco de dados resiliente, imutável, transparente e seguro. A literatura apresenta diversas soluções envolvendo *blockchain* e mercados energéticos, todavia mais pesquisa é fundamental para tal implantação. Escalabilidade, privacidade, arquitetura de mercado e segurança do usuário são alguns dos problemas ainda não resolvidos neste tipo de aplicação. *Hyperledger Fabric* predomina na literatura acerca de mercados de energia e também é usado na implementação do modelo deste trabalho. Este trabalho revisa a literatura a respeito de *blockchain* em mercados de energia, propõe um modelo, implementa-o, realiza experimentos e analisa a escalabilidade da rede junto com a proporção da sua geração de dados. O modelo permite que energia limpa e validada seja comercializada por compradores anônimos, evitando a exposição dos seus padrões de consumo. O contrato inteligente do *Hyperledger Fabric* recebe dados proveninentes de sensores e julga se as alegações de geração energética são válidas. Por exemplo, sensores capturadores da velocidade do vento podem evitar que vendedores de energia eólica comercializem quantidades de energia acima da sua capacidade de geração. Depois de validada, a energia é comercializada entre participantes da rede. Modificações no *Hyperledger Fabric* foram necessárias para implementar o modelo definido. O desenvolvimento da proposta é dividido em três partes: desenvolvimento da rede, desenvolvimento do contrato e desenvolvimento da aplicação. Este trabalho adaptou a implementação do *Fabric* para realizar os experimentos planejados. Os experimentos foram executados em três fases com configurações distintas para testar a capacidade da rede. A capacidade máxima de transferência foi atingida em uma configuração com 5000 sensores, 5000 compradores e 5000 vendedores. Tanto o ritmo de geração de dados quanto o custo de implantação foram analisados para julgar a viabildade da rede. Este trabalho complementa com resultados empíricos a literatura, a qual carece destes resultados. Além disso, a estrutura do experimento serve como base para pesquisas futuras com *Hyperledger Fabric*. Ademais, a participação de pesquisadores com formação em engenharia de energia é necessária para o aprimoramento do processo de validação de energia. Este trabalho explorou um conjunto limitado de configurações e trabalhos futuros podem realizar diversos aperfeiçoamentos neste modelo.

Palavras-chave: Energia. Blockchain. Performance. Escalabilidade. Anonimidade. Hyperledger.

RESUMO EXPANDIDO

Introdução

Blockchain é uma tecnologia que suporta um banco de dados decentralizado no contexto Peer-to-Peer (P2P) e é amplamente conhecida por causa da cripto moeda Bitcoin (RAHOUTI *et al.*, 2018), com uma estrutura segura contra adulterações. Blockchain permite a ocorrência de transações seguras entre nodos, sem a necessidade de uma terceira parte confiável. Também é reconhecido como uma tecnologia com potencial parar aprimorar o papel de consumidores em um mercado de energia, aumentando segurança e reduzindo custos (WANG, N. *et al.*, 2019).

Na maioria dos sistemas de distribuição energética, residências com fontes de energia renováveis podem vendê-la apenas para a companhia de transmissão, o que impede ampla negociação de preço entre múltiplos compradores. Pesquisadores têm avaliado blockchain como um facilitador para a implantação de um mercado energético decentralizado, onde residências poderiam negociar energia entre elas.

A adoção de energia renovável, como solar e eólica, é considerada uma importante medida para reduzir a emissão de gases relacionados ao efeito estufa. No final de 2018, o estado da Califórnia tornou obrigatória a instalação de painéis solares em novas construções. A queda abrupta dos custos de instalação também impulsiona a adoção de fontes renováveis (ORSINI *et al.*, 2019). Mesmo que a lei da Califórnia não tenha sido totalmente aplicada, ela representa um movimento em direção à energia limpa e decentralizada.

Medidores inteligentes são uma outra tecnologia necessária em um mercado energético decentralizado. Eles medem a quantidade de energia comprada e vendida na rede de energia. Medidores inteligentes, aliados a blockchain, permitem que companhias de energia e *prosumers* - residências que eventualmente vendem energia à rede - participem em um mercado de energia decentralizado em tempo real, sem pagar nenhuma taxa (JOGUNOLA *et al.*, 2019).

(ALAM, Asraful *et al.*, 2019) defende que um mercado de energia em um microgrid poderia aumentar a receita dos *prosumers*, reduzir o custo de energia e impor eficiência energética. Um microgrid é uma fonte alternativa quando os grandes sistemas de transmissão sofrem algum apagão. Condições de tempo severas que causaram apagões motivaram a criação do Brooklyn Microgrid em Nova Iorque (MENGELKAMP *et al.*, 2018). Lá, o microgrid pode operar em modo “ilha” independente das grandes linhas de transmissão, obtendo maior resiliêcia.

Além de melhorar o sistema energético, o crescimento de fontes energéticas renováveis trazem benefícios ao meio ambiente. Considerando essas vantangens e que blockchain é visto como uma possível ferramenta para viabilzar um mercado de energia decentralizado, pesquisa neste tema pode conduzir a conclusões mais firmes a respeito da aplicação de blockchain a mercados de energia.

(JOHANNING; BRUCKNER, 2019) avaliou projetos envolvendo blockchain e energia. Apesar de reconhecer o potencial de se usar blockchain para comercializar energia, os autores criticaram a documentação rasa dos projetos existentes. Também argumentaram que trabalhos futuros devem trazer contribuições com mais profundidade científica para convencer de que o risco de aderir a blockchain compensa.

(WANG, N. *et al.*, 2019) também analisou soluções com blockchain nos mercados de energia. O artigo deles indica a demanda por aperfeiçoamentos no incentivo do sistema, consenso, regulação, testes com camada física e o impacto da escalabilidade. (BLOM, 2018) apontou que medidores inteligentes talvez não possuam capacidade de processamento para participar na blockchian, dependendo da complexidade do mercado. Os dados provenientes de sensores levantam um outro problema lidado na literatura: como um ambiente blockchain suportaria sensores Internet of Things (IoT) enviando grandes quantidades de dados.

Um exemplo de trabalho que lida com o problema mencionado dos sensores é o de (LE-DANG; LE-NGOC, 2019), que utiliza REST Application Programming Interfaces (APIs) em dispositivos IoT para previnir uma alta utilização de recursos na blockchain. Considerando estes problemas apresentados, é notavel que pesquisa na aplicação de blockchain com IoT para viabilizar um mercado de energia decentralizado pode contribuir significativamente na área de segurança computacional, visto a relevância do tópico.

Objetivos

O objetivo principal deste trabalho é propor e analisar um mercado de energia P2P que usa blockchain, considerando os modelos de mercados de energia já presentes na literatura. Os objetivos específicos são:

- Propor um novo modelo de comercialização energética P2P em blockchain, conciliando as ideias propostas na literatura e trazer a adoção de tal sistema mais perto da realidade.
- Descobrir e propor uma solução razoável de privacidade para proteger os dados dos participantes e, ao mesmo tempo, não esconder seus atos maliciosos na

rede.

- Descobrir e propor um mecanismo razoável para lidar com a quantidade de dados gerada por medidores inteligentes, mantendo a integridade da rede e permitindo escalabildade.
- Avaliar a performance de um sistema de comercialização de energia P2P em blockchain, com soluções de privacidade e escalabilidade para verificar a possibilidade da sua implantação.

Metodologia

Este trabalho foi realizado através de uma metodologia quantitativa, de natureza aplicada, desenvolvida através de uma pesquisa exploratória contendo revisão da bibliografia e experimentação em um estudo de caso. Para isso, foram levantadas soluções da literatura envolvendo blockchain e mercados de energia. Esses trabalhos foram avaliados em termos de detalhamento nas soluções de privacidade, escalabilidade, arquitetura de mercado, implementação e tecnologia utilizada. Em seguida, um modelo de comercialização de energia em blockchain foi projetado, com anonimização dos compradores e processos de validação da energia ali transacionada. Uma análise acerca da performance de tal sistema foi realizada com o prósito de investigar a escala de suporte de participantes, em uma sequência de três rodadas de experimento contendo um ordenador, um *peer* e um ou dois simuladores de sensores, em uma rede *Hyperledger Fabric*. O experimento foi executado na estrutura de computação em núvem da Amazon Web Services (AWS) para testes com diferentes capacidades computacionais. Com base nos dados provenientes dos experimentos, discussões e conclusões foram elaboradas sobre o modelo proposto.

Resultados e Discussão

Os principais resultados decorridos da realização dos experimentos e análises estão descritos a seguir:

- A implementação e execução do modelo proposto foram analisados em termos de performance. Primeiramente, os dois possíveis bancos de dados suportados pelo *Hyperledger Fabric* foram comparados considerando as demandas no modelo desenvolvido e o *LevelDB* demonstra melhor performance, respondendo a consultas mais rapidamente que o *CouchDB*.
- Explorando diferentes configurações da rede *Hyperledger Fabric* e variando a capacidade do hardware que executava o modelo implementado nas três fases experimentais, a rodada mais bem-sucedida suportou 5000 sensores, 5000 compradores e 5000 vendedores transacionando na rede, o que indica a viabilidade

do modelo em um contexto de bairro.

- A utilização de armazenamento foi medida e avaliada, levando à conclusão de que o modelo proposto deveria ser complementado com alguma solução de resumo dos dados após determinado tempo, visto que os mesmos perdem a importância com o passar do tempo e em um ano um *peer* e um *orderer* gerariam *22.7* Terabyte (TB).
- Com base na demanda por processamento, armazenamento e os custos da estrutura de computação em núvem da AWS, uma estimativa de custo foi calculada e comparada com o custo de transação da rede Ethereum, que apesar de distinta do *Hyperledger Fabric*, é uma tecnologia relevante e presente em muitos trabalhos. O custo estimado foi de $9.92 * 10^{-6}$ United States dollars (USD)/transação, enquanto o custo considerado na rede Ethereum foi de 0.5 USD/transação.
- Comparando aos trabalhos relacionados, o modelo proposto trouxe mais clareza e detalhamento sobre a relação entre blockchain e mercados de energia, cobrindo aspectos como privacidade, escalabilidade, profundidade experimental e dados empíricos. A comparação mais pertitente foi com o trabalho de (BLOM, 2018) que utilizou Ethereum para implementar um mercado de energia com 600 participantes, enquanto o protótipo deste trabalho suportou 15000, somando o número de sensores, compradores e vendedores. O custo da rede aqui analisada também foi significativamente mais baixo.

Considerações Finais

Este trabalho propôs, implementou e analisou um merdcado de energia em blockchain com validação a partir de dados provenientes de sensores IoT. Isso é garantido por um contrato inteligente executado por múltiplas organizações e que requer um quroum mínimo delas para considerar uma geração de energia válida. A implementação protege o padrão de consumo dos compradores de energia por meio de um algoritmo de k-Times Anonymous Authentication (k-TAA). A performance e o custo do modelo foram avaliados e comparados com outro trabalho relevante que utilizou Ethereum como plataforma de implementação. Tal comparação indicou que este trabalho obteve melhores métricas. Como contribuições secundárias, métodos de pesquisa diversamente configuráveis envolvendo *Hyperledger Fabric* foram apresentados. Modificações na implementação do *Hyperldger Fabric* também foram realizadas, visando ampliar o suporte a domínios de aplicações diversos. Além disso, análises dos bancos de dados no contexto do modelo foram apresentadas, podendo contribuir para escolhas adequadas em trabalhos futuros.

Pesquisas futuras devem aprimorar a estimativa de geração energética renovável

com base nos dados meteorológicos originados em sensores IoT. Junto a isso, este trabalho não explorou amplamente configurações da rede *Hyperledger Fabric*, adicionando mais *peers* e *orderers* para avaliação de desempenho, o que pode ser realizado também. Pesquisa com o acoplamento de um banco de dados georreferenciado ao blockchain pode contribuir para a performance do modelo proposto. A participação de sensores precisa ser estudada mais profundamente, visto que neste trabalho eles foram apenas simulados e nenhuma análise a respeito da sua capacidade de participar em uma rede blockchain foi feita.

Palavras-chave: Energia. Blockchain. Performance. Escalabilidade. Anonimidade. Hyperledger.

ABSTRACT

Renewable energy use has increased with environmental concerns due to the pollution generated by energy sources like coal and oil. Even though the cost of renewable energy was initially much higher than power from dirty sources, the gap in cost has been decreasing. With lower prices, people install solar panels to reduce their electricity bill or, in some cases, even sell the surplus generated energy to the grid and earn credits from the grid operator. When people sell power to the grid, they are named prosumers. Generally, prosumers are limited to trade the energy they generate with the grid company, dominant in price determination. Decentralized energy markets might increase both market competitiveness and incentive to further people's adoption of renewable energy. Also, a centralized energy market presents security vulnerabilities and a lack of resiliency. In this context, blockchain is a widely studied technology to provide decentralization for energy markets, mainly because of blockchain‘s capabilities of being a cyber-resilient, immutable, transparent, and secure distributed database. The literature shows many solutions to coupling blockchain and energy markets, but much research is still needed to enable it. Scalability, privacy, market design, and user security are some of the open research topics of this kind of application. Hyperledger Fabric predominantly appears in literature proposals of blockchain solutions in the energy markets context, and it is the tool used for the model implementation. This work analyzes the literature related to blockchain and energy markets, proposes a model, implements it, performs experiments, and analyzes network scalability and data generation. The model enables validated clean energy trading with anonymized buyers to prevent consumption pattern exposure. The Hyperledger Fabric chaincode constantly receives sensors data and judges sellers' energy generation claims to be valid or not. For example, sensors capturing wind speed might help prevent dishonest wind power sellers from selling more than they could generate. Once the energy is validated, it can be exchanged among participants. Modifications on Hyperledger Fabric were necessary to implement the defined model. The proposal development is sectioned into three parts: network deployment, chaincode development, and applications development. This work adapted Fabric's implementation to perform scalability experiments with an increasing number of buyers, sellers, and sensors. The experiments consist of three phases with configurations changes aiming to increase the network capacity. The maximum transaction throughput was achieved with 5000 sensors, 5000 buyers, and 5000 sellers. The data generation rate by the network and the baseline deploy costs were also analyzed to judge the network viability. This work brings empirical results on a topic which the literature lacks. Furthermore, the experiment structure serves as a guideline for new research with Hyperledger Fabric, regardless of the application field. Energy engineering researchers' participation is required for enhancing the proposed models' energy validation process. This work explored a limited set of configuration variables, and future works have countless different settings to analyze.

Keywords: Energy. Blockchain. Performance. Scalability. Anonymity. Hyperledger.

LIST OF FIGURES

LIST OF FRAMES

LIST OF TABLES

LIST OF ABBREVIATIONS AND ACRONYMS

AMI	Amazon Machine Image
API	Application Programming Interface
APIs	Application Programming Interfaces
AWS	Amazon Web Services
BRP	Balance Responsible Party
CA	Certificate Authority
CLI	Command-Line Interface
CoAP	Constrained Application Protocol
CPU	Central Process Unit
DLL	Dynamic-link Library
DNS	Domain Name System
DRBG	Deterministic Random Bit Generator
DSO	Distribution system operator
DTLS	Datagram Transport Layer Security
EBS	Elastic Block Store
EC2	Elastic Compute Cloud
GB	Gigabyte
Gbps	Gigabits per second
GiB	Gibibyte
gRPC	Remote Procedure Calls
HTTP	Hypertext Transfer Protocol
IoT	Internet of Things
IP	Internet Protocol
JSON	JavaScript Object Notation
KB	Kilobyte
k-TAA	k-Times Anonymous Authentication
kWh	Kilowatt-hour
MPC	Multiparty Computation
MSP	Membership Service Provider
OU	Organizational Unit
P2P	Peer-to-Peer
PoA	Proof-of-Authority
PoS	Proof-of-Stake
PoW	Proof-of-Work
protobuf	Protocol Buffers
RAM	Random Access Memories
SDK	Software Development Kit
SDKs	Software Development Kits

SI	International System
SSD	Solid-State Drive
SSH	Secure Shell
TB	Terabyte
TCP	Transmission Control Protocol
TiB	Tebibyte
TLS	Transport Layer Security
TSO	Transmission system operator
UDP	User Datagram Protocol
USD	United States dollars
YAML	YAML Ain’t Markup Language

LIST OF SYMBOLS

Ysc	Yearly storage cost (USD)
$GBMc$	AWS charge for monthly gigabyte storage allocation (USD)
Yg	Monthly data generation (GB)

CONTENTS

*

1 INTRODUCTION

This Chapter presents our work motivation on the blockchain energy markets theme, our research questions, and our work objectives. It briefly introduces the concepts, tools, and problems which we deal with during the text.

1.1 MOTIVATION

Blockchain is a technology that enables a decentralized database in a P2P context. It is widely known because of Bitcoin cryptocurrency (RAHOUTI et al., 2018), and its structure is secure against tampering. Blockchain allows transactions between nodes in a safe manner, without a TTP (Trusted Third-Party). It is considered as having the potential to enhance the role of consumers in the energy trading system by increasing security and reducing costs (WANG, N. et al., 2019).

In most energy distribution systems, residences with renewable energy sources can only sell their excess produced energy to the utility company, which impedes broader price negotiation with multiple bidders. Researchers have explored blockchain as an enabler of a decentralized energy trading market, where residences could trade electricity with each other.

Renewable energy adoption, e.g., solar and wind, is considered a relevant action to reduce greenhouse gas emissions. In late 2018, the state of California made installing solar panels mandatory on newly constructed buildings. Another driving factor for its adoption is that renewable energy costs have steeply declined (ORSINI et al., 2019). Even though the California 2018 law was not 100% applied, it represents an upcoming move towards clean and decentralized power.

One relevant experiment with renewable energy is the Brooklyn Microgrid (MENGELKAMP et al., 2018). In this project, the company LO3 Energy installed a microgrid on top of the existing distribution grid and implemented a peer-to-peer energy market on a small scale using blockchain. Until May 2021, the regulations for peer-to-peer energy trading in New York still do not exist, which prevents a broader adoption of the system.

Smart meters are another necessary technology for a decentralized energy system operation. They measure the amount of energy bought and the amount of energy that is sold to the grid. Smart meters, alongside blockchain, allow energy companies and prosumers - residences that eventually sell electricity to the grid - to adopt business-to-business or prosumer-to-prosumer energy trading in real-time with, potentially, no fees (JOGUNOLA et al., 2019).

There are still open research topics on the blockchain energy trading schemes in terms of system design, privacy methods to protect user data, and scalable solutions to deal with the amount of data collected through the smart meters (WANG, N.

et al., 2019). (ANDONI et al., 2019) argues that consumers might resist blockchain use in the energy markets due to lack of privacy and that this context requires a blockchain with low latency and delay. These topics must be addressed to bring this type of system closer to adoption. We focus on analyzing the **scalability** of our blockchain energy trading model with **privacy**-preserving tools.

1.2 JUSTIFICATION

(ALAM, Asraful et al., 2019) argue that a power market in a microgrid could increase the prosumer revenue, reduce energy cost, and enforce efficient energy utilization. A microgrid is an alternative power source when the main grid systems blacks out. Severe weather events that caused blackouts in New York City motivated the implementation of the Brooklyn Microgrid project (MENGELKAMP et al., 2018). In Brooklyn, the microgrid can operate in island mode, even when the primary grid does not provide power, which improves power resilience.

Beyond energy system improvement, the growth in renewable energy production brings environmental benefits. Considering the mentioned advantages and that blockchain is seen as a possible tool for achieving a decentralized energy market in a microgrid, research on the theme can lead to firmer conclusions on the viability of coupling blockchain with energy markets.

(JOHANNING; BRUCKNER, 2019) surveyed blockchain-based energy projects. Even though they recognized the potential of utilizing blockchain to trade energy, they also found how barely documented the existing schemes were. Authors argued that future work must provide scientific depth to convince taking the risk of using blockchain in the energy system.

In (WANG, N. et al., 2019), blockchain-based energy trading solutions were also analyzed. The paper indicates the need for improvements in system incentive, consensus, regulation, tests with the physical layer, and scalability impact. (BLOM, 2018) considered that smart meters might not have enough processing capacity to join the blockchain, depending on the market system's complexity. The sensors data raises another problem currently researched: how does the blockchain environment fit with constrained IoT sensors sending a high load of data to the network.

An example of work that deals with the previously presented problem is (LE-DANG; LE-NGOC, 2019), which uses REST APIs on IoT devices to prevent the issue of high resource utilization by blockchains. Thus, it is notable that research on the blockchain application with IoT to enable decentralized energy trading systems can also lead to significant contributions in the computer security field, since it is a relevant topic.

1.3 RESEARCH QUESTIONS

The research questions of our work are the following:

- At what scale blockchain supports an energy trading scheme?
- How can users' consuming data be protected while still keeping them accountable for misbehaving?
- How can the network guarantee that consumers buy the requested type of energy (e.g, solar and wind)?

1.4 OBJECTIVES

1.4.1 Main objective

The main objective of this work is to propose and analyze a P2P energy market system using blockchain, considering the already proposed decentralized energy market models in the literature.

1.4.2 Specific objectives

The specific objectives are:

- Propose a new blockchain P2P energy trading model, conciliating different ideas proposed in the literature and bring the adoption of such a system closer to reality.
- Find and propose a reasonable blockchain privacy solution that protects user data and, at the same time, does not cover malicious actions in the network.
- Find and propose a reasonable mechanism to deal with the data amount generated by smart meters, keeping network integrity and allowing scalability.
- Evaluate performance metrics of a blockchain P2P energy trading system with privacy and scalability solutions to verify how feasible they are.

1.5 WORK STRUCTURE

The rest of this master's thesis is structured as follows: our theoretical framework is placed in Chapter 2. In Chapter 3, we present our systematic literature review. Chapter 4 shows the proposed model architecture, while Chapter 5 presents how we implemented and deployed it. Performed experiments and their objectives are described in Chapter 6, and their results are discussed in Chapter 7. Chapter 8 concludes our work and indicates possible future work.

2 THEORETICAL FRAMEWORK

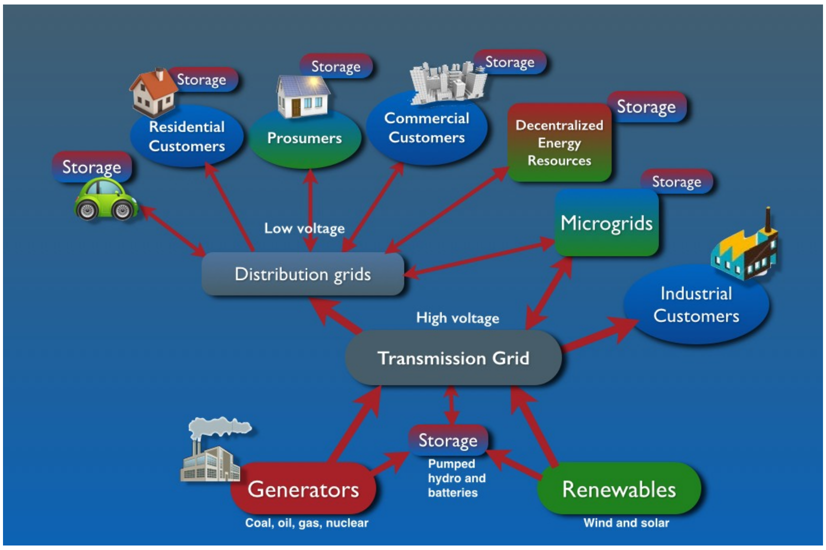
In this Chapter, we will elucidate three key concepts that are part of our work environment. We are going to explain the essential characteristics of microgrids, blockchain, and smart meters.

2.1 GRIDS AND MICROGRIDS

“*Microgrids are stand-alone power networks within small communities using renewable energy and energy storage systems*” (KANG et al., 2018). They are also defined by (ALAM, Asraful et al., 2019) as “*a cluster of distributed resources, loads and energy storage devices within clearly defined electrical boundaries, which offers higher local reliability and flexibility through the integration of energy resources*.” Figure 1 presents the main stakeholders of microgrids.

When microgrids/grids have smart meters as part of their structure, they become smart grids (AVANCINI et al., 2019). Smart meters favor energy automation and reliable power distribution control. They are elucidated in section 2.3.

Figure 1 – Grid/Microgrid actors



Source: (ENERGY, n.d.)

2.1.1 Grid/Microgrid actors

Different actors from the power structure ensure that the energy can flow from the generating companies to the consumers. The main ones are described below.

- **Generating company**: Responsible for generating power through coal, wind, sun, water, nuclear and other types of resources.
- **Ancillary Services**: Generating companies with a fast start and responsible for keeping frequency stability. They can be required by the Transmission system operator (TSO) or Distribution system operator (DSO).
- **TSO (Transmission System Operator)**: It is a company responsible for transmission lines on a country scale, using ultra-high voltage lines.
- **DSO (Distribution System Operator)**: Actor who receives energy from the TSO and distributes it to consumers.
- **Consumer**: Residence, business, or industry that consumes energy.
- **Prosumers**: A type of consumer that also generates energy and sells it to the grid. The most usual form is by installing solar panels on a rooftop of a building.
- **Batteries**: A tool to store generated energy to be consumed in the future.
- **ESS (Energy Store System)**: Entity with the role of storing energy on a large scale. Usually, it works with batteries or storing water in the high ground in the form of potential energy.

2.1.2 Energy markets

Energy markets enable energy trading and help suppliers sell their energy and consumers buy power. Usually, not all consumers are allowed to negotiate in energy markets because they must meet some criteria. Nordpool is a relevant example of the energy market and covers nine European countries. Austria, Belgium, Denmark, France, and Germany participate in this network (POOL, 2020). There are generally three types of energy markets, each one with different purposes. They are defined below based on (PINSON, 2018).

- **Day-ahead market**: Supply and demand offers are negotiated for the following day. Offers are matched through an auction.
- **Intra-day market**: Supply and demand offers are negotiated for the following hours of the day. It closes at least 5 minutes before the hour of the scheduled supply. For example, if consumer A desires to buy 20 MWh for the 15:00-16:00 period, they must close the deal until 14:55. Offers are not auctioned but consolidated by a bilateral contract.
- **Balancing market**: System operators (TSO or DSO) negotiate with ancillary

services to keep power stability close to real-time. It is used for sudden imbalances or when the imbalances could not be resolved through the intra-day market.

Market clearing is a crucial energy trading concept. It represents the process for matching the supply and demand offers, maximizing social welfare. Linear Program is referenced by (PINSON, 2018) as a possible algorithm for market clearing. After this process, a system price is found. Supply offers below the system price, and demand offers above the system price are matched.

2.1.3 Benefits of decentralization

Unlike a centralized distribution, characterized by large power plants generating energy in a centralized, monopolistic way, the decentralized energy schemes propose a fairer system. The traditional centralized energy model is not efficient, has high costs, presents problems with security/privacy, and its development has met a bottleneck (WANG, N. et al., 2019).

The first advantage of a decentralized scheme is that energy is produced near the consumers, implying less power loss due to transmission distance. Decentralized grids also enable trading between small producers like prosumers. Otherwise, the prosumers could only trade with the DSO, usually by a price or credit established by the DSO, in a centralized way.

In a market with more players, prosumers are more likely to receive a better value for their produced energy, which raises the incentive for adopting clean energy and reducing greenhouse gas emissions. Lastly, a decentralized energy system improves power resilience because any generation problem at a large power plant can be mitigated by the local prosumers supplying energy.

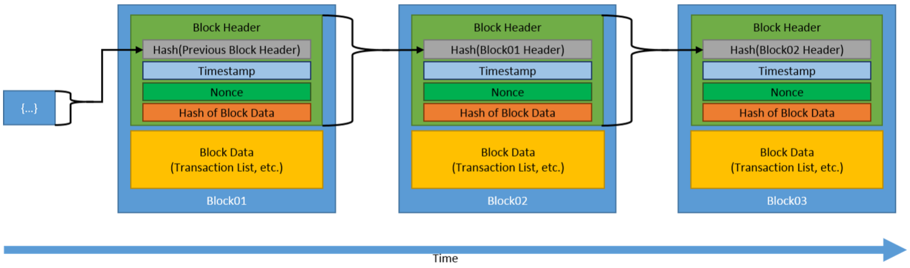
2.1.4 Challenges of decentralization

The increasing number of energy market participants due to energy market decentralization escalates the trading complexity because instead of only trading with the DSO, prosumers can trade with each other. The intermittence of solar and wind generation by prosumers also represents a challenge since it makes it harder for the energy system, as a whole, to keep stability. Clouds suddenly covering the solar panels or wind stoping are typical examples of instability causes. The author of (KAUR et al., 2016) deals with this problem by designing a forecast model.

2.2 BLOCKCHAIN

Blockchains are distributed ledgers, usually without a central authority, with a tamper-resistant and tamper-evident structure (YAGA et al., 2018), enabled through public-key cryptography. This technology became well known for being part of Bitcoin currency. Figure 2 shows how blockchains are formed. Each block is formed by a header and a data segment. The transaction list is part of the data segment, while the block's cryptographic hash, the previous block's cryptographic hash reference, and a timestamp are parts of the header segment. The nonce represented in the figure is not always essential but depends on the consensus mechanism, which will be further explained. Each transaction performed by a node is digitally signed and can be verified by all nodes using the public key.

Figure 2 – Blockchain general structure



Source: (YAGA et al., 2018)

2.2.1 Categorization

There are two main categories of blockchain: permissionless and permissioned. Permissionless blockchains allow anyone to join the network, reading, and writing to the ledger as wished. Permissionless blockchains are usually open source. Their consensus rewards publishing protocol-conforming blocks and requires some expense - work or stake - to validate blocks. These consensus constraints exist in permissionless blockchains due to unknown user participation, who might act maliciously without possible accountability (YAGA et al., 2018).

Permissioned blockchains have some restrictions on who can take certain actions in the network. The level of restrictions can vary according to the permissioned blockchain policy. The ledger might be public for reading, but it may have access control for transacting. These settings depend on the context where the network is applied. With known users, the network consensus algorithm can be lighter than permissionless ledgers (YAGA et al., 2018).

2.2.2 Consensus

Every node of the blockchain network stores the public ledger locally. Therefore they must agree on the state of the ledger. Otherwise, each node would form new blocks following different criteria, and the network would become inconsistent. Consensus algorithms are mechanisms to ensure that all nodes on the network (or most of them) agree on the state of the legder. As an example, Bitcoin uses Proof-of-Work (PoW) consensus algorithm.

Generally, in PoW networks, each newly created block's hash *H* must meet determined criteria to be considered valid. The criteria might be that *H* is less than a number *X*, or *H* must start with a number of zeros (YAGA et al., 2018). The nonce plays an essential role in the process of meeting the criteria. Miners, nodes that try to find the next valid block, calculate the block's hash following the formula *H = hash(transaction list + nonce)*. The nonce is changed until *H* meets the criteria. The miner that finds a valid hash for the block is usually rewarded.

PoW consensus takes time and resources from nodes. This is good in a sense because once a miner finds the valid block number *Y*, the smartest decision for the other miners is to immediately try to mine the next block *Y+1* since most likely they will not earn rewards for the block *Y*.

However, this approach has limitations, like high energy consumption and low throughput of transactions/second (WANG, S. et al., 2019). Proof-of-Stake avoids high energy consumption because instead of requiring work, it requires a stake (coins) "deposit," which can be taken away if the block validator behaves against the network protocol.

2.2.3 Smart contracts

At first glance, blockchains might seem to be only a ledger to record data, but smart contracts allowed the technology to go beyond that. Smart contracts are a collection of code and data deployed on the blockchain ledger with deterministic execution capabilities, enabling distributed applications.

A participant node deploys a contract in the network, and other nodes might call functions from this contract. The deployer programs the contract's logic as wished and might impose restrictions on who can call and execute the contract's functions.

The contract's programming language is highly attached to the blockchain tool (e.g., Ethereum, Hyperledger), and will be discussed more deeply in subsection 2.2.4. In terms of cost, the contract can be charged by the miners/executors based on its complexity, which also will be discussed below.

2.2.4 Tools

There are different blockchain implementations with different architectures. The most cited by the related work in our literature review are Ethereum and Hyperledger. Both will be explained below.

2.2.4.1 Ethereum

Ethereum is a blockchain platform with a native cryptocurrency called Ether and supports smart contracts, which are programmed in the Solidity programming language. This language was designed to target the Ethereum Virtual Machine (EVM). The main goal of Ethereum creation was to overcome Bitcoin's scripting language limitations and enable Turing-complete applications in a blockchain (VUJIČIĆ et al., 2018). PoW is the consensus algorithm in Ethereum main net, and users pay miners transaction fees using Ether to encourage the transaction execution.

Accounts represent users and smart contracts identities in the Ethereum network. *Externally owned accounts* identify users, and *contract accounts* identify contracts. The network uses public-key cryptography to create accounts. Creating an account is equivalent to generating a key pair: a private key and a public key. The account address is derived from the public key. Each account has a balance, while contract accounts also have contract data storage. The network saves this data - the current ledger state - and updates it every time a new block is mined (TRÓN; JAMESON, 2020).

Deployed smart contracts might be programmed maliciously to harm the blockchain. Programming an infinite loop on a smart contract could potentially halt the network. Ethereum uses the *gas* mechanisms to prevent this type of attack. Gas is the fundamental unit of computation. Usually, one gas represents one computational step.

When a peer wants to perform a transaction calling a smart contract function, they must explicitly inform the maximum transaction *gas*, limiting the number of the computational operations taken by the miner. Also, the peer must tell in the transaction how much they will pay for each gas (*GASPRICE* parameter). This tool discourages malicious behavior from peers since they must pay miners to execute each computational step (VUJIČIĆ et al., 2018).

2.2.4.2 Hyperledger

A set of blockchain frameworks forms Hyperledger. Burrow, Fabric, Indy, Iroha, and Sawtooth are among those frameworks. Each of them has its purpose and design goals, and we will focus on the Hyperledger Fabric, considering its adaptability for a wide application range (BLUMMER et al., 2018).

Linux Foundation guides the development of Hyperledger Fabric to improve prior permissioned blockchains limitations. Some of those limitations are related to the consensus protocol, contract language inflexibility, denial-of-service prevention from malicious contracts, and smart contract confidentiality (ANDROULAKI et al., 2018).

Hyperledger Fabric introduces a blockchain architecture to achieve resiliency, flexibility, scalability, and confidentiality, enabling applications written in a standard programming language. In 2020, Fabric was supporting smart contracts written in Go, Java, and JavaScript programming languages.

The major differences between Fabric and Ethereum are the lack of built-in cryptocurrency (e.g. Ether) and the *execute-order-validate* approach instead of *order-execute* in Ethereum. Hyperledger provides a very modular architecture.

In Ethereum, the smart contracts are public and, to validate a new ordered block created by a miner, all nodes execute the contracts' called functions (order-execute). Whereas, Fabric consensus protocol follows a different structure. There are three types of peers: endorser, orderer, and validator.

First, a client submits a transaction proposal for endorsers. The endorsers execute the transaction and send back the transaction with the result to the client, digitally signed by them. Then, the client sends the signed transaction to the orderers.

The orderers follow a protocol to agree on which transactions will form the next new block. When they achieve agreement, orderers propagate the new block through the network, and all peers validate it. The validation process follows three sequential steps.

- Validators check if all transactions followed the endorsement policy, requesting that at least X endorsers sign each transaction.
- Validators check if there is any conflict on the ledger state version used by the transactions.
- If the transaction meets the two mentioned criteria, the ledger is updated with the transaction result.

Hyperledger Fabric form ensures more smart contract confidentiality and transaction throughput than Ethereum since only some endorsers know and execute a specific contract. This represents an advantage over Ethereum, mainly due to the permissioned architecture of Fabric.

In Section 2.5, we will present a more in-depth view of Hyperledger Fabric. This tool deserves an exclusive section because it is the one in which we perform our experiments.

2.2.5 Side-chains

The side-chains structure allows transactions occurring off the main chain, like Ethereum's main chain, and improves the scalability of blockchains, which is fundamental in scenarios where IoT devices are part of the network (JEON; HONG, 2019). The efficiency can be obtained because the side-chain might have its design in terms of block structure, consensus, block time, or other characteristics. Plasma is a framework for using side-chains in the Ethereum environment. Periodically, the Plasma chain can send the Merkle roots of the transactions to the main chain.

Ethereum's main net uses Proof-Of-Work consensus, which is costly and slow. A Plasma side-chain can achieve much higher transaction throughput by adopting a more efficient consensus algorithm, like Proof-of-Authority (Proof-of-Authority (PoA)) or Delegated Proof-of-Stake, thus providing a faster blockchain network (ZIEGLER et al., 2019).

2.2.6 Privacy

A Blockchain's important characteristic is the immutability and the possibility to verify performed transactions on the ledger. However, blockchains face challenges for ensuring the privacy of network users' data and complying with privacy regulations. Solutions like SMPC (Secure Multiparty Computation), Zero-Knowledge Proofs, mixing services, ring signatures, commitment schemes, homomorphic encryption, attribute-based encryption, and secret sharing are possible tools that might be used to improve blockchains' privacy (BERNAL BERNABE et al., 2019).

2.3 SMART METERS

General meters have the primary function of regularly and precisely measuring the power flow into buildings. Their most common and old type is the electromechanical analogical meter. The more modern ones are integrated with digital microtechnology and are called smart energy meters, without mechanical moving parts (AVANCINI et al., 2019).

The first generation of smart meters' main characteristic was to report consumption remotely to the power provider, which helped load-leveling the power network and reduce the human labor necessary to bill consumers. Different from analogical meters, smart meters can send information in short time intervals, like 15 min. Analogical meters only display the total consumption.

A smart meter must have at least one communication interface but might have more than one to ensure communication reliability. It might use Ethernet, power line, ZigBee, Wi-Fi, mesh, cellular, and other types of communication networks (AVANCINI et al., 2019).

A large number of smart meters must be efficiently handled since they generate large loads of data in the order of terabytes (AVANCINI et al., 2019). Because of that, smart grids need an infrastructure with data centers, servers, storage, database, and virtualization systems to handle these data.

2.3.1 Smart meter usage and security concerns on decentralized energy markets

The (AVANCINI et al., 2019) authors consider it feasible to estimate household characteristics from the information captured by smart meters. How many occupants, how long each occupant stays at home, how many electronic devices are there, and the presence of security systems are some of the information that can be estimated by analyzing data sent from smart meters.

This possible inferring on people's behavior represents a security vulnerability that must be considered when dealing with smart meter data. Otherwise, instead of helping energy consumers and producers, they could potentially cause harm.

Smart meters are part of many energy market types and indispensable on a decentralized one. They collect energy flow data, which is used to track the consumption/generation of each consumer/prosumer at a specific time interval.

The authors of (KAMAL; TARIQ, 2019) researched how to provide security solutions on a smart meter infrastructure. They treated light-weight security algorithms as a requirement for this type of device, considering its low computational capabilities.

Those smart meter characteristics must be considered when conceiving an energy market design. If their role on the network overcomes their capacity, the decentralized market will not work.

2.4 SMARTDATA

SmartData is a standardized high-level Application Programming Interface (API) that facilitates IoT-related application development. It gathers a set of relevant attributes regarding data measured by sensors as the unit, spatial location, timestamp, and reliability (MEDEIROS FRÖHLICH, 2018).

Code 2.1 demonstrate how SmartData is represented in JavaScript Object Notation (JSON) format. The field **version** determines if the device is stationary, in version "1.1", or moving, version "1.2". The metric is sensed by the device of identification **dev** and has a **value** related to a **unit**. An **uncertainty** degree about the data might be declared. The coordinates **x, y, z** express the measure absolute spatial location associated with a specific instant represented by timestamp **t**. **Version** "1.2" also supports the SmartData cryptographic **signature** (LISHA, 2020a).

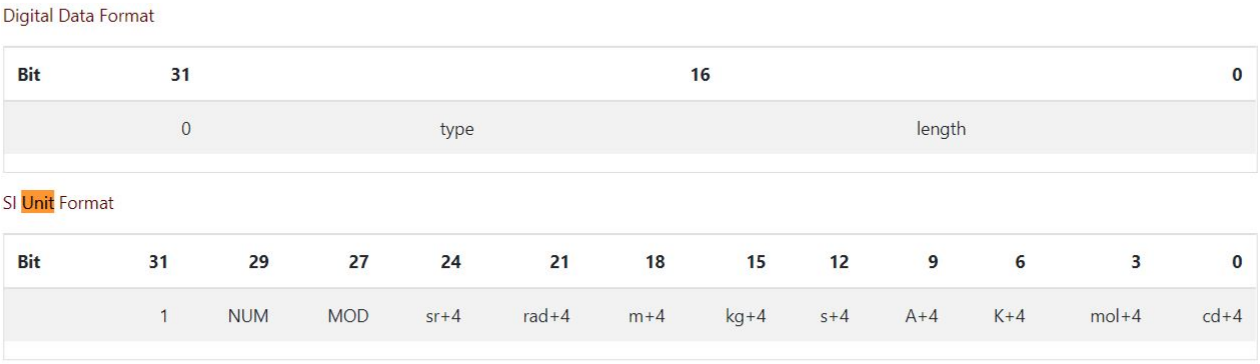
Code 2.1 – Smart Data fields

```
{
    "version" : unsigned char
    "unit" : unsigned long
```

```
    "value" : double
    "uncertainty" : unsigned long
    "x" : long
    "y" : long
    "z" : long
    "t" : unsigned long long
    "dev" : unsigned long
    "signature": string
}
```

Figure 3 presents the organization of the **unit** field. The **unit** bit 31 indicates whether the **value** is digital data - images, audio, switches, and buttons - or an International System (SI) physical measure like temperature, acceleration, electric current, fluid flow, and others. Even though Code 2.1 displays **value** as *double*, the bytes in it might store a *32-bit integer*, a *64-bit integer*, or a *32-bit float* IEEE 754 (COMMITTEE, 2019). The correct interpretation is indicated by the two **NUM** field bits. The **MOD** field determines whether the unit is directly described, represents the ratio of units, is in logarithmic scale, or represents a logarithmic ratio.

Figure 3 – SmartData unit field semantics



Digital Data Format

Bit	31		16		0
	0	type		length	

SI Unit Format

Bit	31	29	27	24	21	18	15	12	9	6	3	0
	1	NUM	MOD	sr+4	rad+4	m+4	kg+4	s+4	A+4	K+4	mol+4	cd+4

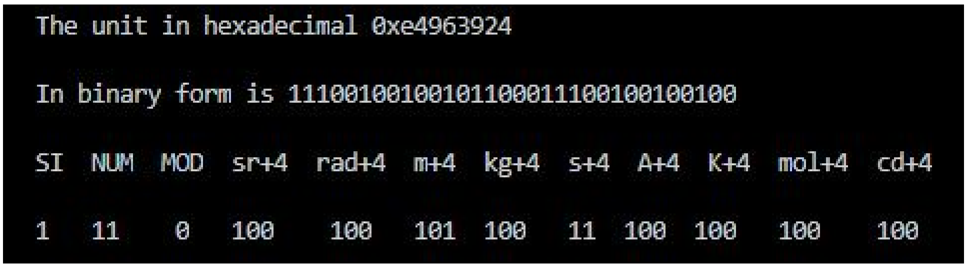
Source: (LISHA, 2020b)

The bits 26 to 0 are divided into unsigned three-bit fields, each representing a SI unit exponent that determines the SmartData unit as a whole (LISHA, 2020b). The exponent ranges from -4 to 3 and can be calculated by Equation (1). Figure 4 provides an example of an encoded unit, with the first line showing this unit is represented as hexadecimal. The following lines have the unit binary representation and the discrimination of each field value.

$$exponent + 4 = unitThreeBitField \tag{1}$$

The bits 31 to 27 indicate that the example measured data is from the SI, encoded as a double IEEE 754, and the unit is directly described. In the interval of bits 26 to 0, all the exponents with the binary value ‘100’ indicate the absence of that SI unit since their value is equivalent to 4 in base ten arithmetic. Therefore, based on Equation (1), their *exponent* is 0. In Figure 4, only the exponents *m+4* (meter) and *s+4* (second) are different from ‘100’, with exponents of, respectively, 1 and -1. This leads to the conclusion that the SmartData unit is a *meter per second* ($m^1 * s^{-1}$).

Figure 4 – SmartData unit field example



```
The unit in hexadecimal 0xe4963924

In binary form is 11100100100101100011100100100100

SI  NUM  MOD  sr+4  rad+4  m+4  kg+4  s+4  A+4  K+4  mol+4  cd+4

1   11    0   100    100   101  100   11  100  100   100    100
```

Designed by author

2.5 HYPERLEDGER FABRIC

In this Section, we explain the key concepts of the Hyperledger Fabric network. We intend to give a solid idea about how the network works without bringing all the details, which can be found in Hyperledger Fabric’s documentation (TEAM, F. D., 2020a). We first present the network elements separately. Then in Section 2.5.10, we summarize how all those elements work together.

For more practical information on the tools used to implement Hyperledger Fabric and how to start a Fabric network refer to Chapter 5. In this Chapter, we focus on the theoretical architecture and principles.

2.5.1 Main differences between Ethereum and Bitcoin

Hyperledger Fabric’s main differences to platforms like Ethereum and Bitcoin are the **permissioned** structure and distributed transaction execution. In Hyperledger Fabric, the transaction can be validated by only some peers, whereas in Ethereum, all peers validate the transactions.

At first glance, this validation process might seem insecure, but we must remember that in Hyperledger’s model, there is some implicit trust among network participants. Also, the workload of validating transactions can be distributed more efficiently, increasing the network’s throughput, because only a subset of peers can be required to validate each transaction.

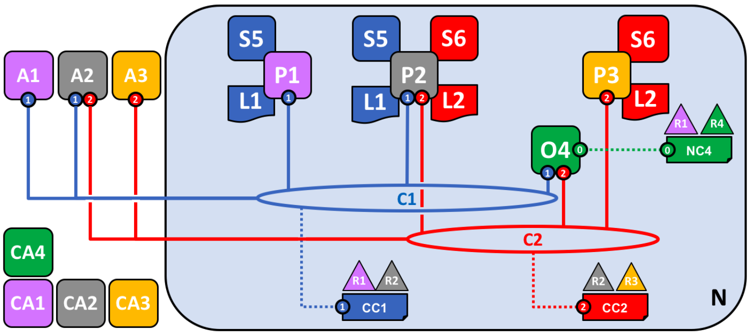
Another difference is the way Fabric stores data. For this purpose, Fabric has two components: the ledger and the World State. The ledger is simply the log of all transactions performed in the network. For example, if an asset was first owned by *User1*, transferred to *User2*, and finally sent to *User3*, all these transactions are registered in the ledger.

The World State only stores the current valid network state. Considering our asset transfer example, the only information present in the World State would be that *User3* owns the asset. No information about previous owners would be stored in the World State.

2.5.2 Network architecture

Figure 5 displays a complete Hyperledger Fabric version *2.3* network containing the main stakeholders. The blockchain core elements are inside the **N** frame, with peers, orderers, smart contracts, and configurations. Outside the blockchain core, applications and certificate authorities participate in the network.

Figure 5 – Hyperledger Fabric network



Source: (TEAM, F. D., 2020a)

- **N**: Network
- **R1, R2, R3, R4**: Organizations represented by different colors
- **CA1, CA2, CA3, CA4**: Certificate authorities
- **NC4**: Network configuration
- **O4**: Orderer
- **C1, C2**: Channels

- **CC1, CC2**: Channels' configuration
- **P1, P2, P3**: Peers
- **L1, L2**: ledgers
- **S5, S6**: Smart contracts
- **A1, A2, A3**: Applications

2.5.3 Organizations

Organizations represent a company, a group of people, or a set of machines. Each member of an organization has their membership validated by *Certificate Authorities*. In Figure 5, there are four *Certificate Authorities* (CA1, CA2, CA3, CA4), one for each organization, even though different organizations may use the same *Certificate Authority*. We must remember that Hyperledger Fabric enables a **permissioned** network in which different members have different roles and permissions. The *Certificate Authorities* have a key role in helping the permission identification of each network user.

2.5.4 Network administration and configuration

The network can be configured to have multiple administrators since creation. However, in Figure 5, only the **R4** organization created the network, and it can add other nodes to manage the network and participate in network policies definition. We can see that just above the **NC4**, there are the organizations **R4** and **R1**. In this scenario, **R4** was the initial network creator and added **R1** as an administrator afterward.

2.5.5 Peers

Peers are members of organizations that can interact with the network and might host ledgers and smart contracts. In Figure 5, they are represented by **P1, P2**, and **P3**. *Applications* (**A1, A2, A3**) hosted outside the network must interact with the network ledger and contracts through a peer. A peer can participate in multiple *channels*, which will be explained in Section 2.5.8.

Information is transmitted among peers through a gossip protocol that continuously discovers other peers, keeps block synchronism, and updates ledger data while maintaining speed. Every organization has a gossip **leader** that communicates to the ordering service to pull blocks (PROJECT, 2020). The leadership can be static or dynamically established through elections. If the current leader stops sending heartbeats, a new election starts.

Every organization also has at least one anchor peer declared in the channel configuration. Anchor peers have information about all peers from the same organization and are responsible for passing it to other organizations. For example, assuming that Figure 5 omits a peer **P4** from organization **R3** and **P3** is an **R3** anchor peer, **P3** will inform **P2** about **P4's** existence if they gossip. Otherwise, peers from different organizations would never gossip or know about one another's existence.

A Hyperledger Fabric peer can have two types of assignments in the network:

- **Committing peer**: A node that receives blocks from the orderer validates and commits them to its local ledger. Every peer is a committing peer.
- **Endorsing peer**: A node that hosts a smart contract and endorses transactions sent by client applications (This type of peer is further explained in Sections 2.5.8.1 and 2.5.8.2).

2.5.6 Orderers

Orderers are responsible for enforcing the consensus on the network. They receive transactions from applications and form blocks sent to peers, resulting in the network consensus since the same block will be replicated in every peer. The ordering service acts according to the NC (*Network Configuration*) file.

The orderers participate in one common channel, called *syschannel*, to order the blocks. Before each orderer is initialized, they usually receive a *genesis block* file built based on the NC (*Network Configuration*) file. The information about certificates, organizations, consortiums, and orderers' hosts is in the *genesis block*.

2.5.7 Consortium

A consortium is defined as a set of network members who need to transact with one another. In Figure 5, it is possible to see two consortiums. One is formed by **R1** and **R2**, while **R2** and **R3** form the other consortium. All consortiums must be defined by network admins in the *Network Configuration* file, represented by **NC4** in Figure 5.

2.5.8 Channel

After the consortium's creation, it is possible to set up a channel for its members. The channel serves as a private communications mean among the consortium's participants. There is one individual ledger to each channel. Only peers from the channel's organizations can host and execute smart contracts.

We can verify these characteristics in Figure 5 by looking at the organizations' and peers' colors. In channel **C1**, peer **P1** is a member of organization **R1**, and peer

P2 is a member of organization **R2**, as they match colors.

Only channel members can define the channel configurations through the *Channel Configuration* file, represented by **CC1** and **CC2** in Figure 5. Channel admins can decide dynamically what ordering nodes can participate or must abandon the channel blocks' ordering.

2.5.8.1 Smart Contracts (Chaincode)

Smart contracts are programs that enforce rules and business models while members transact in the network. These contracts receive transaction requests with inputs and may reply to the requester or modify the ledger. In Hyperledger Fabric version *2.3*, each smart contract might be programmed in Go, JavaScript, and Java programming languages.

The chaincode is formed by a set of smart contracts. A peer can host multiple chaincodes, such as one for channel **C1** and another for channel **C2**. It is also possible that the same chaincode can modify multiple ledgers, if multiple channels host it.

The smart contracts within the same chaincode share the same State, or, for better understanding, the same "memory space." Suppose a smart contract A within the same chaincode as smart contract B writes to the variable C. In that case, the smart contract B can also access variable C. Even though it is possible to have two or more smart contracts in a chaincode, the most common practice is to put only one smart contract for each chaincode.

2.5.8.2 Transactions and Policies

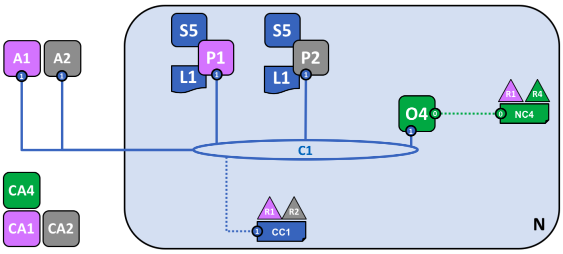
Hyperledger Fabric does not require that every peer validate every transaction. Instead, the validation process follows the endorsement policy. Let us consider the following scenario in Figure 6. An external application, **A1**, wants to call a smart contract function that performs a simple change in the World State. The **C1** channel's endorsement policy states, "every transaction must be signed by at least **two** peers to be considered valid."

To effectively call the smart contract function, the external application will have to send the transaction proposal to **two** different peers (**P1** and **P2**). The peers will receive the proposal, simulate the transaction, and sign a response with the resulting World State change.

After collecting all signatures, the application sends the transaction to the ordering service **O4**. When **O4** decides to form a new block for channel **C1**, **O4** will propagate the block for **P1** and **P2**. Then, finally, the peers will modify their local ledgers and the World States, after checking the transaction validity.

Every transaction has a Read/Write key set, informing the read and modified states. A transaction might be invalidated because some read state was modified by another transaction ordered before. As an example, consider that transaction A reads state X and writes to state Y. Simultaneously, transaction B modifies X. If transaction B reaches the orderer before transaction A, then transaction A will not be validated because it read the old X value. We elaborate on the ordering service block creation in Section 2.5.9.

Figure 6 – Hyperledger Fabric network with a **single** channel



Source: (TEAM, F. D., 2020a)

2.5.8.3 Private data collection

Private data collection allows private transactions among members of the same channel without creating a new one. The *transaction* flow for private data is different from regular transactions. First, the client application sends the proposal for the endorsing peers (private data collection peers), which endorse the transaction, save the transaction result in a *temporary data store* and respond with a signed proposal. The proposal only contains hashes about private data keys and private data values.

The client application forwards the endorsed proposal to the orderer. When a new block is created, the private data collection peers will either apply the change saved in the *temporary data store* or fetch the data from another peer.

2.5.9 Consensus

The PoW consensus, used by Bitcoin and Ethereum, is not deterministic due to the lack of guarantee that a mined block will be accepted as the next block by the whole network. In other words, there is the possibility of chain **forks** happening. A fork occurs when two miners generate two different valid blocks simultaneously, and the network's nodes are not clear on which one should be considered the next block. Usually, forks are resolved by waiting for further blocks to be added to the chain. The longest chain is considered the correct one.

In Hyperledger Fabric, there is no such complexity since the consensus process is deterministic. This is achieved through tools like **Raft** and **Kafka**, executed by the **orderers**. Both tools are crash fault-tolerant and meant for environments with certain trust among stakeholders.

Raft is the recommended consensus in Hyperledger Fabric version 2.0, as **Kafka** is deprecated. Both tools have a *leader and follower* architecture. In section 2.5.9.1, we describe the consensus idea, and in section 2.5.9.2, we describe how **Raft** enables this consensus idea.

2.5.9.1 Consensus general idea

Considering a network with four orderers, each orderer receives endorsed transactions from applications, and the received transactions set will probably vary among the orderers. Because of that, they must have a process to agree on the transaction sequence of the next block.

To ensure consensus, Hyperledger Fabric orderers follow a *leader/follower* protocol. The four orderers must elect a leader, and the remaining three orderers are classified as followers. The followers send their received transactions to the leader, and the leader decides on the transaction order of the next block. After that, the followers receive the next block from the leader and send it to peers.

When peers receive the new block, they validate each transaction following the endorsement policy and considering the World State modifications. Every peer follows the same validation process, which ensures that the ledger changes are the same.

All Hyperledger Fabric current consensus implementations are crash fault-tolerant (CFT) but not Byzantine fault-tolerant (BFT). A CFT consensus guarantee that the network will function even if some nodes crash, while a BFT consensus guarantee network normal function even if some nodes act maliciously (PODGORELEC et al., 2019)

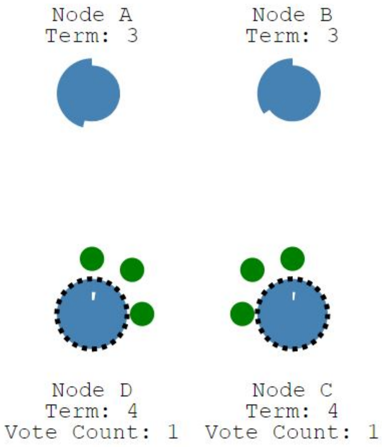
2.5.9.2 Raft

Raft is also a crash fault-tolerant consensus tool for the Hyperledger Fabric, and, different from Kafka, which Apache implements, Fabric has its native implementation of Raft. It follows the *leader/follower* architecture, and it is embedded into each orderer.

All nodes start as *followers*. If a node does not receive a heartbeat or blocks from the leader after some time, it enters the *candidate* state when a node asks other nodes for votes. If the candidate collects enough votes to satisfy the channel *quorum*, it is considered a *leader*. In the case of a leader crash, another leader is elected by the same steps.

Figure 7 shows the raft election beginning with four nodes. Each Raft node represents a single orderer. *Node D* and *Node C* have the timeout expired at the same moment. The green circles represent vote requests sent to other nodes.

Figure 7 – Raft election example



Source: (DATA, 2020)

After the election, all transactions received from client applications by followers are routed to the leader. After some time or after a collected transaction quantity, the leader orders transactions, forms a block and sends the block to followers. The followers and the leader forward the new block to the peers.

Each channel selects a *consenter set* of orderers. There is one instance of the Raft protocol **for each channel**, which implies one different election, one different leader, and a separate consensus. Even though this design choice might seem unnecessarily expensive, the designers claim that it represents the first step for achieving a future Byzantine fault-tolerant consensus in Hyperledger Fabric (TEAM, F. D., 2020d).

2.5.10 Summarizing

First, a node creates the network by defining a *Network Configuration* file and instantiating an ordering service. Other nodes are added following the *Network* policies and are always associated with a *Certificate Authority*. When already in the network, two or more organizations are free to create a consortium associated with a channel. Only the members of the channel define its policies in the *Channel Configuration* file. They decide who can interact with the ledger by reading and writing. Also, they decide policies for administration, ordering, and endorsement.

For example, the channel's administrators might decide that each transaction will require at least half of the peers' signatures. Peers in the network can install a chaincode, which serves as a container for related smart contracts. Nodes can communicate privately using *Private Data Collection* within the same channel, and only private data hash is published publicly to the channel.

When channels begin to generate transactions and data, the orderers group them and assemble blocks. The orderers know nothing about the transaction semantics, as they only verify the endorsement policy and assemble blocks for each channel to achieve consensus.

We presented the most common action flow in a Hyperledger Fabric network, but many other configurations are possible since the network is very modular. For example, in Fabric's version *2.3*, the channel admins can decide what orderers participate in the channel, even though Figure 5 shows only a single orderer.

2.5.11 Idemix - identity mixing

Hyperledger Fabric supports zero-knowledge proof transactions with idemix. Zero-knowledge proof lets a prover P convince a verifier V that P knows a secret without explicitly exposing the **secret** (CAO; WAN, 2020). V makes random requests with some parameters for P to perform calculations and reply with the result. After some rounds of successful requests and answers between P and V, V decides that it is highly probable that P knows the **secret**.

Idemix, briefly mentioned by (CAMENISCH; LYSYANSKAYA, 2004) and specified by (AU et al., 2006), allows that applications transact anonymously in the network. Each organization must define whether it signs with default x509 or idemix. However, the downside of using idemix is that organizations cannot endorse transactions or approve chaincode definitions. Therefore, idemix organizations also cannot have active peers or orderers.

Ideally, a network should have as few idemix organizations as possible because some policies require that most organizations sign a change. If half of the organizations have idemix, changes will be hindered. Idemix provides a way to make all transactions by the same entity unlinkable to one another. To accomplish that, Hyperledger generates a new pseudonym to sign every new transaction. Overall, it is a useful tool if the network requires anonymous transactions from applications.

3 LITERATURE REVIEW

The following questions drove this literature review:

- What are the current solutions presented in the literature about blockchain energy trading?
- What are the limitations of the current schemes?

We performed a search on IEEE (Institute of Electrical and Electronics Engineers) through the query in Frame 1. We selected the 18 most relevant papers to be analyzed through abstract and full-text reading. Two papers were immediately rejected. One of them was rejected due to having a publication date before 2018, and the other because its main focus was vehicular energy trading. We read the remaining 16 papers, plus the other eight relevant works cited by these 16 since their content complemented some aspects not covered by the ones found by query results. We assessed the full text of these 24 papers and selected the most related to our proposal.

Quadro 1 – Research query

```
"All Metadata":"Blockchain" AND "All Metadata":"energy" AND
("All Metadata": "trade" OR "All Metadata":"trading")
```

The authors of (PEE et al., 2019) propose a decentralized energy trading scheme enabled by an ESS (Energy Store System), with the price set by the DSO (Distribution System Operator). Sellers and buyers inform the energy amount to be sold or bought, and an Ethereum smart contract performs the matching. The work provides a general idea of a blockchain energy trading system. However, it is far from meeting the real world’s needs due to its simplicity and not mentioning scalability and privacy.

(KANG et al., 2018) also proposes a blockchain energy trading solution using Ethereum. A contract is created and deployed on the blockchain for each energy transaction between consumer and prosumer. The consumer calls a function from the deployed contract offering a bid to the prosumer. When there is a match between bids, the energy is transferred, and the payment is performed. Even though the last cited work admits to being only a starting idea, it is interesting to mention its weaknesses. There is no market clearing process, and each transaction requires a new contract, which is unnecessary in terms of storage usage. The solution uses a consensus method like PoW in a private Ethereum network, whereas the authors could have taken advantage of the privacy and use a lighter consensus algorithm.

In (LU et al., 2019), a blockchain-based energy trading scheme is designed with two layers. The first layer consists of a private blockchain to support local energy trading in a community context. In the second layer, regional energy aggregators trade energy cross-regionally. The energy aggregators are also responsible for coordinating the transactions locally, acting as a third-party manager.

The work proposes a Proof-of-Stake consensus method based on each node's credit score, and tokens signed by aggregators serve as energy credits to be consumed. The authors performed a mathematical analysis of their system performance, and they did not mention any blockchain tool. Payment off-chain and mixing private and consortium blockchains are considered as solutions to protect user privacy. They suggest improving scalability and performance as to future work.

The proposal in (HUSSAIN et al., 2019) describes a blockchain solution for energy trading between the DSO/Ancillary Services and their clients only. It does not enable P2P energy trading, while it recognizes that P2P trading's complete implementation still faces challenges. The authors advocate using permissioned blockchain tools like Enterprise Ethereum and Hyperledger Fabric in energy schemes, as they allow more efficient consensus algorithms like PoA and Proof-of-Stake (PoS). Lighter consensus methods are more suitable for constrained smart meters.

An Ethereum smart contract is presented. Each new block is added by authorized nodes using PoS consensus with grades calculated from past network behavior. The authors argue that blockchain can mitigate smart meters' communication security vulnerabilities. As a future direction, the work indicates the implementation of a fully decentralized energy trading system.

The authors of (ALAM, A. et al., 2019) suggest a double-chained blockchain energy trading scheme. One chain stores smart contracts that report the power status of a user, and the other chain enables energy negotiation. The work argues that a decentralized blockchain energy trading scheme provides cyber resilience, eliminates monopoly, is transparent, and provides security. Research on optimizing the consensus and protecting the network against DoS (Denial of Service) is mentioned as future work. The paper describes only a generic blockchain energy trading scheme.

On (DORRI et al., 2019) proposes a P2P energy trading scheme through an Ethereum private blockchain. On the scheme, the producer pays or requests for the network authorities to join the network. Once in the network, the producer deploys a smart contract to keep their price and available energy amount updated. The consumer performs a *Commit to Pay* transaction, putting the money on hold until the producer releases the energy. When energy is delivered, the consumer/buyer informs the network through an *Energy Receipt Confirmation*, and the funds are transferred to the producer. Transaction fees reward the block miners and serve as an incentive.

The authors present a proof-of-concept with two Raspberry Pi, using a Python extension to interact with the Ethereum blockchain and using Ether cryptocurrency as a payment method. They evaluated the system behavior with reliable and unreliable nodes. The authors also presented performance metrics like end-to-end delay, monetary cost, transaction throughput, and blockchain size.

The work of (KODALI et al., 2018) presents a blockchain P2P energy trading scheme implemented with Hyperledger Fabric. It considers three main stakeholders: energy nodes, energy aggregators, and smart meters. Residents can trade with the utility (DSO) or with other residents. The architecture has its own coin that can be converted to fiat money. The energy aggregator acts as a broker and manages the trades. Even though the scheme is not very detailed, the work describes and classifies different Hyperledger services. Fabric, Iroha, and Sawtooth are compared with each other in consensus algorithm, consensus approach, and advantages.

The paper (WANG, S. et al., 2019) brings a more detailed description and a more in-depth analysis of blockchain decentralized energy markets if compared to the previously described papers. It shows a market structure closer to real centralized solutions like Nordpool (POOL, 2020). The market is split into two phases: the day-ahead and the real-time market, explained in Section 2.1.2.

In their system, prosumers are classified as Type 1 or Type 2. Type 1 prosumers submit entirely to the power operator (DSO), and negotiations between them occur in the day-ahead market. Type 2 prosumers can trade with the DSO and between each other, but only in the real-time market. Since the authors' predominant areas are electrical and electronic engineering, the scheme focuses on Optimization Power Flow (OPF) solution and energy distribution.

They chose Hyperledger Fabric as an implementation tool because it met their requirements. The authors wanted a permissioned chain restricted to consumers/prosumers within the distribution area, efficient smart contract execution, practical consensus, and a model that protected users' privacy. It is essential to highlight that they did not consider any other privacy protection mechanism beyond simply using Hyperledger Fabric. Even though they propose a decentralized market system, the DSO still plays a central authority role.

In (JEON; HONG, 2019), the blockchain energy trading model introduces side-chains (Plasma and Plasma Cash) on Ethereum to solve scalability problems and address smart meter computational constraints. Smart meters act as automated agents to trade energy. The energy trading process between microgrid's participants happens on the side-chain, and the Merkle root of each block on the side chain is published in the main chain (Ethereum main net).

There is a centralized operator responsible for managing the side-chain. The authors cite higher throughput, reducing main net use, and the reliance on the main net as major advantages for their model. They claim that future research should design real use cases of microgrid energy trading.

The authors of (HUANG et al., 2019) focus on the blockchain energy trading system's IoT part. They propose a proof-of-concept with Sigfox for smart meter communication and Ethereum as the blockchain tool. On their solution, smart meters send information and requests directly to the Sigfox Cloud, and blockchain miners are responsible for retrieving the data and publishing it on the chain.

The main contribution of (HUANG et al., 2019) is on IoT communication. They tested the Sigfox technology communication range and concluded that in a 1 km range, the Sigfox delivery success rate was 100%. The integration with the blockchain part of the system is mentioned as future work.

The Master thesis presented by (BLOM, 2018) evaluates the feasibility of a blockchain energy trading system. It covers aspects like motivations for adopting such a system, (*Norwegian*) regulation, required infrastructure, challenges, desired blockchain characteristics, and implementation. The market design was divided into three parts: day-ahead, real-time, and load curtailment. The market clearing is performed off-chain by a node and verified on-chain.

The three market parts were simulated with Ethereum smart contracts, which are published on Github. The day-ahead simulation and real-time markets were simulated with 600 nodes, while the load curtailment market was simulated with 25 nodes. The author analyzed the cost of the system based on the *gas* spent by all transactions.

Finally, the author classifies eight statements about the proposed scheme feasibility as true, false, or probable. The conclusion briefing was that the feasibility could not be proven, but some good evidence indicates that decentralized P2P blockchain energy trading is feasible. The work gives explicit and implicit future directions. Some are listed below.

- Perform tests on blockchain platforms other than Ethereum.
- Test the proposed system with real computers and smart meters.
- Propose privacy-preserving schemes for the blockchain energy market.
- Improve network scalability.

A privacy scheme with Multiparty Computation (MPC) is presented in (ABIDIN et al., 2018). Their algorithm design is based on blockchain energy trading models, but the solution was implemented in C++ and was never tested in a blockchain environment. The authors affirm that the simulation performed with real energy data from Belgium indicates that the solution is feasible in a blockchain tool. The authors also present performance metrics regarding CPU operations, and their protocol was analyzed in security aspects with the Universal Composability framework. Optimizing the MPC implementation is mentioned as future work.

The work (AHL et al., 2020) analyzes blockchain use in the energy sector regarding technology, economy, society, environment, and institutions in the Japanese context. The authors mention blockchain's technological challenges when applied to energy markets. Throughput, latency, storage, security are some of those challenges. Multi-chain communication, side chains, and off-chain storage are considered solutions for scalability problems.

A case pilot project in Misono, Japan, is presented with ten consumers, five prosumers, and one mall. The stakeholders exchange energy through a blockchain network and three possible power lines. They are equipped with solar panels, batteries, smart meters, and communications systems to interact with smart meters and the blockchain energy market. The chosen platform was Ethereum, with a PoA consensus.

The authors also affirm that privacy measures, such as pseudonymity, are critical next steps in the blockchain and energy integration context. Other research opportunities are enumerated: consensus mechanisms development, sharding, state channels, smart meter blockchain integration (via light clients), and privacy measures.

Our work proposes, implements, and validates an energy trading scheme in Hyperleder Fabric. We ensure the buyers' privacy through identity mixing and analyze the implementation throughput and data generation rate to elucidate the proposal's scalability. Buyers and sellers can exchange only validated energy generated in the past. The chaincode judges the energy generations as valid based on sensors measures periodically published to the chain. Our model considers the participation of utility companies and payment companies to settle the payment of anonymous buyers.

In table 1, we compare the related work characteristics. Cells filled with an *X* represent that the work has such characteristics. In the case of (BLOM, 2018), the *X* in the *Hyperledger* column indicates that Hyperledger was widely discussed, even though the solution was implemented using Ethereum.

The columns with a symbol like "◑" indicate how deeply the work addressed such topics and the quality of their solution for each topic. The symbol "●" represents maximum depth and quality, while "○" represents the lowest depth and quality. For example, (KANG et al., 2018) propose a simple blockchain energy market model and do not cover some stakeholders, like the utility company. They do cover a context with only prosumers and consumers. However, the model of (BLOM, 2018) supports three types of energy markets and considers the utility company. Therefore, in the **Depth of market design** topic, (KANG et al., 2018) is rated "◔", while (BLOM, 2018) is rated "◕".

Related work \ Characteristics	Permissioned	Deals with privacy	Deals with scalability	Implemented	Ethereum	Hyperledger	Depth of market design
(PEE et al., 2019)	X	◔			X		◔
(KANG et al., 2018)	X	◔		X	X		◔
(LU et al., 2019)	X	◑	◑				◔
(HUSSAIN et al., 2019)	X	◔			X		◔
(ALAM, A. et al., 2019)			◔				◔
(DORRI et al., 2019)	X	◔	◔	X	X		◑
(KODALI et al., 2018)	X	◔	◑			X	◔
(WANG, S. et al., 2019)	X	◔	◑	X		X	◕
(JEON; HONG, 2019)	X	◑	◑		X		◑
(HUANG et al., 2019)		◔	◑		X		◔
(BLOM, 2018)	X	◑	◑	X	X	X	◕
(ABIDIN et al., 2018)		◕					◑
(AHL et al., 2020)	X	◑	◑		X		◔
This work	X	◕	◕	X		X	◕

Table 1 – Related work comparison

The work (AHL et al., 2020) was included based on the relevance of its experiments, and it did not come from the review process. Thus, we decided to add the work in this section to serve for further base and comparison with our model.

4 BLOCKCHAIN ENERGY TRADING AND VALIDATION MODEL

We defined a blockchain energy trading model to perform experiments regarding scalability, privacy, and traceability - which are the topics of our research questions. The model is implemented using Hyperledger Fabric, and it will be presented in this Chapter.

4.1 MODEL'S LITERATURE MOTIVATION

The work presented in (WANG, N. et al., 2019) surveyed blockchain energy trading schemes and listed the main challenges of those systems. Among the challenges are low efficiency, privacy protection, and scalability issues. The authors claim that avoiding statistical predictions and behavior model analysis while preserving nodes' rights and interests in the network may be a severe challenge.

On (JOHANNING; BRUCKNER, 2019), the authors evaluate the whitepapers of blockchain-based peer-to-peer energy trading projects. They analyzed the projects' characteristics, transaction elements, and energy ecosystem. The conclusion was that most of the evaluated projects were described too macroscopically and that future research must address this topic with more scientific depth.

Table 1 shows that there is still much improvement room in the blockchain energy trading schemes in terms of privacy, scalability, and market design based, on our literature review. In our model, we address these three research gaps: *privacy*, *scalability*, and *market design*.

4.2 ENTITIES AND THEIR ACTIONS

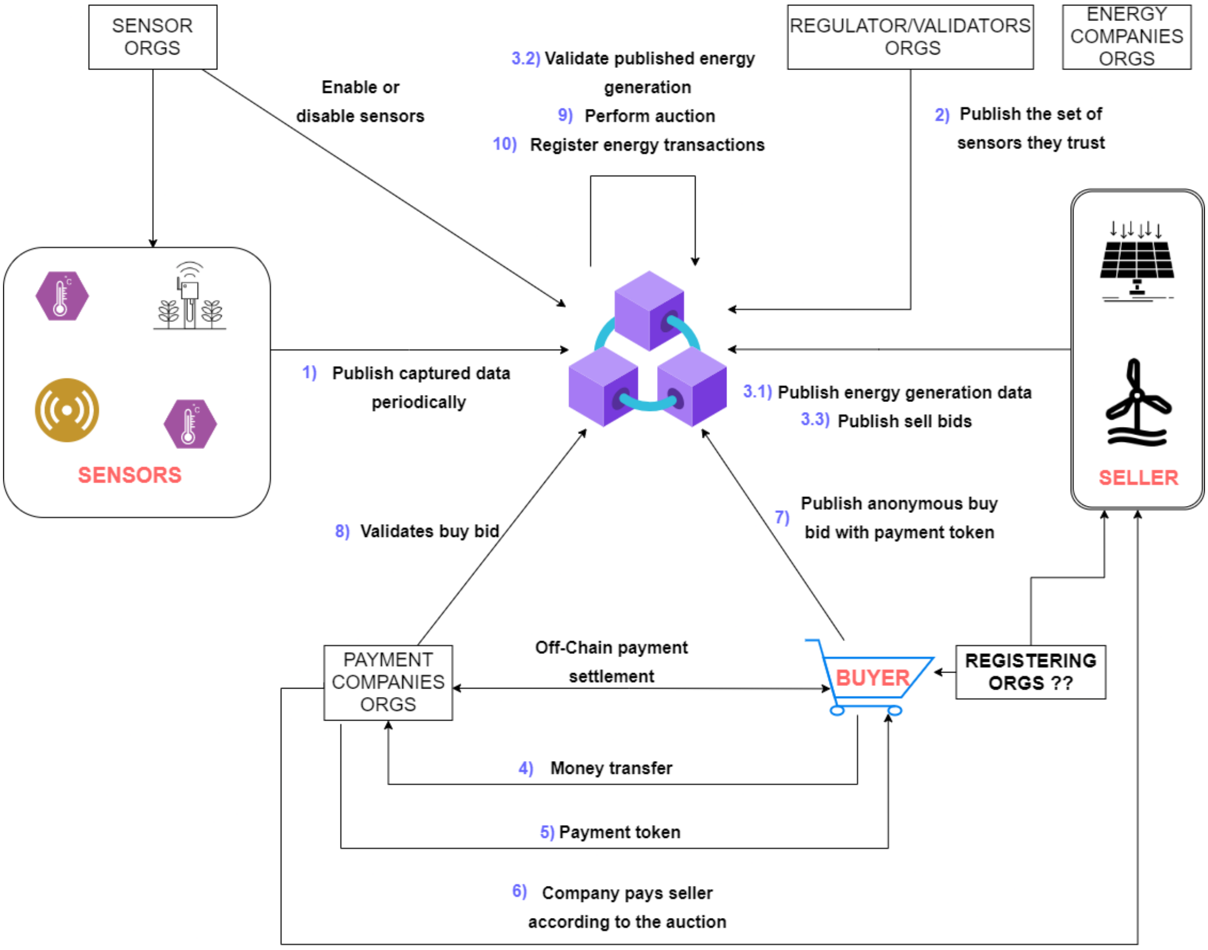
Our model consists of a blockchain network that aggregates five entities: **sensors**, **energy sellers**, **energy buyers**, **validators**, and **payment companies**. Each one is entitled to perform their actions by calling different functions on a **chaincode** (smart contract) related to their roles. Figure 8 presents the main actions of each network entity. Their actions will be described and explained in Sections 4.2.1, 4.2.2, 4.2.3, 4.2.4, and 4.2.5, introducing their behavior in our model.

4.2.1 Sensors

The sensors capture environment metrics and publish them to the blockchain network - action **1** in Figure 8. They can measure temperature, luminosity, humidity, wind speed, air pressure, electric current, and other relevant energy generation metrics. The sensors data is used to validate the energy sellers' generation claims - action **3.2** in Figure 8. This process is described in further detail in section 4.2.4.

Each sensor is registered to network with their spatial coordinates, which en-

Figure 8 – Model entities and their actions



Designed by the author

able the network to infer an environment metric in a specific location in a specific time window. With data coming from many different sensors around a location, the network mitigates the attack of a sensor, intentionally or not, sending incorrect measurements to induce improper behavior.

4.2.2 Energy sellers

Energy sellers generate a specific energy type - solar, wind, hydroelectric or other - and publish their energy generation in the blockchain by invoking a chaincode function - action **3.1** in Figure 8. They might be prosumers or local energy generation companies. The energy generation claims are validated before the energy amount is available for selling.

After the validation, sellers might publish sell bids so that buyers can match it - action **3.3** in Figure 8. The buyer's payment company is responsible for paying the seller after the auction - actions **6** and **9** in Figure 8.

Observation: our model does not define the organization responsible for registering buyers and sellers. We assume that the registering role could be done by regulators or utility companies.

4.2.3 Energy buyers

Energy buyers use the network to buy a desired type of energy. For example, they might be concerned about pollution and want to buy only solar or wind energy. Since the network validates each energy bid, buyers have assurance about the origin of the energy they buy. The energy buyers prove to their energy distribution company that they bought energy in the network and receive discounts on electricity bills.

For example, if the buyer acquires 10 Kilowatt-hour (kWh) on Energy Network and their meter registers the total consumption of 50 kWh, the utility company might charge only 40 kWh. This is possible because the utility company trusts the blockchain network to verify the energy generation since the buyer proves to have bought energy through the network. Therefore, when a buyer proves ownership of the transaction that bought energy, the company accepts it.

The buyer's utility company knows that the buyer consumed 50 kWh based on their smart meter reads. However, when the buyer presents the acquirement proof of 10 kWh in the blockchain market to the utility company, it is only entitled to charge for the difference: 40 kWh.

Buyers publish an **anonymous** buy bid with a token from a payment company - action **7** in Figure 8. With this token, only the payment company **might** know the buyer's identity while guaranteeing to the seller that they will be paid.

4.2.4 Validators and validation

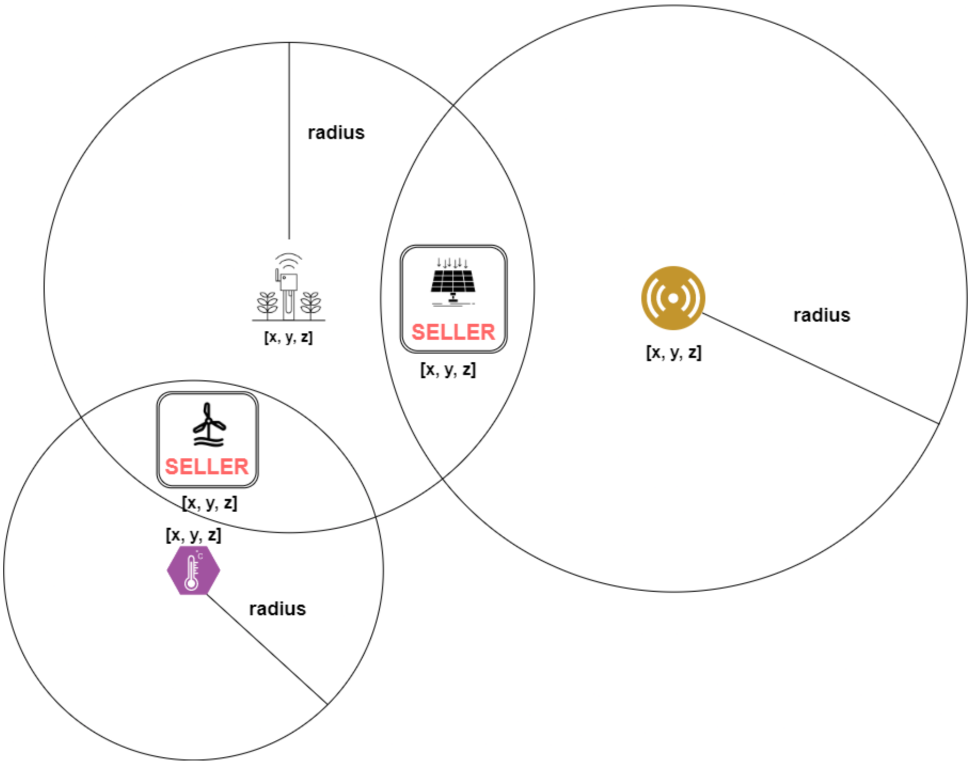
Every time an energy seller publishes their energy generation to the blockchain network, validators use the sensors' data to judge whether the generation is legitimate. For example, a prosumer might have a solar panel that produces the maximum amount of 85 kWh on a sunny day. If this prosumer publishes an 85 kWh generation claim, but luminosity sensors near the prosumer indicate a cloudy wheater, the validators must reject the bid and not endorse it. Even if the prosumer's smart meter indicates that they are feeding the grid with 85 kWh, they might try to trick the energy network by selling dirty energy as clean energy.

State regulators, transmission line owners, private regulators, big energy sellers, or others can perform the validator role. Validators indicate to the network the sensors they trust - action **2** in Figure 8. When an energy seller publishes an energy generation claim, the near trusted sensors help the claim validation - action **3.2** in Figure 8. A minimum number of validators are needed to endorse the energy generation claim.

Figure 9 shows how sensors are selected in terms of location. Sensors have spatial coordinates and a relevance radius that indicates the area where the sensor's captured environment metric is equal or closely similar. The metric sensor unit must be related to the seller's energy type. For example, to validate a **solar** energy generation claim, sensors that measure wind speed should not be selected, but luminosity sensors should.

Our model does not define the precise rules and criteria for validating the energy based on the environment metrics captured by sensors, as it would require knowledge from the electrical engineering field. We only assume that such calculation is possible, and we represent it **symbolically** by averaging the sensors' data near the seller and multiplying it to a constant.

Figure 9 – Considered physical topology



Designed by the author

4.2.5 Payment companies

Payment companies are organizations in the network responsible for settling the payments, off-chain, between buyers and sellers. They receive funds from the buyers to send a token to compose the buy bid - actions **4** and **5** in Figure 8. This token represents a payment guarantee for the seller, who can withdraw the funds presenting proof of transaction.

As soon as the buyer publishes their buy bid, they must request the validation of their bid by the payment company. After the request, the payment company validates the buy bid - action **8** in Figure 8 - by informing the network how much funds the token covers in the buy bid. If a buyer tries to publish a buy bid offering more funds than the payment company claims to cover, the network will not let the bid validation.

The validation avoids token theft and usage by a malicious user since there is no ownership information on the token. Without validation, a peer could read the token, reject the original buyer's transaction, and utilize the token to buy energy for a third party.

Even though the token could be digitally signed by the payment company and reference the buyer credential to avoid the validation step, we opted not to add cryptography. A cryptographic token would create other problems, increasing the processing time due to cryptographic operations and decaying the model scalability. In these conditions, the token would need standardization across payment companies so that the chaincode could process it, increasing the chaincode's complexity unnecessarily.

4.3 ACTIONS FULL SEQUENCE

The sequence diagram presented in Figures 10 and 11 shows a possible action sequence performed by entities. All these actions would happen in a real deployed network, simultaneously with multiple sellers, buyers' payment companies, utility companies, and validators. However, the diagram clarifies the usual action sequence by each entity type.

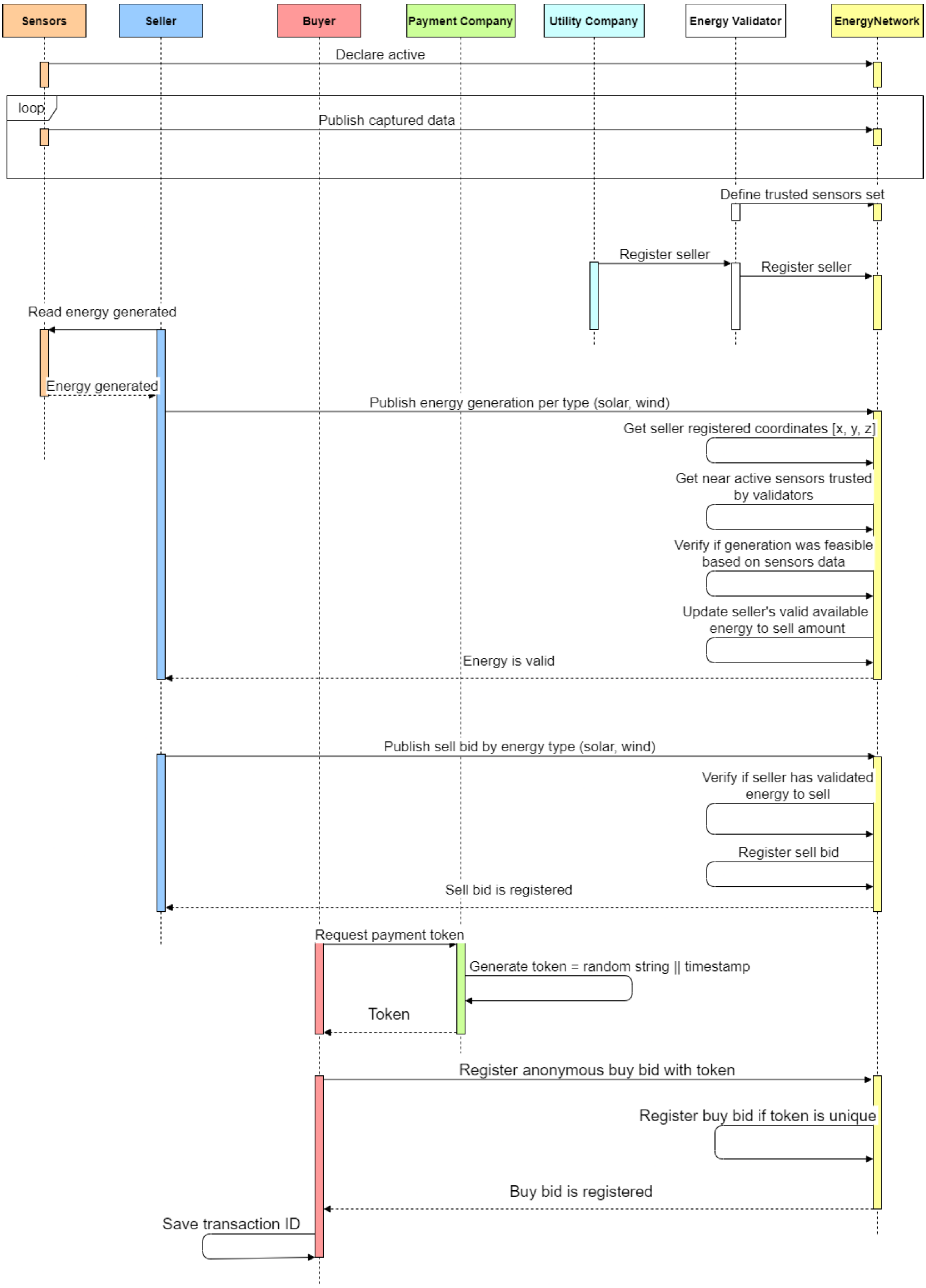
First, each sensor declares itself active to the network and starts publishing its captured data. Following that, energy validators can define the sensor set they trust to be considered in their validation policy. As soon as the seller is registered, they or their automated gateway can publish energy generation claims. The chaincode judges the claims as valid or invalid based on the seller's location and the sensors' published data.

In case the energy generated is ruled valid, the seller publishes a sell bid. A buyer desires to match this sell bid, and they request a token from the payment company before sending a buy bid. The buyer sends the buy bid to the network and requests the bid validation to the payment company, which validates it. After that, the buy bid participates in the network double-auction and matches the sell bid.

The buy bids and the sell bids are matched, and the energy transactions are registered to the ledger. Proving the bids issuance, buyers and sellers might request, respectively, energy bill discounts and payment for the sold energy. The utility company and the payment company respond accordingly after verifying the proofs.

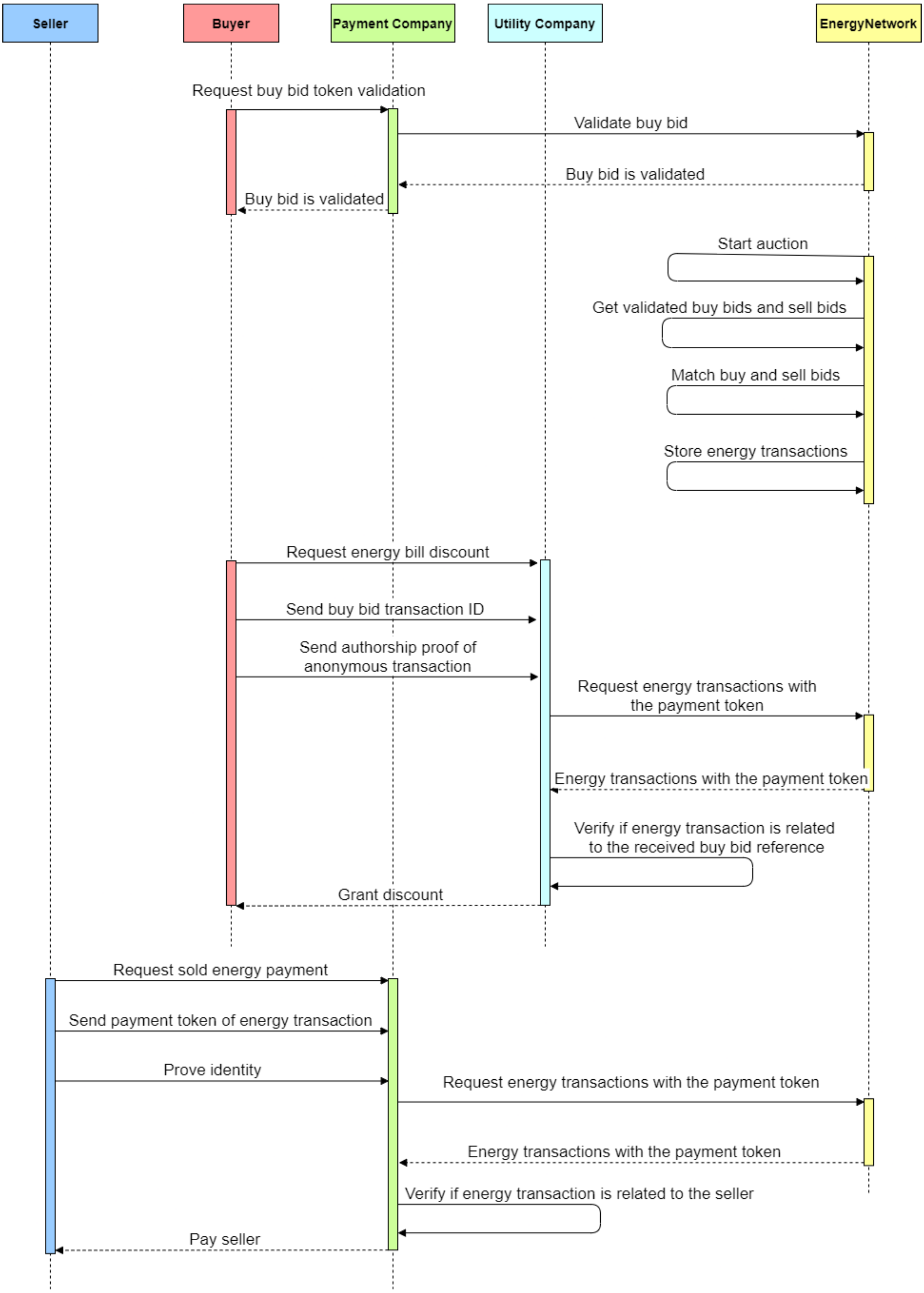
The utility company tries to charge the energy customer for the consumption amount indicated on their smart meter. Nevertheless, if the consumer bought some energy on the blockchain network, they require a discount on their bill after providing the necessary evidence. In the seller's case, they inform the payment token received after selling energy. The payment company verifies if the seller is the designated part of receiving the funds, and then the seller is paid.

Figure 10 – Sequence diagram (continues in Figure 11)



Designed by the author

Figure 11 – Sequence diagram continuation



Designed by the author

4.4 MODEL MAIN CHARACTERISTICS

The proposed model increases trust in the energy sellers because their energy generation claim is verified by multiple regulators, utility companies, or other validators, based on many sensors' collected data. Buyers can have significant assurance on the bought energy origin. Every kWh exchanged through the network can be mapped to the sensors that validated the generation.

Buyers can keep their anonymity while performing energy transactions to such an extent that network participants cannot infer the buyers' energy consumption patterns. Even though our model does not specify if the energy bought is consumed instantly, it might be, depending on the deployment context. In such a scenario anonymizing the buyer becomes essential.

In our model, energy sold has to be generated in the **past** to simplify and avoid energy delivery verification complexity. Thus, sellers do not correspond to a Balance Responsible Party (BRP), and they are not obliged to generate during a specific time window. This lack of responsibility might difficult for the utility companies to work to solve power imbalances. However, the vast amount of sensor data can serve as a counterbalance and, from another perspective, help to predict power imbalances.

Sell bids and buy bids can be partially matched, always generating energy transactions registering the energy quantity and settlement price. With that, buyers prove their ownership of the bought energy and request discounts on their energy bills. Sellers contact the payment company to receive the funds related to their transactions.

4.5 FURTHER MODEL DETAIL

Blockchains are heterogeneous and suit problems distinctly. Therefore, models' specificities depend on the selected technology. (CALDARELLI, 2020) considers proposals that explicitly define the blockchain technology to be more grounded and realistic. Furthermore, they argue that building a hypotheses without defining the blockchain technology may hardly provide a concrete contribution.

For example, our model was designed based on the Hyperledger Fabric blockchain, which has an organization oriented architecture. Thus, the same model would not fit the Ethereum straightforwardly. For that reason, some further details of our model are described in Chapter 5, where we elaborate on the model implementation.

5 PROPOSAL DEVELOPMENT

In this chapter, we will explain the development of our proposal, which we separated into four main sections. Section 5.1 presents the necessary steps and configurations to locally deploy a basic Hyperledger Fabric production network with a chaincode installed. Section 5.2 focuses on our chaincode design by explaining the main data structs, some important design principles, and the reason for modifying Hyperledger Fabric.

Section 5.3 explains how we utilized the adapted versions of fabric-sdk-java and fabric-gateway-java to implement applications for buyers, sellers, sensors, utility, and payment companies. Our experiments were performed fully inside the AWS cloud infrastructure, and Section 5.4 describes how we performed such deployment after adapting our local deploy scripts.

5.1 NETWORK LOCAL DEPLOYMENT

To develop and test our proposal, we had to deploy a Hyperledger Fabric network with organizations, peers, orderers, and applications. The Hyperledger Fabric network can be constructed in different ways (TEAM, F. D., 2020a). This section describes the general steps in network creation to provide enough understanding of our network development.

Hyperledger Fabric provides two alternatives for network deployment and testing: the **testnet** and **production networks**. The testnet is designed to run locally in a pre-defined network structure, with a couple of peers and a single orderer. It provides a simple and easy deployable environment so application designers can execute tests without deploying a production network, which is more complex.

The production network is the one used network in real environments. It allows the creation of as many peers, orderers, admins, and clients as defined. **All of our tests** are performed in a production network so that the results are closer to real applications.

5.1.1 Hyperledger Fabric general creation steps

When we mention the term **client**, we refer to an *application* outside the network, as presented in Chapter 2. The organization **member** represents the generalization of organization admin, client, orderer, or peer. The general steps for network creation are:

1. (*Optional*) Create one root Certificate Authority (CA) for each organization using the *fabric-ca-server* tool.

2. (*Optional*) Create one Transport Layer Security (TLS) CA for all organizations using the *fabric-ca-server* tool.
3. Generate two certificates for each organization member. One certificate only to handle for TLS communication purposes and the other as identification of organization Membership Service Provider (MSP).
4. Set the desired initial configuration for the network in a configuration file *configtx*.
5. Generate the *System Channel* genesis block with information about the organizations in the network, the certificates of all CAs involved, and the certificates of organization administrators.
6. Initialize the orderers with the genesis block created in step 5.
7. Initialize peers.
8. Organizations' admins create channels for the consortiums defined in the configuration file *configtx*.
9. Organizations' admins command their peers to join channels.
10. Organizations' admins install the desired chaincodes on peers.
11. Channel admins approve the chaincodes for the channel by collecting at least *N* signatures, defined in the network configuration file.
12. After enough approvals are collected, the chaincode is committed by a single channel admin.
13. (*Optional*) A channel admin calls the *init* (initialization) function on the chaincodes.

Steps 1 and 2 are optional as they do not necessarily need to be created because existing CAs could be used. Not all chaincodes require an initialization function. That is why step 13 is optional.

5.1.2 Environment with docker images

In a production network context, Fabric CAs, peers, and orderers are docker containers with images available to be downloaded in *Docker Hub* (INC., 2020). A typical deployment procedure consists of defining all CAs, peers, and orderers in a docker-compose file, where the image version and environment variables are defined.

Since we performed modifications on Hyperledger Fabric, the docker images had to be recompiled locally. Thus, the docker images mentioned in this chapter were not fetched from the docker hub, and **nothing** in our implementations can be replicated with the default Hyperledger Fabric docker images.

Code 5.1 shows an example of how to set the orderer configurations in a *docker-compose* file. After properly setting the *docker-compose.yml*, calling the command in Code 5.2 starts an orderer. Every **chaincode** is also a docker container, but they are started by the peers with chaincodes installed. These containers can be deployed in the same physical machine or multiple machines across different networks. Our initial tests were performed by deploying all containers in the same physical machine.

Code 5.1 – Example docker-compose.yml

```
orderer:
    container_name: orderer1-org1
    image: hyperledger/fabric-orderer:2.3.0
    environment:
        - ENV_VARIABLE_1=one
        - ENV_VARIABLE_2=two
    volumes:
        - a/local/machine/path:a/orderer1/virtual/environment/path
    networks:
        - fabric-network
    ports:
    - a-local-machine-port-binded-to:a-orderer1-virtual-environment-port
```

Code 5.2 – Starting an orderer

```
$ docker-compose -f docker-compose.yml up -d orderer
```

5.1.3 Network configuration files

This section will explain the purpose of each configuration file and how they change network behavior. All configurations are formatted in file format *YAML Ain't Markup Language (YAML)* (BEN-KIKI et al., 2020), which has useful features for configuration files, like referencing previous definitions and avoiding the need to repeat entire equal definitions. The main configuration files are:

- *fabric-ca-server-config.yaml* - configurations related to the Fabric CA server.
- *fabric-ca-client-config.yaml* - configurations related to the commands *fabric-ca-client enroll* and *fabric-ca-client register*.
- *configtx.yaml* - main configuration file. Most of the network topology is defined in this file.
- *core.yaml* - configuration file related to the peers.
- *orderer.yaml* - configuration file related to the orderers.

5.1.3.1 fabric-ca-server-config.yaml

In this file, some of the following parameters and characteristics about the Fabric CA server are set:

- Server port;
- Server in debug mode;
- TLS enabled for communication when members are enrolling;
- Certificates and keys for the CA (if they already exist);
- The maximum number that a member can enroll - ask for certificate signature;
- Pre-registered identities (usually for CA admins);
- Database parameters;
- Ldap parameters (if enabled);
- Allowed affiliations for registering;
- Signing parameters, expiry times, signing algorithm, etc;
- Idemix parameters;
- Crypto library;
- Intermediate CAs; and,
- Operation and metrics server endpoints parameters.

5.1.3.2 fabric-ca-client-config.yaml

The command *fabric-ca-client enroll* will look for this file to perform the enrollment. In this file, parameters related to the organization member enrollment are set:

- CA url;
- TLS files for secure communication;
- CSR (certificate signing request) parameters: common name, signing algorithm, country, state, location, organization unit, and others;
- Configurations related to the registration process;
- Enrollment type; and,
- Crypto library configuration.

5.1.3.3 configtx.yaml

This is the main configuration file, as it contains the organizations' information, the default configurations for channel policies, the anchor peers, and orderers. The main parameters defined in this file are:

- Organizations definitions
  - Organization name
  - Organization membership ID
  - Organization MSP type (idemix or x509)
  - Organization policies
  - Organization orderers address
  - Organization anchor peers
- Default access control policies
- Network ordering configuration
  - Ordering tool: kafka or raft
  - Ordering policies
- Default channel configurations
- Profiles
  - System channel definitions
  - Consortiums definitions
  - Specific channels definitions

5.1.3.4 core.yaml

Each peer has its *core.yaml* file from where its configuration is fetched. The main parameters defined in this file are:

- Peer id;
- Listen address and port;
- Gossip protocol configuration - for communication among peers;
- Private data configurations;

- Chaincode configurations;
- Database and ledger configurations; and,
- Operation and metrics server endpoints parameters.

5.1.3.5 orderer.yaml

Similar to the peer, each orderer has its *orderer.yaml* file to fetch the configurations. The main orderer parameters are:

- Listen address and port;
- TLS certificates and keys;
- Bootstrap method;
- Crypto library;
- Kafka settings; and,
- Operation, metrics, and administrative server endpoints parameters.

5.1.3.6 Overriding configuration files

Hyperledger Fabric peers, orderers, and CAs are developed in *Golang*, and all configuration files are parsed using the **viper** library (FRANCIA, 2020). This library loads configuration from YAML files, environment variables, and command flags when calling a command. As an example, consider the orderer configuration field *ListenAddress* in *orderer.yaml* presented in Code 5.3.

Code 5.3 – Piece of orderer.yaml

```yaml
#key1
General:
    # Listen address: The IP on which to bind to listen.
    #key2-inside-key1
    ListenAddress: 127.0.0.1
```

After parsing the *orderer.yaml* file, viper tries to read environment variables based on the *orderer.yaml* key names. The expected environment variable names' follow the general pattern: **filename_key1_key2-inside-key1**. The *ListenAddress* could be overridden by setting the environment variable *ORDERER_GENERAL_LISTENADDRESS* in the orderer docker-compose settings, as shown in Code 5.4.

Code 5.4 – Overriding configuration files

```yaml
orderer:
    container_name: orderer1-org1
    image: hyperledger/fabric-orderer:2.3.0
```

```
    environment:
        - ORDERER_GENERAL_LISTENADDRESS=180.180.180.180
    ...
```

Observation: if a configuration field is formed by a list of attributes, it cannot be overridden by environment variables. The viper priority order for parsing configuration values is:

1. Command flags
    - Ex: fabric-ca-client enroll ... **–enrollment.profile idemix**
2. Environment variables
3. Configuration file values

5.1.4 Automated network creation script

We developed a *bash* script and some python scripts for creating different networks easily. With these tools, we could deploy a network with as many organizations as necessary. Each organization can have as many admins, clients, orderers, and peers as wished. The script code is available on Github (WESTPHALL, 2021). In this section, the created scripts are explained.

The first configuration file to be set is presented in Code 5.5. In this file, we define a list of organizations in the network, describing their names, the admins, clients, peers, and orderers quantity, followed by the membership service provider type: idemix or x509 certificates.

Code 5.5 – Our initial configuration file *CONFIG-ME-FIRST.yaml*

```
organizations:
    - name: ufsc
      admin-quantity: 1
      client-quantity: 0
      peer-quantity: 1
      orderer-quantity: 1
      buyer-quantity: 0
      seller-quantity: 2
      sensor-quantity: 2
      msptype: x509

    - name: parma
      admin-quantity: 1
      client-quantity: 0
      peer-quantity: 1
      orderer-quantity: 1
      buyer-quantity: 0
      seller-quantity: 2
      sensor-quantity: 2
      msptype: x509

    - name: idemixorg
```

```
      admin-quantity: 1
      client-quantity: 1
      peer-quantity: 0
      orderer-quantity: 0
      buyer-quantity: 1
      seller-quantity: 0
      sensor-quantity: 0
      msptype: idemix
```

Then, we call a python script named *partialConfigtxGenerator.py* to parse the file presented in Code 5.5 and generate the *configtx.yaml* file with the organizations' full definitions. The python script also adds all orderers to the raft configurations section in *configtx.yaml*. This leaves only the profiles section or policy changes for manual configuration. Every other field is set according to the organizations described in the file presented in Code 5.5.

The next step is to create a single TLS CA for all organizations and one root CA for each organization's MSP. For this, we use the *docker-compose* command to turn on the CA services defined in our *docker-compose.yml* file, with the names *ca-tls* and *rca*. Even though only one service named *rca* is defined in the *docker-compose.yml* file, Code 5.6 shows how we create multiple root CAs from a single definition.

The *docker-compose* command reads environment variables and substitutes them with their value if any variable is referenced in the *docker-compose.yml*. With this tool, every time we create a root CA for an organization, we simply export the *ORG_NAME* environment variable with the organization name.

Code 5.6 – Root CA docker-compose.yml

```
rca:
    container_name: rca-${ORG_NAME}
    image: hyperledger/fabric-ca:latest
    command: bin/bash -c ’fabric-ca-server start -d -b rca-${ORG_NAME}-admin:rca-${ORG_NAME}-adminpw --port 7053’
    environment:
        - FABRIC_CA_SERVER_HOME=/tmp/hyperledger/fabric-ca/crypto
        - FABRIC_CA_SERVER_TLS_ENABLED=true
        - FABRIC_CA_SERVER_CSR_CN=rca-${ORG_NAME}
        - FABRIC_CA_SERVER_CSR_HOSTS=0.0.0.0
        - FABRIC_CA_SERVER_DEBUG=true
    volumes:
        - ${BASE_DIR}/hyperledger/${ORG_NAME}/ca:/tmp/hyperledger/fabric-ca
    networks:
        - fabric-network
    ports:
        - ${BINDABLE_PORT}:7053
```

There is still one problem: docker does not allow multiple services with the same name in a project. Therefore, docker would only allow the creation of a single service called *rca*, and it is forbidden to have environment variables references in services names. To get around that, we change the service name before creating a new root CA using *perl*. Code 5.7 explicitly shows how we perform this change.

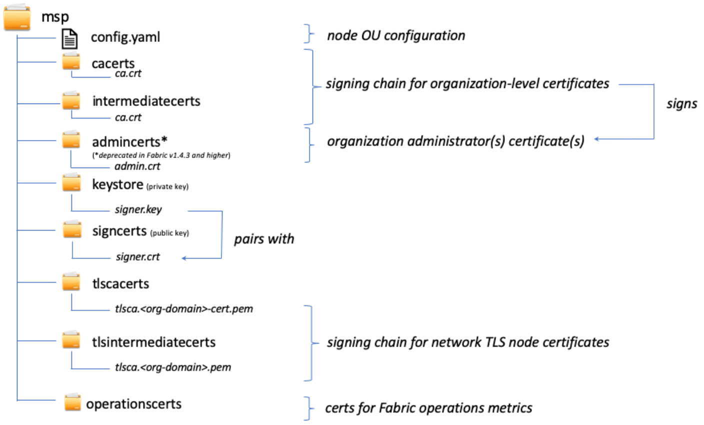
Code 5.7 – Changing service name and starting the service

```
perl -pi -e ’s/rca:/rca-’$ORG_NAME’:/g’ docker-compose.yml
docker-compose -f docker-compose.yml up -d rca-$ORG_NAME
```

Observation: changing the service name will create *orphan* services, and it will trigger warnings from docker. This is not the best practice for creating multiple services with the same docker image, but it is quick and easy. The proper way to do that would be to generate a *docker-compose.yml* with all services definitions. Generating a complete *docker-compose.yml* could be another action performed by the python script *partialConfigtxGenerator.py*. We apply the same principle to create multiple **peers** and **orderers**.

After the CAs creation, we register and enroll every admin, client, peer, orderer, buyer, seller, and sensor, calling the *fabric-ca-client* command. The generated MSP credentials are stored in the host machine folders mounted to the virtual docker containers, as shown in lines 11 and 12 of Code 5.6. The volume mounting allows the docker containers to use the generated MSP credentials. Figure 12 shows the structure of an MSP folder on every orderer, peer, or any other enrolled entity.

Figure 12 – MSP folder structure

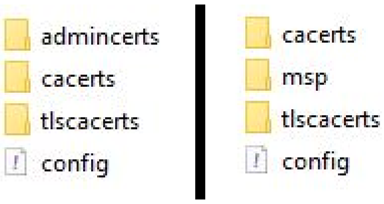


Source: (TEAM, F. D., 2020a)

Following all MSP credentials creation, the next step is to make an *MSP folder* for each organization. The organizational *MSP folder* must have three folders and one *config.yaml* file. One folder with organization admin certificates, one folder containing the root CA certificate, and another folder with the TLS CA certificate. The file *config.yaml* maps the *Organizational Unit (OU)* field in the MSP certificates to admin, client, orderer, or peer roles. In our network case, all sellers, buyers, and sensors are registered as clients.

If the organization uses idemix, the admin certificates folder is substituted by a folder named *msp* with the idemix issuer public key and revocation public key. Figure 13 shows the organization MSP folder structure in two different cases. Organizations with x509 MSP have the folder structure presented in the left part of Figure 13. The folders on the right are required for an idemix MSP.

Figure 13 – Organization MSP folder structure - x509 in the left and idemix in the right



Designed by the author

Before the genesis block is generated, our script requires that the *system channel*, the consortiums, and the application channels are manually declared in the *configtx.yaml*. Code 5.8 present a definition for the *system channel* called “SampleMultiMSPRaft” and a definition for an application channel called “SampleMultiMSPRaftAppChannel.”

It is worth noticing that orderer configurations are only declared in the *system channel* profile, with the organizations responsible for the network ordering process. Also, the application channel belongs to the consortium “SampleConsortium,” which corresponds to the consortium declared in the *system channel* profile.

Code 5.8 – System channel and application channel definitions

```
Profiles:
    # SampleMultiMSPRaft is a profile to the syschannel.
    # Remeber to add the organizations and the consortiums
    SampleMultiMSPRaft:
        <<: *ChannelDefaults
        Orderer:
            <<: *OrdererDefaults
            OrdererType: etcdraft
            Organizations:
                - *UFSC
                - *PARMA
        Consortiums:
            SampleConsortium:
                Organizations:
                    - *UFSC
                    - *PARMA
                    - *IDEMIXORG

    # SampleMultiMSPRaftAppChannel is a profile to application channels.
    # Remeber to add the organizations and the consortium name
```

```
    SampleMultiMSPRaftAppChannel:
        <<: *ChannelDefaults
        Consortium: SampleConsortium
        Application:
            <<: *ApplicationDefaults
            Organizations:
                - <<: *UFSC
                - <<: *PARMA
                - <<: *IDEMIXORG
```

For the genesis block generation, the command *configtxgen* is called as shown in Code 5.9 and outputs the genesis block from reading the previously generated and edited *configtx.yaml* as input. Then, the genesis block is copied to every orderer MSP directory.

Code 5.9 – Generating the genesis block

```
configtxgen -configPath $BASE_DIR/generated-config -profile
    SampleMultiMSPRaft -outputBlock
    ${BASE_DIR}/hyperledger/tempgenesis.block -channelID syschannel
```

With *genesis block* present in every orderer file system, the orderers are started. The path to the genesis block is set in an environment variable, as shown in Code 5.10. As they start, the orderers will fetch information about other orderers from the *genesis block* and communicate with them via the *system channel*.

Code 5.10 – Setting the path to the genesis block

```
orderer:
    container_name: orderer${ORDERER_NUMBER}-${ORG_NAME}
    image: hyperledger/fabric-orderer:2.3.0
    environment:
            ...
            - ORDERER_GENERAL_BOOTSTRAPFILE=/tmp/hyperledger/${ORG_NAME}/orderer${ORDERER_NUMBER}/
                 genesis.block
            ...
```

The next step is peer initialization. As soon as each peer is started, they look for other peers' endpoints in the same organization to start communicating. The environment variable *CORE_PEER_GOSSIP_BOOTSTRAP* defines the list of possible peers to start communicating with on bootstrap. We make every peer communicate to "peer1" of their organization on initialization, as demonstrated in Code 5.11.

Code 5.11 – Peer bootstrap configuration

```
peer:
    container_name: peer${PEER_NUMBER}-${ORG_NAME}
    image: hyperledger/fabric-peer:2.3.0
    environment:
        ...
        - CORE_PEER_GOSSIP_BOOTSTRAP=peer1-${ORG_NAME}:7051
        ...
```

To configure the peers, we use the Hyperledger Fabric *Command-Line Interface (CLI)* container to interact with peers using administrators' credentials. We instantiate a single *CLI* container to manage all organizations' entities. All commands executed via *CLI* could be called from any machine. However, it is easier to utilize the *CLI* since it is inside the virtual docker network, and commands can reference names of the virtual network Domain Name System (DNS).

Code 5.12 presents an example of a command executed with a *CLI*. Notice how the *CLI* allows us to reference the name "peer1-ufsc", which is only available inside the virtual docker network. If this same command were called outside the virtual docker network, it would not work since the name "peer1-ufsc" would not be resolved.

Code 5.12 – Cli command example

```
docker exec -e CORE_PEER_LOCALMSPID=UFSC -e
   CORE_PEER_ADDRESS=peer1-ufsc:7051 -e
   CORE_PEER_MSPCONFIGPATH=/tmp/hyperledger/ufsc/admin1 /msp -e
   CORE_PEER_TLS_ENABLED=true -e
   CORE_PEER_TLS_ROOTCERT_FILE=/tmp/hyperledger/ufsc/
   admin1/tls-msp/tlscacerts/tls-0-0-0-0-7052.pem cli-ufsc peer lifecycle
   chaincode install energy.tar.gz
```

We create the genesis block for the application channel definition "SampleMultiMSPRaftAppChannel" presented in Code 5.8 and use the *CLI* container to make all peers join the channel. Code 5.13 demonstrates the command that creates the channel genesis block for the consortium "SampleConsortium."

Code 5.13 – Cli command example

```
configtxgen -configPath $BASE_DIR/generated-config -profile
   SampleMultiMSPRaftAppChannel -outputCreateChannelTx
   ${BASE_DIR}/hyperledger/$firstOrgInChannelLower/ admin1/$channelID.tx
   -channelID $channelID --asOrg $orgName
```

The application channel requires a definition for organizations' **anchor** peers to allow peers from different organizations to gossip. We set the organization's first peer as an anchor by altering the application channel configuration. A single anchor peer per organization is enough to enable all organization's peers to be discovered from outside of it. This definition is also required to enable that applications perform *Service Discovery* on the channel to know the endorsing peers for a transaction, explained in Section 5.3.5.

After all the peers are in the channel, we install our developed smart contract called "**energy**," presented in section 5.2. The contract installation goes through the steps of packing, installing, approving, and committing. Code 5.14 presents the commands related to the steps to install the contract. We omit the command flags and environment variables settings, but they are available in our Github (WESTPHALL, 2021).

Code 5.14 – Cli command example

```
cli-$orgNameLower peer lifecycle chaincode package #each organization admin package each chaincode once
cli-$orgNameLower peer lifecycle chaincode install #each organization admin install the chaincode in every peer of their organization
cli-$orgNameLower peer lifecycle chaincode approveformyorg #each organization admin approves each chaincode definition once
cli-$committerOrgLower peer lifecycle chaincode commit #only one admin on the channel needs to commit the chaincode, after most organizations approved the chaincode definition
```

Finally, some chaincodes might require the admin to call the initialization function before they are available for normal use. Depending on the chaincode, calling the initialization function is the last required step to have it all ready for use. The "**energy**" smart contract, as an example, requires initialization.

The applications that transact with the "**energy**" smart contract are executed in an ubuntu container, called *cli-applications*, within the same network as peers and orderers. Deploying applications in this container avoids problems with endpoints' names (*DNS*) and ports found by the discovery service, which is a tool that can increase transaction throughput.

5.1.4.1 Network created

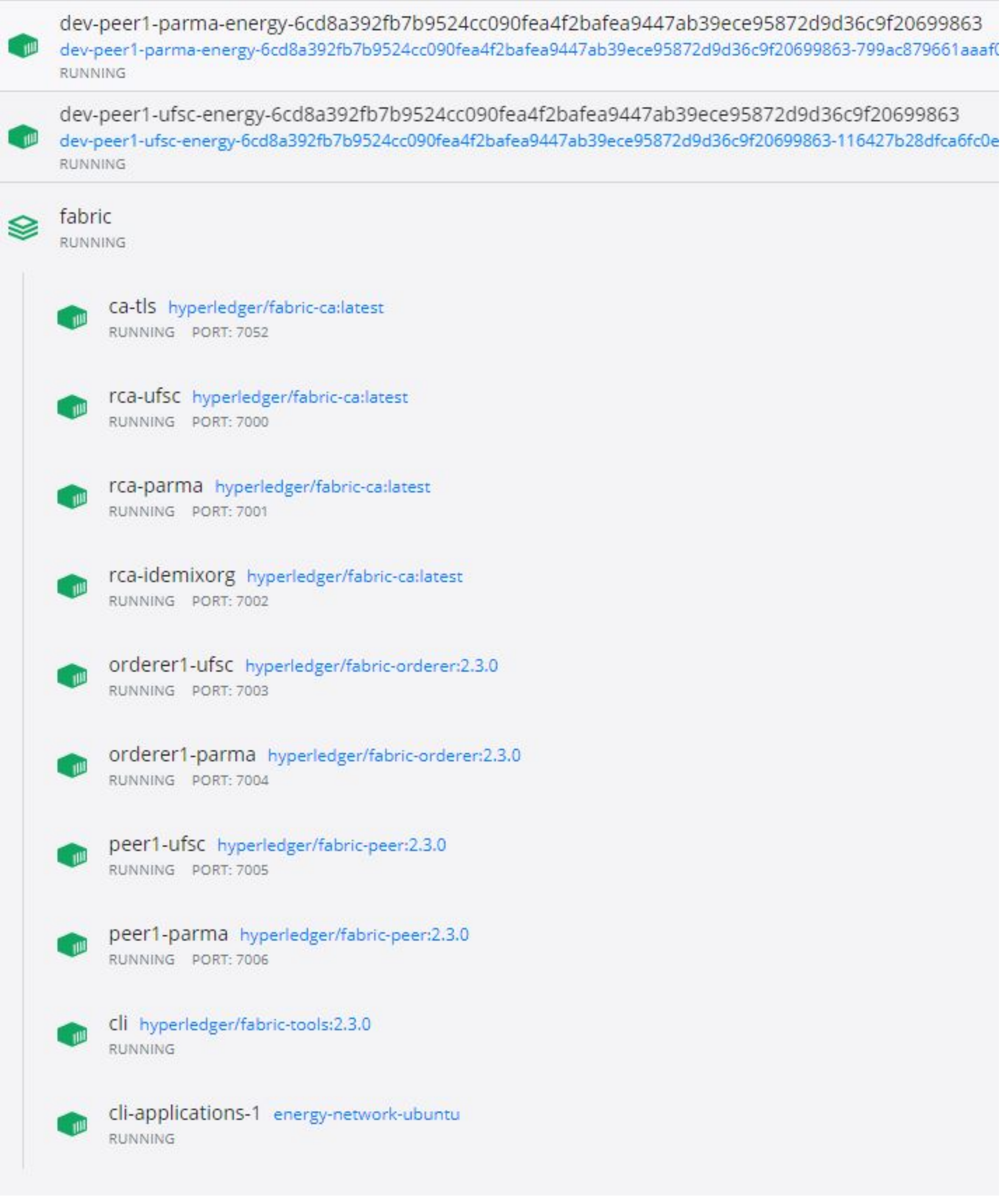
Figure 14 shows the resulting docker private network after executing our local deploy script. The first two containers are the chaincodes for the two peers. The *ca-tls* generates x509 certificates to support TLS communication, while each organizations' *rcas* provide the membership x509 certificates.

The *parma* and *ufsc* organizations have each one peer and one orderer. The *CLI* container helps the deployment process for chaincode installation, and *cli-application-1* executes applications that interact with the blockchain.

5.2 CHAINCODE DEPLOYMENT

In this section, we present the developed chaincode that executes in every peer of the network. It contains functions to execute the actions displayed by the sequence diagram in Figures 10 and 11.

Figure 14 – Resulting network in docker



dev-peer1-parma-energy-6cd8a392fb7b9524cc090fea4f2bafea9447ab39ece95872d9d36c9f20699863
dev-peer1-parma-energy-6cd8a392fb7b9524cc090fea4f2bafea9447ab39ece95872d9d36c9f20699863-799ac879661aaaf
RUNNING

dev-peer1-ufsc-energy-6cd8a392fb7b9524cc090fea4f2bafea9447ab39ece95872d9d36c9f20699863
dev-peer1-ufsc-energy-6cd8a392fb7b9524cc090fea4f2bafea9447ab39ece95872d9d36c9f20699863-116427b28dfca6fc0e
RUNNING

fabric
RUNNING

ca-tls hyperledger/fabric-ca:latest
RUNNING PORT: 7052

rca-ufsc hyperledger/fabric-ca:latest
RUNNING PORT: 7000

rca-parma hyperledger/fabric-ca:latest
RUNNING PORT: 7001

rca-idemixorg hyperledger/fabric-ca:latest
RUNNING PORT: 7002

orderer1-ufsc hyperledger/fabric-orderer:2.3.0
RUNNING PORT: 7003

orderer1-parma hyperledger/fabric-orderer:2.3.0
RUNNING PORT: 7004

peer1-ufsc hyperledger/fabric-peer:2.3.0
RUNNING PORT: 7005

peer1-parma hyperledger/fabric-peer:2.3.0
RUNNING PORT: 7006

cli hyperledger/fabric-tools:2.3.0
RUNNING

cli-applications-1 energy-network-ubuntu
RUNNING

Designed by author

Go was the chosen language to implement our model since this is a general recommendation for developers because it matches the Hyperledger Fabric implementation language. Usually, the new chaincode features become available first in *Go*, besides generating smaller docker images. Also, the authors of (FOSCHINI et al., 2020) analyzed each chaincode language - *Go*, Java and Javascript - and identified a better performance on contracts written in *Go*.

5.2.1 World State keys and values

Hyperledger Fabric transactions interact with the ledger and World State, usually reading or writing to a State. Fabric's State database stores *key-value* pairs representing different states, with the *key* as a string and the *value* stored as bytes. In our chaincode, we store *Go* structs after serializing them to *Protocol Buffers (protobuf)*.

A key can be **simple** with a single name identification with only *utf-8* characters. Another possibility is the **composite** one, when the goal is to form the key with an *object type* and many attributes. The *object type* and the attributes are placed in sequence and separated by the minimum Unicode character (\u0000), aiming to avoid collisions with **simple** keys.

Hyperledger Fabric enables fetching State *values* by providing the full *key* or a *key prefix*. Code 5.15 exhibits the *GetState* functions and their appropriate context. Deletions and insertions are also possible by providing the full State *key*.

Code 5.15 – *ActiveSensor* struct

```
//get state by its full key
GetState(key string) ([]byte, error)

//get SIMPLE key states within the range [startKey, endKey[ - alphabetic order
GetStateByRange(startKey, endKey string) (StateQueryIteratorInterface, error)

//get COMPOSITE key states formed by: objectType|U+0000|attr1|u+0000|attr2...
GetStateByPartialCompositeKey(objectType string, keys []string) (StateQueryIteratorInterface, error)
```

5.2.2 Choosing the most appropriate database

Hyperledger Fabric supports two databases to store the ledger and World State. Go LevelDB has a simple key-value architecture and only supports key, key range, and composite key queries (TEAM, F. D., 2020b). On the other hand, CouchDB supports more diverse queries, as long as the data is modeled in JSON format. After some tests presented and discussed in Section 7.1, we opted for **LevelDB** due to significantly better performance results.

5.2.3 Identifying chaincode function callers

Fabric-chaincode-go provides a *Client Identity Chaincode Library* to read certificate attributes and ensure attribute-based access control (TEAM, F., 2021). Code

5.16 exemplifies how to restrict access to a chaincode function by testing the value of an attribute on the caller x509 certificate.

Code 5.16 – Ensuring that only callers with the attribute "energy.seller" execute a function execution

```
...
//only sellers can execute this function
err := cid.AssertAttributeValue(stub, "energy.seller", "true")
if err != nil {
    return shim.Error(err.Error())
}
...
```

Every network participant can be uniquely identified by the concatenation of their *MspID* and *CertificateID*. The *MspID* is equivalent to the participant's organization name, and the *CertificateID*, unique within the same organization, derives from the x509 certificate Distinguished Names (DN) as displayed in Code 5.17. We resort to this identification technique in many of our chaincode structs, described in Section 5.2.4.

Code 5.17 – How IDs are generated from the x509 certificate (from https://github.com/hyperledger/fabric-chaincode-go/)

```
func (c *ClientID) GetID() (string, error) {
    ...
    // The leading "x509::" distinguishes this as an X509 certificate, and
    // the subject and issuer DNs uniquely identify the X509 certificate.
    // The resulting ID will remain the same if the certificate is renewed.
    id := fmt.Sprintf("x509::%s::%s", getDN(&c.cert.Subject),
       getDN(&c.cert.Issuer))
    return base64.StdEncoding.EncodeToString([]byte(id)), nil
}
```

5.2.4 Main data structs

To provide a general view of our chaincode, in the following subsections, we explain the main structs defined in it. The struct name is always used as a prefix to the World State key, together with other fields. Almost all fields containing "*ID*" in the name refer to the identification ways presented in Section 5.2.3.

5.2.4.1 ActiveSensor struct

The *ActiveSensor* struct, displayed in Code 5.18, gathers the sensor's *MspID*, *SensorID*, and the indication if the sensor is active or not. In our implementation, we assume that the sensors are stationary. Therefore, they have the fixed coordinates *X*, *Y*, *Z*. The *Radius* represents the maximum distance from the coordinates with similar environmental characteristics as measured by the sensor. The coordinates and the radius are fetched from the sensor's x509 certificate attributes.

This struct has two main purposes, with the first being to enable or disable the sensor by changing the *IsActive* value. The other purpose involves identifying the sensors near a seller and fetching these sensors' *SmartData* to validate an energy generation claim. The *ActiveSensor* World State *key* is in line 1 of Code 5.18.

Code 5.18 – *ActiveSensor* struct

```
 1  //key in the World State = stub.CreateCompositeKey("ActiveSensor",
       []string{MspID, SensorID}
 2  type ActiveSensor struct {
 3      MspID                    string   ‘protobuf:"..." json:"..."‘
 4      SensorID                 string   ‘protobuf:"..." json:"..."‘
 5      IsActive                 bool     ‘protobuf:"..." json:"..."‘
 6      X                        int32    ‘protobuf:"..." json:"..."‘
 7      Y                        int32    ‘protobuf:"..." json:"..."‘
 8      Z                        int32    ‘protobuf:"..." json:"..."‘
 9      Radius                   float64  ‘protobuf:"..." json:"..."‘
10  }
```

5.2.4.2 SmartData struct

Every time a sensor publishes an observed metric to the ledger, the chaincode stores the corresponding *SmartData*. Each *SmartData* field is explained in Section 2.4, but the *AssetID*, which is composed of the smart meter's *MspID* and the *SensorID* from the x509 certificate. This struct plays an essential role in the energy validation process when the chaincode fetches near sensors' *SmartData* to evaluate the energy generation claim trustworthiness.

The coordinates fields are not in this struct because they are already set in *ActiveSensor*. Considering that the coordinates are always fetched from the x509 certificate, duplicating the coordinates in the *SmartData* struct only increases memory usage unnecessarily. If our chaincode accepted data from moving sensors, this struct would require redesign.

Its World State key is presented by Code 5.19 in lines 1 and 2. Different from the *ActiveSensor* struct, the *SmartData* key is not a *composite key* but a simple one. This design choice enables efficient queries when requesting a set of *SmartData* from a specific sensor within a timestamp range. The Hyperledger chaincode function *shim.ChaincodeStubInterface.GetStateByRange(startKey, endKey string)* executes these queries efficiently, but it works only with simple keys. We discuss these *SmartData* queries' performance further in Section 7.1.1.1.

Code 5.19 – *SmartData* struct

```
//AssetID = SensorMspID + SensorID
//key in the World State = "SmartData" + AssetID + getMaxUint64CharsStrTimestamp(Timestamp)
type SmartData struct {
    AssetID                 string   ‘protobuf:"..." json:"..."‘
    Version                 int32    ‘protobuf:"..." json:"..."‘
    Unit                    uint32   ‘protobuf:"..." json:"..."‘
    Timestamp               uint64   ‘protobuf:"..." json:"..."‘
    Value                   float64  ‘protobuf:"..." json:"..."‘
    Error                   uint32   ‘protobuf:"..." json:"..."‘
    Confidence              uint32   ‘protobuf:"..." json:"..."‘
    Dev                     uint32   ‘protobuf:"..." json:"..."‘
}
```

Since the *GetStateByRange()* checks if a key is within the desired range based on the alphabetical order, the function *getMaxUint64CharsStrTimestamp*, shown in Code 5.20, forces every *timestamp* string representation to have the same length. For example, two *SmartData* from a sensor, one published in *timestamp 2* and the other in *timestamp 10*, without a correction, would have the keys, respectively, *AssetID|2* and *AssetID|10*.

In this context, if the chaincode tried to retrieve the *SmartDatas* with timestamp between *1* and *15*, the *SmartData* of key *AssetID|2* would not be fetched because, considering the alphabetic order, the string *AssetID|2* is greater than *AssetID|15*. To avoid this failure type, the function *getMaxUint64CharsStrTimestamp* generates an equivalent timestamp string representation as lengthy as the greatest *uint64* by adding zeros in the beginning to fill the difference.

Code 5.20 – *getMaxUint64CharsStrTimestamp* function

```
func getMaxUint64CharsStrTimestamp(timestamp uint64) string {
    timestampStr := strconv.FormatUint(timestamp, 10)
    for i := len(timestampStr); i < maxUint64Chars; i++ {
        timestampStr = "0" + timestampStr
    }
    return timestampStr
```

```
}
```

5.2.4.3 SellerInfo struct

When the chaincode receives a seller registration request, it stores their related information with the struct *SellerInfo*, with the seller's and their smart meter's certificate identification. This struct also contains the owned wind turbines and solar panels quantity so that the chaincode might have parameters to calculate the maximum possible energy generation amount.

After validating the energy generated, the chaincode increments the map *EnergyToSellByType*, which stores the seller's salable energy quantity, in kWh, per type - solar, wind, or other. The seller can only request energy validation for a certain time interval *[start time, end time[* if the *start time* is greater than the *LastGenerationTimestamp* value. Otherwise, the chaincode will deny the request.

When the seller desires to liquidate the validated energy, they publish a *SellBid*, described in Section 5.2.4.5, partly identified by the value of *LastBidID* added to one. Subsequently, the fields *LastBidID* and the *EnergyToSellByType* are updated in the *SellerInfo* struct.

Observation: *WindTurbinesNumber* and *SolarPanelsNumber* merely represent the needed information to achieve a reasonable maximum energy generation estimation. More information could be required in a real application, but this specificity is out of our scope.

Code 5.21 – *SellerInfo* struct

```
//key in the World State = stub.CreateCompositeKey("SellerInfo",
   []string{MspIDSeller, SellerID})
type SellerInfo struct {
    MspIDSeller             string             ‘protobuf:"..." json:"..."‘
    SellerID                string             ‘protobuf:"..." json:"..."‘
    MspIDSmartMeter         string             ‘protobuf:"..." json:"..."‘
    SmartMeterID            string             ‘protobuf:"..." json:"..."‘
    WindTurbinesNumber      uint64             ‘protobuf:"..." json:"..."‘
    SolarPanelsNumber       uint64             ‘protobuf:"..." json:"..."‘
    EnergyToSellByType      map[string]float64 ‘protobuf:"..." json:"..."‘
    LastGenerationTimestamp uint64             ‘protobuf:"..." json:"..."‘
    LastBidID               uint64             ‘protobuf:"..." json:"..."‘
}
```

5.2.4.4 MeterSeller struct

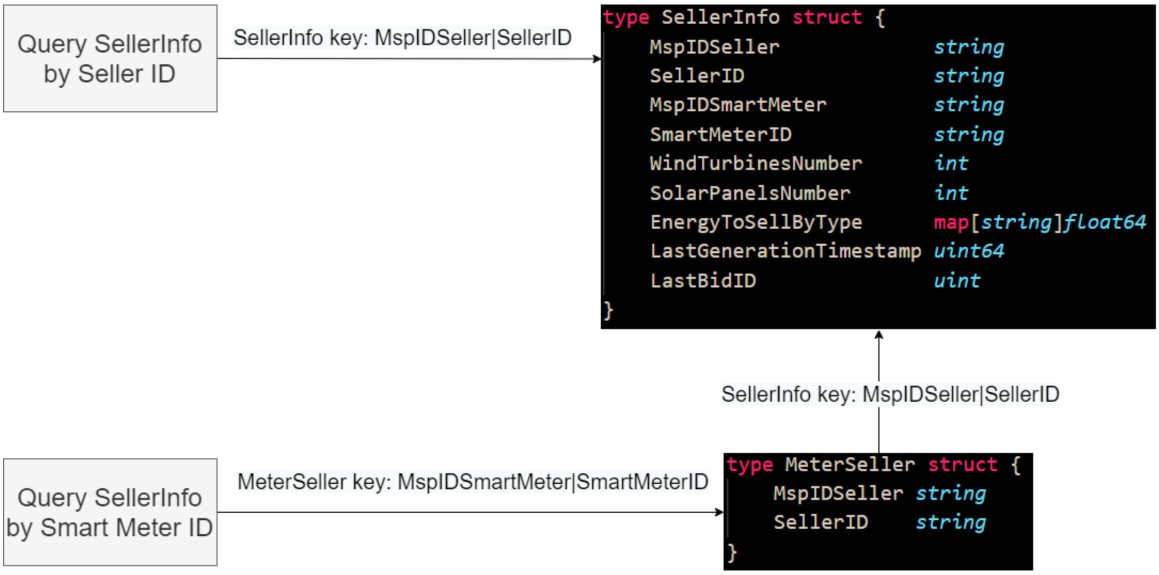
The *MeterSeller* struct, displayed in Code 5.22, serves as a pointer for the chaincode to find a *SellerInfo* by the smart meter credentials. Figure 15 elucidates two possible queries to retrieve a *SellerInfo*, one by directly informing the key and the other by using the *MeterSeller* struct as a pointer.

We created this auxiliary struct after our preliminary metrics findings described in Section 7.1 when we concluded that LevelDB performs outstandingly faster than CouchDB. However, LevelDB has the downside of only supporting full or partial key state queries, whereas CouchDB allows more specific JSON ones. Opting for CouchDB would imply in the *MeterSeller* struct unnecessity, at a performance cost.

Code 5.22 – *MeterSeller* struct

```
//key in the World State = stub.CreateCompositeKey(objectType,
   []string{mspIDSmartMeter, smartMeterID})
type MeterSeller struct {
    MspIDSeller          string   ‘protobuf:"..." json:"..."‘
    SellerID             string   ‘protobuf:"..." json:"..."‘
}
```

Figure 15 – Possible ways to fetch a *SellerInfo* from World State



```
type SellerInfo struct {
    MspIDSeller              string
    SellerID                 string
    MspIDSmartMeter          string
    SmartMeterID             string
    WindTurbinesNumber       int
    SolarPanelsNumber        int
    EnergyToSellByType       map[string]float64
    LastGenerationTimestamp  uint64
    LastBidID                uint
}
```

```
type MeterSeller struct {
    MspIDSeller string
    SellerID    string
}
```

Designed by the author

5.2.4.5 SellBid struct

Once a seller's energy is validated, they can offer it to buyers by publishing a *SellBid* to the chaincode. This struct saves the seller's identification, the sequential sell bid number, the energy quantity in kWh to be sold, the price per kWh, and the energy type - wind, solar or other. The chaincode collects all *SellBids* in the World State to execute the auction from time to time.

After the auction, all satisfied *SellBids* are deleted from the World State to avoid complexities when retrieving *SellBids* for the following auctions. Still, the *EnergyTransaction* struct, explained in Section 5.2.4.7, stores the satisfied *SellBids* fields.

Code 5.23 – *SellBid* struct

```
//SellBid aprox. memory size = 10 + 177 + 4 + 8 + 8 + 10 = 217 bytes
//key in the World State = stub.CreateCompositeKey("SellBid",
   []string{MspIDSeller, SellerID, SellerBidNumber})
type SellBid struct {
    MspIDSeller         string   ‘protobuf:"..." json:"..."‘
    SellerID            string   ‘protobuf:"..." json:"..."‘
    SellerBidNumber     uint64   ‘protobuf:"..." json:"..."‘
    EnergyQuantityKWH   float64  ‘protobuf:"..." json:"..."‘
    PricePerKWH         float64  ‘protobuf:"..." json:"..."‘
    EnergyType          string   ‘protobuf:"..." json:"..."‘
}
```

5.2.4.6 BuyBid struct

Unlike the *SellBid*, the *BuyBid* does not contain any buyer information because it is published by a buyer with idemix credentials, ensuring pseudonymity. The fields *MspIDPaymentCompany*, which is the payment company organizational name, and the payment *Token* uniquely identify the *BuyBid*. To avoid possible attacks on different utility companies, the field *UtilityMspID* specifies the buyer's utility company. Otherwise, they could maliciously lend their credentials to a client of a different utility company, enabling two bill discounts for the same *BuyBid*.

EnergyQuantityKWH, *PricePerKWH*, and *EnergyType* have the same function as in the *SellBid*. Every *BuyBid* needs to be validated by the payment company before it can participate in an auction, confirming to the seller that they will get paid in case of matching a *BuyBid*. Code 5.24 lines 1 and 2 present the two possible keys for a *BuyBid* struct.

We also identify the validity in the key, *true* for validated and *false* otherwise. This design pattern allows efficient validated *BuyBids* fetching by the partial key "*BuyBid|U+0000|true*", increasing the auction speed. The satisfied *BuyBids* are also deleted from the World State after the auction.

Code 5.24 – *BuyBid* struct

```
//keys in the World State = stub.CreateCompositeKey("BuyBid",
   []string{"false", mspIDPaymentCompany, token})
// or = stub.CreateCompositeKey("BuyBid", []string{"true",
   mspIDPaymentCompany, token})
type BuyBid struct {
    MspIDPaymentCompany  string   ‘protobuf:"..." json:"..."‘
    Token                string   ‘protobuf:"..." json:"..."‘
    UtilityMspID         string   ‘protobuf:"..." json:"..."‘
    EnergyQuantityKWH    float64  ‘protobuf:"..." json:"..."‘
    PricePerKWH          float64  ‘protobuf:"..." json:"..."‘
    EnergyType           string   ‘protobuf:"..." json:"..."‘
}
```

5.2.4.7 EnergyTransaction struct

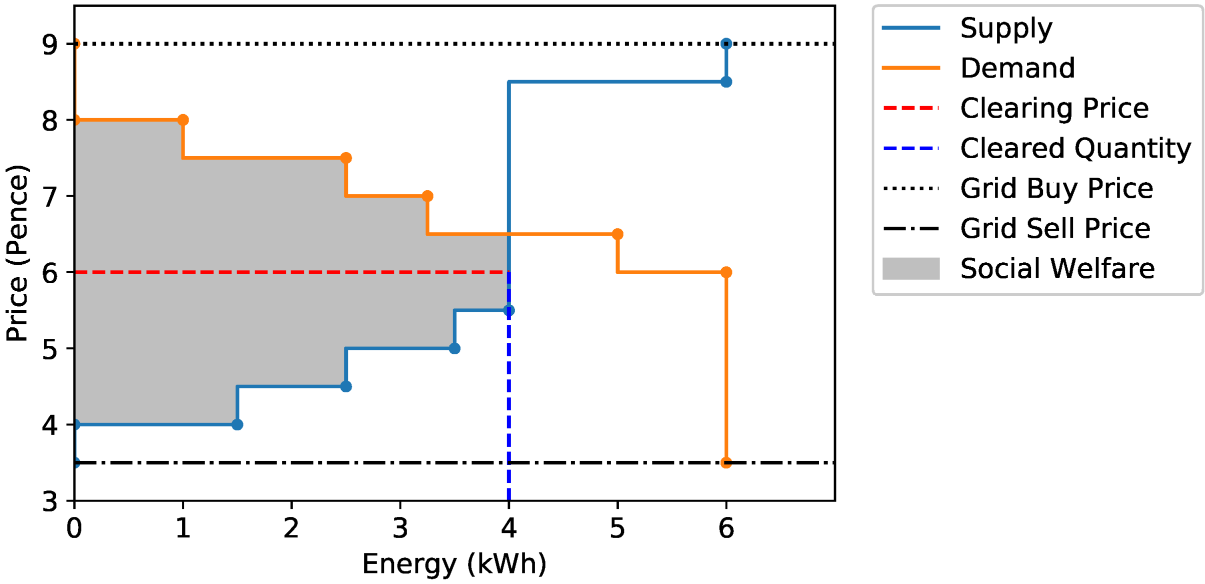
The *EnergyTransaction* struct, presented in Code 5.25, results from matching a *BuyBid* and a *SellBid*, created during the auction process. It joins the main fields of the two structs, enabling that the seller requests their payment and that the buyer asks for a bill discount. *EnergyQuantityKWH* and *PricePerKWH* probably differ from the bids because the auction might only partially satisfy a bid or need multiple *SellBids* to satisfy a single *BuyBid*. To uniquely identify an *EnergyTransaction*, we form the key as displayed in line 1 of Code 5.25.

Code 5.25 – *EnergyTransaction* struct

```
 1  //key in the World State = stub.CreateCompositeKey("EnergyTransaction",
       []string{MspIDPaymentCompany, Token, MspIDSeller, SellerID,
       SellerBidNumberStr})
 2  type EnergyTransaction struct {
 3      MspIDSeller          string   ‘protobuf:"..." json:"..."‘
 4      SellerID             string   ‘protobuf:"..." json:"..."‘
 5      SellerBidNumber      uint64   ‘protobuf:"..." json:"..."‘
 6      MspIDPaymentCompany  string   ‘protobuf:"..." json:"..."‘
 7      Token                string   ‘protobuf:"..." json:"..."‘
 8      BuyerUtilityMspID    string   ‘protobuf:"..." json:"..."‘
 9      EnergyQuantityKWH    float64  ‘protobuf:"..." json:"..."‘
10      PricePerKWH          float64  ‘protobuf:"..." json:"..."‘
11      EnergyType           string   ‘protobuf:"..." json:"..."‘
12  }
```

Figure 16 illustrates how our chaincode executes the auction process, plus presents the conditions when transacting energy in an alternative energy market is ideal. If the prices are better than with the main grid, people transact in the alternative market. The chaincode sorts the *BuyBids* in price-descending order and the *SellBids* in the ascending. The bids are sequentially matched while the *BuyBid* price exceeds the *SellBid* price. When this condition changes, the matching stops, and the *Clearing Price* is calculated from the average price of the last matched *BuyBid* and *SellBid*. Everyone receives or pays this specific price for the energy transacted, maximizing the participants' welfare.

Figure 16 – Double auction in an alternative energy market



Source: (ALABDULLATIF et al., 2020)

5.2.5 Energy validation

As soon as a seller invokes the *publishEnergyGeneration* chaincode function, the network tries to validate the claim against the *SmartData* published by sensors. We implemented two **representative** functions to check the validity of wind and solar energy generation claims.

The chaincode considers the seller's location and lists all the near sensors trusted by validators. After that, it fetches the sensors' published *SmartData* within the generation claim interval. As an example, if the seller declares that the energy was generated between 1:00 PM to 2:00 PM, only *SmartData* within this period will be retrieved from the World State.

After receiving a solar energy generation claim, the chaincode calls *getMaxPossibleGeneratedSolarEnergyInInterval*, displayed in Code A.1. This function loops through the *SmartData* list and selects only the ones with *Candela* (luminosity) unit. First, the function calculates each sensor *SmartData* average, **assuming they were published with constant frequency**. Then, it uses these averages to calculate the average of all sensors. Finally, the maximum possible energy generation is returned based on this last average and the sellers' solar panels quantity.

It is important to reinforce that this function **was not designed** to accurately calculate the maximum energy amount on a real deploy environment, which would require more expertise. However, the function applies a database load equivalent to a real application when fetching *SmartData*, satisfying our experiment needs.

5.2.6 Auction chaincode events

Chaincode functions can trigger events to applications after the transaction with the function call is published in a block. Regardless of the transaction being ruled valid or invalid, the event is sent to applications that subscribed to it. This tool avoids that applications constantly poll the chaincode to find out about state changes.

Our developed chaincode generates an event when an auction transaction is published to the channel block. Then, buyers and sellers can query the channel to verify if their bids were matched, resulting in an *EnergyTransaction*. We show eventing examples in Section 5.3.3.2.

5.2.7 Avoiding transaction invalidation due to changes in Read/Write key set (Phantom reads)

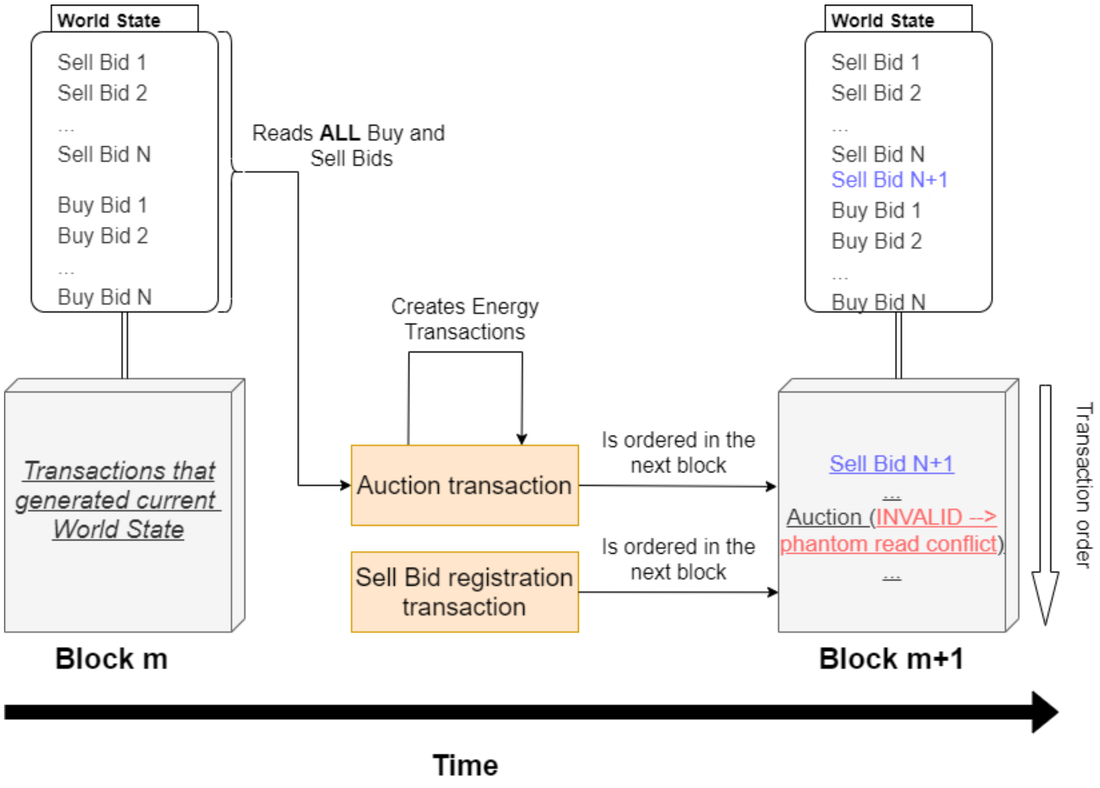
In the Hyperledger Fabric **execute-order-validate** transaction flow, delivering the transaction to the orderer guarantees that it will be published in a block. However, it does not imply anything on its validity. A modification on a read value by another transaction might cause invalidation, depending on how they are ordered. In our chaincode, this could happen if a *BuyBid* is validated or a new *SellBid* is published before the auction transaction is committed. Even though no bids were modified but only added, the auction can be invalidated due to a *phantom read*.

A *phantom read* happens when a transaction queries states using the function *GetStateByRange(startKey, endKey)* and the query result in the simulation phase is different than in the validation phase (TEAM, F. D., 2020c). Figure 17 illustrates the problem in our chaincode context. The auction reads all Buy and Sell bids present in the World State of **Block m** during the simulation phase. Then, the auction transaction is ordered in the **Block m+1**, just after a sell bid registration transaction that created *Sell Bid N+1*.

At validation time, the peers notice that the auction read all sell bids, but the *Sell Bid N+1* was not read. By default, peers will invalidate the auction transaction, assuming that missing the new *Sell Bid N+1* could cause an error. However, this does not lead to an error in our application, as the new sell bid would be processed in the next auction.

Initially, we thought adding a priority to the auction transaction - to force the orderer to place it as the first transaction of a block - would solve the problem. After modifying fabric's source code to let different transactions having distinct priorities, we realized it only solved phantom reads in **sequential** blocks. Figure 18 shows how our first solution fails when phantom read conflicts happen in a non-sequential block context.

Figure 17 – Phantom read conflict in sequential blocks



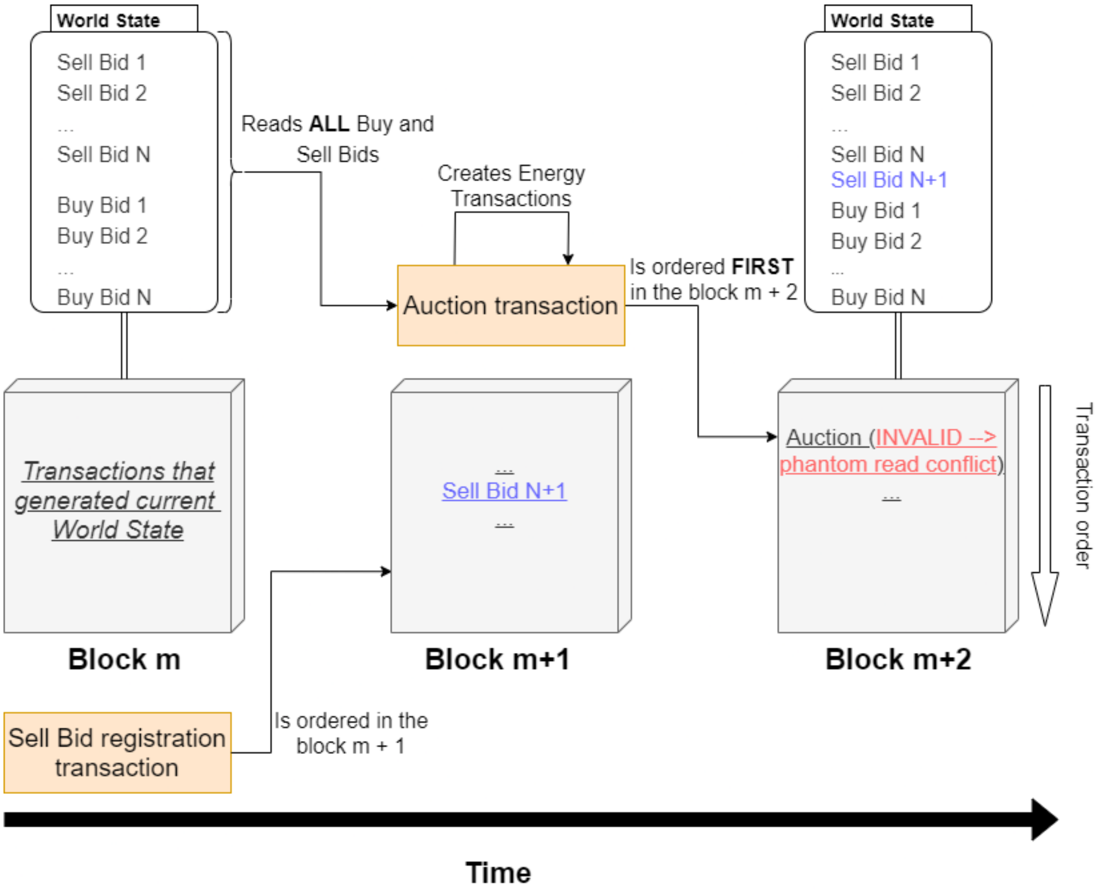
Designed by author

The auction simulation takes more time than a sell bid registry transaction, possibly causing the auction simulated with **Block m's** World State to be ordered in **Block m+2**. By principle, blockchains do not allow modifications on previous blocks, so, to permanently avoid phantom read conflicts, we enabled chaincode functions to define if this type of check should be executed at validation time.

We chose to keep the first solution modifications, regarding transaction priorities, in our patched Hyperledger Fabric because, although we did not need it, blockchain researchers discuss solutions related to transaction ordering methods to avoid invalidations, like the authors of (GOEL et al., 2018) and (XU et al., 2021). As presented in Code 5.26, our solution lets the more costly transaction functions *publishEnergyGeneration* and *auction* to bypass the *phantom read* check and not be invalidated by other recurring functions that alter or add *SellBids*, *BuyBids*, *ActiveSensors*, and *SmartData*.

With our modification, at validation time, the peers will check the transaction response. If it returned with the function *SuccessWithPriorityBypassPhantomReadCheck*, they will not perform the phantom read check and validate the transaction. The full modification patch is available in our Github (WESTPHALL, 2021).

Figure 18 – Phantom read conflict in non-sequential blocks



Designed by author

Code 5.26 – Enabling chaincode function return setting a priority, preventing the transaction invalidation due to the reads on lines 3, 10 and 12

```
1 func (chaincode *EnergyChaincode) publishEnergyGeneration(stub
     shim.ChaincodeStubInterface, t0 uint64, t1 uint64,
```

```
   energyByTypeGeneratedKWH map[string]float64) pb.Response {
    ...
    stub.GetStateByPartialCompositeKey("ActiveSensor", []string{}) //called indirectly by getActiveSensorsList()
    ...
    return shim.SuccessWithPriorityBypassPhantomReadCheck(
        []byte(successMessage), pb.Priority_MEDIUM)

}
func (chaincode *EnergyChaincode) auction(stub shim.ChaincodeStubInterface) pb.Response {
    stub.GetStateByPartialCompositeKey("SellBid", []string{})
    ...
     stub.GetStateByPartialCompositeKey("BuyBid", []string{"true"})
    ...
    return shim.SuccessWithPriorityBypassPhantomReadCheck(nil,
        pb.Priority_HIGH)
}
```

5.3 APPLICATION DEPLOYMENT

After the network is deployed, applications can interact with it through chaincodes. The *peer* commands are a tool to transact in the network, but calling terminal commands and storing their return from applications developed in a general-purpose programming language is impractical. For that reason, Hyperledger Fabric provides SDKs.

This Section describes how Fabric's Java SDK and Java Gateway served as tools to build applications related to our chaincode. Some modifications on the SDK and Gateway were required to satisfy our model and experiment needs. We implemented one application for each stakeholder in our network: buyer, seller, sensor, utility company, and payment company.

5.3.1 Fabric SDKs

Hyperledger Fabric documentation defines the SDKs as *"a layer of abstraction on top of the wire-level protobuf based communication protocol used by client applications to interact with a Hyperledger Fabric blockchain network"* (TEAM, F., 2020b). Until Fabric version 2.3, there were Fabric SDKs in four programming languages: Java, Javascript, Go, and Python. All actions and commands performed in Section 5.1 could have been done with SDK functions, as they possess similar resources.

SDKs have some differences from one another. As an example, the Python SDK supports mainly 1.4.x Fabric versions, while the others support the 1.4.x and 2.x.x. The Java SDK is the only one with Idemix support. In our case, since we deal with Idemix MSP, we write our applications using the Java SDK. More specifically, we work with the Fabric Java Gateway.

The *fabric-sdk-java* had to be adapted to allow our required needs when dealing with idemix credentials. The modifications, highlighted in red, presented in Codes 5.27 and 5.28 enabled calling serializing methods outside the *fabric-sdk-java* package. After signing a transaction, we desired to store the used idemix pseudonym to later transaction authorship prooving. Codes 5.29 and 5.30 display the necessary changes to achieve that. The whole *fabric-sdk-java* was recompiled and used as a dependency to recompile the *fabric-gateway-java*.

Code 5.27 – Allowing outside package access to the serialization method of *Idemix-Credential* class

```
public Idemix.Credential toProto();
```

Code 5.28 – Allowing outside package access to the serialization method of texti-tIdemixIssuerPublicKey class

```
public Idemix.IssuerPublicKey toProto();
```

Code 5.29 – Adding a method to retrieve the *SigningIdentity* from the *Transaction-Context* class

```
public SigningIdentity getSigningIdentity() {
    return signingIdentity;
}
```

Code 5.30 – Allowing outside package access to the random part of the *Idemix-Pseudonym* class

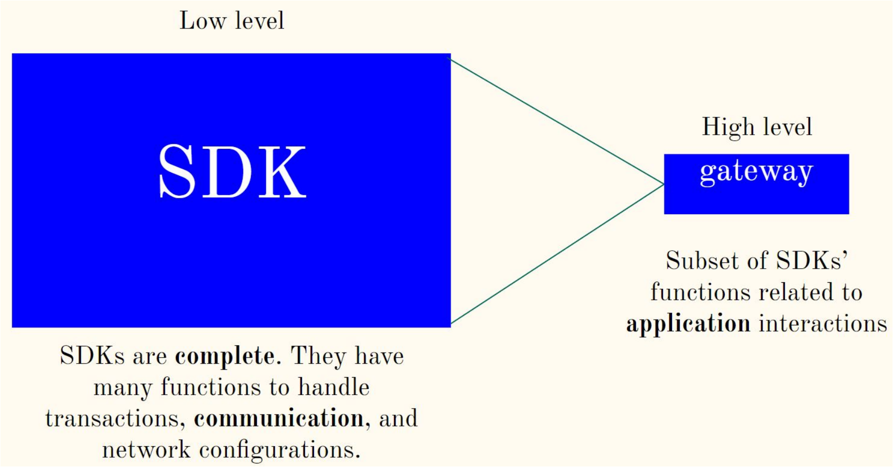
```
public BIG getRandNym() {
    return RandNym;
}
```

5.3.2 Fabric gateways

Fabric gateways provide minimal necessary functions to applications submit transactions and query ledger contents (TEAM, F., 2020a). It is built from a subset of Fabric SDKs methods. Figure 19 present the relation between SDKs and gateways.

We chose fabric-gateway-java as the tool to interact with our Fabric network. Fabric-gateway-java currently supports only identities with x509 certificates, even

Figure 19 – SDK vs. Gateway comparison



Subset of SDKs' functions related to **application** interactions

SDKs are **complete**. They have many functions to handle transactions, **communication**, and network configurations.

Designed by the author

though the fabric-sdk-java supports x509 and Idemix. Since we intended to test our model's privacy solutions, we followed the x509 identity example and implemented the required Identity interfaces to provide Idemix support.

We implemented the classes *IdemixIdentity*, *IdemixIdentityImpl*, and *IdemixIdentityProvider* to support Idemix (WESTPHALL, 2021). We also modified the *Identities*, *GatewayImpl*, and *WalletImpl* classes. Codes 5.31, 5.32, and 5.33 present the signature of the added/modified methods highlighted in **red**.

Code 5.31 – Added methods to the gateway class *Identities*

```
public static IdemixIdentity newIdemixIdentity(final String mspId, Path
   ipkPath, Path revocationPkpath,
            Path signerConfigPath);

public static IdemixIdentity newIdemixIdentity(final String mspId, final
   IdemixEnrollment enrollment);

public static IdemixIdentity newIdemixIdentity(final String mspId, final
   IdemixIssuerPublicKey idemixIpk, final PublicKey revocationPublicKey,
   final JsonObject signerConfigJson) );

public static PublicKey readPublicKey(final String pem);

public static PublicKey readPublicKey(final Reader pemReader);

private static SubjectPublicKeyInfo asSubjectPublicKeyInfo(final Object
```

```
    pemObject);

public static String toPemString(final PublicKey publicKey);
```

Code 5.32 – Modifications on methods in the *GatewayImpl* class

```
@Override
public Builder identity(final Identity identity) {
    if (null == identity) {
        throw new IllegalArgumentException("Identity must not be null");
    }
    if (!(identity instanceof X509Identity || identity instanceof
        IdemixIdentity)) {
        throw new IllegalArgumentException("No provider for identity type:
            " + identity.getClass().getName());
    }
    this.identity = identity;
    return this;
}

private HFClient createClient() {
    HFClient client = HFClient.createNewInstance();
    // Hard-coded type for now but needs to get appropriate provider from
        wallet (or registry)
    if (identity instanceof X509Identity)
        X509IdentityProvider.INSTANCE.setUserContext(
        client, identity, "gateway");
    else if (identity instanceof IdemixIdentity)
        IdemixIdentityProvider.INSTANCE.setUserContext(
        client, identity, "gateway");
    return client;
    }
```

Code 5.33 – Modifications on map declaration in the *WalletImpl* class

```
public final class WalletImpl implements Wallet {
    private final WalletStore store;
    private final Map<String, IdentityProvider<?>> providers = Stream
            .of(new IdentityProvider<?>[]
            {X509IdentityProvider.INSTANCE,
            IdemixIdentityProvider.INSTANCE})
            .collect(Collectors.toMap(IdentityProvider::
            getTypeId, provider -> provider));
```

```
    }
```

Furthermore, we enabled access to the *TransactionContext* instance of a transaction, as presented in Codes 5.34 and 5.35. After our *fabric-sdk-java* modifications, the transaction context has a public method to retrieve the *SiginingIdentity* of any message. With that, a buyer application can store the *IdemixPseudonym* to prove the credentials ownership to a utility company by signing a message with the same *IdemixPseudonym* as the *BuyBid* transaction.

Code 5.34 – Adding the getTransactionContext() method declaration to the fabric-gateway-java class *Transaction* class

```
TransactionContext getTransactionContext();
```

Code 5.35 – Adding the getTransactionContext() method implementation to the fabric-gateway-java class *TransactionImpl* class

```
    @Override
public TransactionContext getTransactionContext() {
    return transactionContext;
}
```

5.3.3 Applications implementation

The applications were implemented with our modified *fabric-gateway-java* version, even though only the buyer's and utility's applications had this unavoidable requirement due to idemix utilization. The other entities' applications could have been implemented in any other fabric-sdk supported language.

Before transacting with the network, we provide three ways to load the certificates or idemix credentials. They can be fetched by **enrolling** with the MSP's CA, loading from the **file system**, or loading from a *fabric-gateway-java* **wallet**. The only difference between the file system and wallet loading is the storage format. The first loads the x509 certificate and the private key directly, while the other loads a JSON formatted file containing both the certificate and private key.

Peers, orderers, and CAs are defined in a configuration file describing their addresses and some network characteristics. Because our network secures communication with TLS, the configuration file also has the path to the entities' TLS CAs certificate. We did not enable the TLS mutual authentication. Therefore the applications are not authenticated to the orderers and peers.

5.3.3.1 Sensor's application

A sensor with a valid MSP certificate reads an environment metric, and the data is stored in the blockchain after a call to the chaincode function *publishSmartData*. The coordinates considered by the chaincode are present in the sensor's certificate attributes. Depending on the sensor's processing constraints, the blockchain interaction and the certificate management would probably be performed by a gateway device. Code 5.36 exhibits the described sensor's action and, since it is a testing source, we deal with random smart data instead of a real measure.

Code 5.36 – How sensors publish *SmartData* to the blockchain

```
SmartData smartData = getRandomSmartData(unit, threadNum, publish);
Transactiontransaction = contract.createTransaction("publishSensorData");
byte[] transactionResult =
   transaction.submit(Byte.toString(smartData.version),
    Long.toString(smartData.unit), Long.toString(smartData.timestamp),
    Double.toString(smartData.value), Byte.toString(smartData.error),
    Byte.toString(smartData.confidence), Integer.toString(smartData.dev));
```

5.3.3.2 Buyer's application

As displayed in Code 5.37, before the buyer can publish *BuyBids* to the network, they need to put funds in their payment account and request a token from their payment company. We establish a representative Hypertext Transfer Protocol (HTTP) connection between the buyer and the payment company, even though a real deployment would require secure communication. After the *registerBuyBid* call carrying the *BuyBid* information, the buyer requests that the payment company validates it.

When the auction occurs, and the *BuyBid* is effectively matched to a *SellBid*, the buyer requests a nonce to their utility company. Then, they sign the transaction ID concatenated to the nonce and add the bid information to a list. The energy bill discount request is performed by the auction event listener presented in Code 5.38, which communicates with the utility through HTTP.

Code 5.37 – Buyer's application main function calls

```
putFundsOnPaymentAccount(1000);

String token = requestPaymentToken();

Transaction transaction = contract.createTransaction("registerBuyBid");
byte[] transactionResult =
   transaction.submit(cmd.getOptionValue("paymentcompanyid"), token, "UFSC",
```

```
    cmd.getOptionValue("energyamountkwh"),
        cmd.getOptionValue("priceperkwh"), cmd.getOptionValue("energytype"));

requestBuyBidValidation(token);
...
publishedBids.add(new PublishedBuyBid(paymentCompanyId, token,
    transactionID, ipk, signingId, Double.parseDouble(energyQuantity)));
```

Code 5.38 – Piece of auction event listener code

```
Consumer<ContractEvent> auctionPerfomedListener = new
    Consumer<ContractEvent>() {
    @Override
    public void accept(ContractEvent t) {
        if (t.getName().equals("auctionPerformed")) {
            int utilityNonce = getUtilityCompanyNonce();
            ...
            requestEnergyDiscount(buyerFullName,
                publishedBid.transactionID, publishedBid.ipk,
                ipkOwnershipSignatureProof);
            ...
        }
    }
};
contract.addContractListener(auctionPerfomedListener, "auctionPerformed");
```

5.3.3.2.1 Random generation configuration

By Hyperledger design, each new transaction signed with idemix requires a new pseudonym. The pseudonym creation uses java *SecureRandom* and, in Linux, the secure random default algorithm is *NativePRNG*. In this method, seeds are fetched from */dev/random*, consuming much time in an operating system lacking entropy sources. In a test context, buyer applications simulating multiple buyers will recurrently block and wait for the */dev/random* to print entropy.

To surpass this restriction, in our tests, we execute the buyer applications with the flag in red displayed in Code 5.3.3.2.1. This forces *SecureRandom* to use a Deterministic Random Bit Generator (DRBG) algorithm, which does not block for that long, with default configuration displayed in Code 5.39.

```
mvn exec:java@buyer-test -Dexec.mainClass="applications.AppBuyerForTest"
-Djava.security.egd=file:/dev/./urandom
```

Code 5.39 – *java.security* default configuration for DRBG algorithms

```
# The default value is an empty string, which is equivalent to
#   securerandom.drbg.config=Hash_DRBG,SHA-256,128,none
```

5.3.3.3 Seller's application

The seller's application, in Code 5.40, follows the same principles of the buyer one, but with a few extra function calls. For testing purposes, we configured the seller's application to publish energy generation transactions and then create a sell bid for the generated energy. However, in a real deployed network, the role of publishing energy generation claims could be performed by a gateway connected to the seller's smart meter.

Like the buyer's application, there is also an auction event listener in this one, as demonstrated by line 1 of Code 5.40. The only difference being that in this case, the event handler deals with requesting funds for the payment company due to the energy sold.

Code 5.40 – Seller's application main function calls

```
1 registerAuctionEventListener(contract, (X509Identity) identity,
     publishedBids, sellerFullName);
2 ...
3 transaction =
     contract.createTransaction("publishEnergyGenerationTestContext");
4 transaction.submit(generationBeginningTime, generationEndTime, "solar",
     randomGeneratedEnergy);
5 ...
6 Transaction transaction = contract.createTransaction("registerSellBid");
7 byte[] transactionResult =
     transaction.submit(cmd.getOptionValue("energyamountkwh"),
8     cmd.getOptionValue("priceperkwh"), cmd.getOptionValue("energytype"));
9 ...
```

5.3.3.4 Utility's application

The utility company application consists of a representative *HTTP* server receiving nonce and discount requests from buyers, as presented in Code 5.41. The nonce is solicited by clients who bought energy anonymously in the blockchain and increases the security of the discount request. The utility application performs the following steps before granting bill discounts to clients:

1. Receive a discount request containing the *client name*, the *ID* of the transaction

that published their *BuyBid*, the idemix issuer public key, and the *ID* concatenated to the retrieved nonce.

2. Verify if the transaction stored a *BuyBid* in the blockchain.
3. Retrieve the idemix *pseudonym* that signed the *BuyBid* publishing transaction.
4. Verify if the same *pseudonym* generated the signature described in step 1.
5. Verify if the *BuyBid* was partially or fully matched in an auction.

Code 5.41 – Utility application HTTP server resources and start

```
HttpServer server = HttpServer.create(new InetSocketAddress(80), 0);
server.createContext("/noncerequest", new NonceRequestHandler());
server.createContext("/discountrequest", new DiscountRequestHandler());
ExecutorService executor = Executors.newCachedThreadPool();
server.setExecutor(executor);
server.start();
```

5.3.3.5 Payment company’s application

The payment company is also a representative HTTP that receives four types of requests displayed in Code 5.42. A buyer can put funds in their account by posting to the resource “*/putfunds*” and request a payment token to compose the *BuyBid* by posting to “*/gettoken*.” After the *BuyBid* is published, the buyer will post to the “*/validatebuybid*” resource to require the payment company validation. After the auction, the seller will post to “*/requestpayment*” to get paid for the sold energy. At the execution beginning, the application retrieves all CA *MSP’s* certificates to attest later that a seller trying to request payment for an *EnergyTransaction* is the designated receiver.

Code 5.42 – Payment company application HTTP server resources and start

```
HttpServer server = HttpServer.create(new InetSocketAddress(81), 0);
server.createContext("/putfunds", new PutFundsHandler());
server.createContext("/gettoken", new GetTokenHandler());
server.createContext("/validatebuybid", new ValidateBuyBidHandler());
server.createContext("/requestpayment", new RequestPaymentHandler());
ExecutorService executor = Executors.newCachedThreadPool();
server.setExecutor(executor);
server.start();
```

5.3.4 Fabric-sdk-java logging and configurations

It is possible to activate the fabric-sdk-java logging so the application execution might be tracked. Even though the instructions in the repository (TEAM, F., 2020b) claim that logging is enabled through environment variable settings, we achieve it by passing the flags in red presented in Code 5.3.4 when we run an application. These two flags point to files that contain the logging configurations.

```
mvn exec:java@auction -Dexec.mainClass="applications.AppPeriodicAuction" -Dexec.args="..." -Djava.util.logging.config.file=commons-logging.
properties -Dlog4j.configuration=log4j.properties
```

The SDK's *Config* class defines all default configurations regarding wait times, security parameters, maximum thread numbers, and other configurations. A "*config.properties*" file in the root of the maven project directory is necessary to override the default parameters. In this file, we only change the *Service Discovery* period to avoid frequent discovery requests and increase performance. We also change the orderer default response timeout.

5.3.5 Service Discovery x Network file description

Fabric's java SDK lets applications know the network topology, with peers, orderers, channels, and CAs, by performing a *Service Discovery* or reading a network description file. When an application starts, the network description file is always read to contact peers and orderers. Then, it can perform a *Service Discovery* to find out about all known peers, orderers, policies, and settings in the channel. After that, the next transactions are sent only to the needed number of endorsers based on the discovered channel policy.

Nothing prevents the network description file is solely used, but this practice can impact the overall channel transaction performance. Since the endorsement policies are unknown to the applications without *Service Discovery*, they will request endorsements for **all** known peers, even if the number of requests exceeds the quantity required by the endorsement policies.

A network with two organizations *O1* and *O2*, running two peers each, with the endorsement policy determining that all organizations must sign every transaction clarifies the performance problem. An application transacting with the network, which uses a network description file containing all four peers, would request an endorsement for each one, even though only two peers would be needed. This does not represent a significant performance impact in a small-scale channel, while the opposite is true for large-scale channels dealing with multiple application instances.

Solving this problem is not as simple as describing the sufficient peers' quantity because the channel topology might change as a peer or orderer goes offline. That is why we opted for enabling the *Service Discovery* after analyzing both behaviors through logging and debugging. Its main performance advantage consists of alternating the endorsing peers, avoiding a bottleneck on a specific peer while making the least requests.

In our experiments with a private docker network, the *Service Discovery* worked properly when the applications were executed from an ubuntu docker container inside the network. Otherwise, despite the fabric-sdk-java overriding the hostname to *localhost*, it does not support port binding as configured with docker. We also changed the default *Service Discovery* period of 2 minutes to 20, as we considered it a reasonable time interval for preserving performance.

5.4 NETWORK AWS DEPLOYMENT

In the previous sections of this chapter, we explained the steps to deploy a localhost containerized blockchain network with a single chaincode and some applications. Experiments in a local context are difficult to scale and are not always representative of a real environment. For this reason, we created the necessary scripts and architecture to adapt the local deployment to an AWS one.

AWS offers vast cloud solutions with products and services related to many different knowledge areas, from databases to robotics. It counts on hosts located in the world's main regions and allows that customers choose where to applications are deployed. The users have a variety of machine configurations at their disposal, depending on resource requirements and budget.

5.4.1 Elastic Compute Cloud

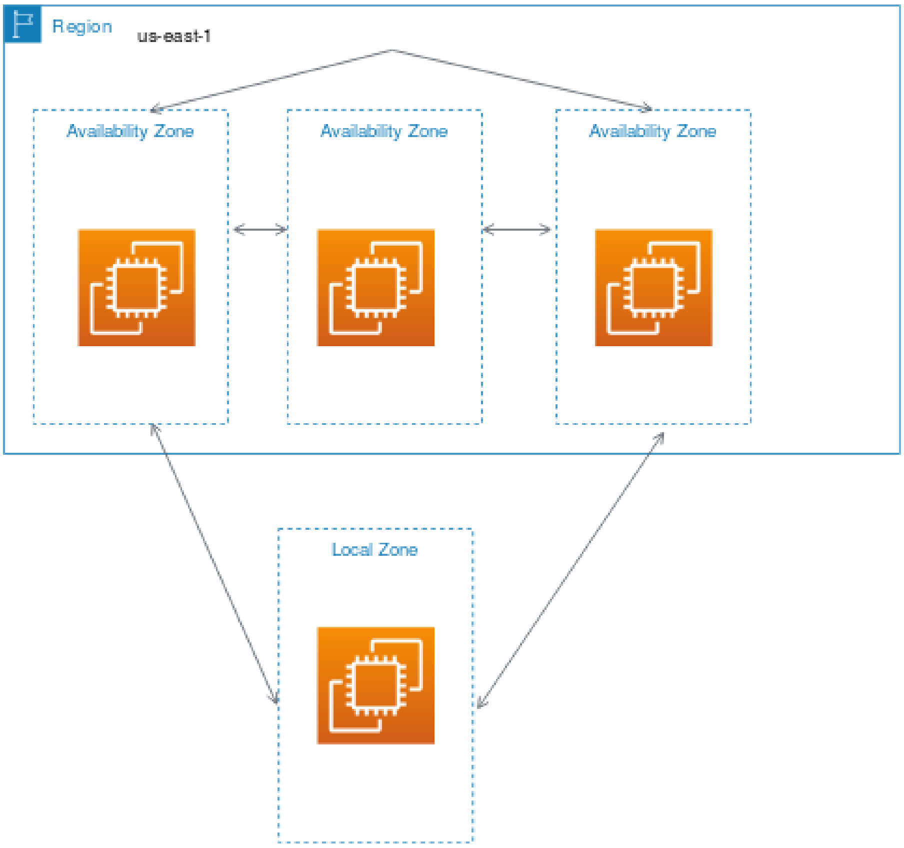
We selected the **Elastic Compute Cloud (EC2)**, a virtual configurable computing environment, to deploy our cloud blockchain network. The general steps to set up a network go through selecting an operational system image (**AMI**), choosing the hardware configurations for each virtual machine (**instance**), and defining their storage type (**SSD**, **IOPS SSD**, **Hard disk**).

An Amazon Machine Image (AMI) stores an operating system snapshot with configurations and applications and can be used by multiple instances. AWS lists the available AMIs and, if they are public, anyone can start an instance, modify the system as wished, and store the changed AMI. The supported architectures for images are i386, amd64, and arm64.

Instances are virtual servers with specific hardware configurations and launch from an AMI. Each instance type runs in a specific processor architecture with determined virtual Central Process Unit (CPU), Random Access Memories (RAM), local storage, and network throughput quantities. For example, the instance type *t2.micro* runs with a single intel virtual CPU, 1 Gibibyte (GiB) of RAM, and low network performance. Meanwhile, the *r6gd.metal* type has 64 *physical* cores of an AWS Arm Gravitron2 Processor, containing 512 GiB of *physical* RAM and network bandwidth up to 25 Gigabits per second (Gbps).

The machines are instantiated in a *Region*, and more specifically, in an *Availability Zone* inside a *Region*. As an example, *AWS* has the South America East *Region*, located in Sao Paulo. *Availability Zones* represent different data centers within the *Region* bounds and reduce the single point of failure risk. If an application resides in multiple *Availability Zones*, a power shortage in one of them could be mitigated, as presented in Figure 20.

Figure 20 – AWS Regions and Availability Zones



Source: (AUTHORS, 2021)

5.4.2 ARM vs. x86-64 deploy and costs

Powerful compute instances were required to run our experiments as close as possible to a real blockchain network for energy transactions - with many sellers, buyers, and sensors. We analyzed the costs to run instances with at least 32 cores and realized the cost disparity between arm and intel instances for budgeting the experiments. The on-demand Linux pricing for the type *m5.16xlarge* (intel) was 3.232 USD per hour, while the arm equivalent *m6g.16xlarge* cost 1.6191 USD per hour.

This 50% difference motivated us to deploy our experiments fully on arm instances, but the peer docker image had a hard-to-diagnose linking segmentation fault. We could solve the linking problem by forcing the docker images responsible for compiling the peer to link **statically**, with the changes displayed in Code 5.43. Besides, we changed the fabric-sdk-java dependency *netty-tcnative-boringssl-static* from version *2.0.34.Final* to *2.0.35.Final* due to a bad arm64 Dynamic-link Library (DLL) naming.

Code 5.43 – Modification on Fabric's *images/peer/Dockerfile* to allow compilation in arm64 and static linking

```
...
RUN apk add --no-cache \
    binutils-gold \
...
# peer must be built STATICALLY to run in arm64 docker
ENV CGO_ENABLED=0
RUN make peer GO_TAGS=${GO_TAGS}
...
```

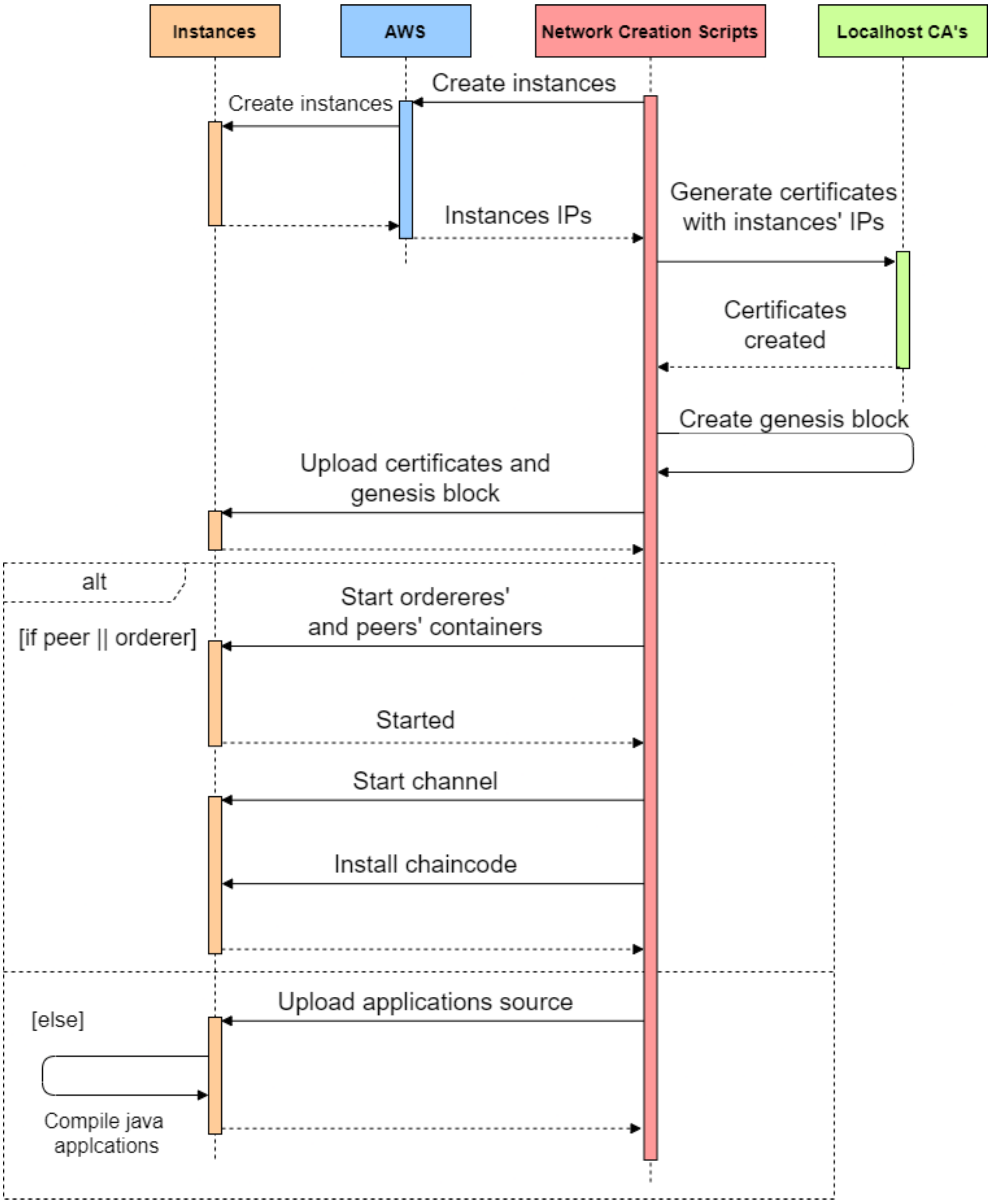
The deployment location also affects costs because *AWS* charges different prices for different *Regions*. Since our experiment is executed fully on the cloud, we could choose the cheapest location, Mumbai. As a reference, the same instance that cost 1.6191 USD in Mumbai could cost up to 3.9168 USD in Sao Paulo.

5.4.3 EnergyNetwork deploy steps in AWS

The first step involved creating modified *AMI*s from the quick start *Ubuntu Server 20.04 LTS (HVM)* image (ami-0a6638920f7143fb2) for arm architecture. We started an instance with this quick start image and installed all required packages, including our patched *fabric* docker images, the *fabric-sdk-java*, and the *fabric-gateway-java*. The script "*create-ami-arm.sh*" in our Github repository (WESTPHALL, 2021) was executed to generate our custom Ubuntu AMI.

Next, we adapted the local docker deploy script to the "*automated-aws-creation.sh*" script, which receives the instance types as arguments, reads the configuration files and deploys the network in AWS. Figure 21 shows the high-level operations of the deployment with peers, orderers, a chaincode, and application instances. Localhost CAs generate the certificates for the hostnames, or Internet Protocol (IP) addresses allocated by AWS to each instance.

Figure 21 – AWS deploy high-level sequence



Designed by the author

6 EXPERIMENTS

This chapter describes the experiments performed locally, in a single machine, and in the AWS cloud with multiple Cloud machines. Also, we define the experiments' objectives and explain some applied techniques to increase the network load in terms of scalability efficiently. In AWS, we performed three phases of experiments with continuous load increases and instance upgrades to measure the network capacity. The data generation rate by the network referent to the experiment phases was also analyzed.

6.1 EXPERIMENT DESIGN GOALS

We designed our experiments to reach answers to our research questions about the scalability of blockchain energy trading with a guarantee of origin. The focus is on how a Hyperledger Fabric network running a chaincode to validate and transact energy, handling vast amounts of transactions. For that, we test different network configurations, varying the quantities of organizations, buyers, sellers, and sensors.

The frequency of transactions - energy validation, auction, buying, and selling - is also configurable. With these settings, we can discover how a specific configuration change impacts the performance of the network. For example, the increase in the frequency of the sensors data publication might require more peers to keep the network operational.

6.2 EXPERIMENT ADAPTATIONS

Our applications and chaincode functions had to be adapted to support larger-scale experiments. HTTP servers and creating a certificate for each buyer, seller, and sensor had to be eliminated. Chaincode execution averages had to be calculated without affecting performance. In this section, we enumerate the required adaptations and briefly explain them.

6.2.1 Test applications

The applications described in Section 5.3.3 were adapted to simulate a higher number of buyers, sellers, and sensors. Each one has a specific purpose of simulating only one type of entity - buyer, seller, or sensor. These applications receive arguments about the transaction period, the number of transactions, path to certificates, or idemix credentials. The quantity of simulated sellers, buyers, or sensors is also passed as an argument and implies creating one **thread** per buyer, seller, or sensor.

To ensure that the auction happens periodically, we developed a specific application that only calls the *auction* chaincode function and sleeps after that. The auction period is passed as an argument and is kept the same in an experiment round.

6.2.2 Bypassing entities identification from certificates' common names

In our initial network deployment, the chaincode identified transactions' authors by the common names in the certificates. However, if we kept this design in our experiments, the required number of certificates would be proportional to the number of buyers, sellers, and sensors. This is not a problem in small tests, but it can be in larger experiments like ours.

Therefore, we enabled that the transaction author identified themselves by passing their names in the chaincode function parameter. For that, we added chaincode test functions that receive the author's name in order to identify them. Code 6.1 displays an example of an adapted test function. The *sellerID* is informed by the chaincode caller, as presented in Code 6.2.

Code 6.1 – Chaincode function adapted to fetch ID from parameter

```
func (chaincode *EnergyChaincode) registerSellBidTestContext(stub
    shim.ChaincodeStubInterface, sellerID string, quantityKWH float64,
    pricePerKWH float64, energyType string) {...}
}
```

Code 6.2 – Application function that calls the chaincode function and passes the *sellerFullName* as parameter

```
transaction = contract.createTransaction("registerSellBidTestContext");
transaction.submit("seller1-organization", ...);
```

6.2.3 One gateway per multiple entities of the same type to improve thread efficiency

We utilize a single *gateway* to handle all the transaction submissions of a specific - buyer, seller, or sensor - simulated application. This avoids redundant connection thread creations to serve each simulated buyer, sensor, or seller. Code 6.3 presents an example with the buyer's application. All threads simulating a buyer interact with the network through the same *gateway* instantiated, in line 1 of Code 6.3. Otherwise, the capacity to simulate entities would decrease since more CPU power would be allocated to the *Remote Procedure Calls (gRPC)* threads.

Observation: On some rare occasions, the *gRPC* java library shows exceptions logging messages - without actually throwing the exceptions - warning about channels previously open that were not properly shut down. This does not affect the delivery of transactions, and it is caused by dereferenced objects being finalized by the garbage collector instead of being explicitly shut down.

Code 6.3 – Same gateway for all buyer threads

```
try (Gateway gateway = builder.connect()) {
    Network network = gateway.getNetwork("canal");
    Contract contract = network.getContract("energy");
    ...
    for (int i = 1; i <= THREAD_NUM; i++) {
        threads[i] = new Thread() {
        ...
        public void run() {
            ...
            contract.createTransaction("registerBuyBid");
            ...
        }
    }
}
```

6.2.4 Sensor application without block event

By default, the fabric-java-gateway always waits for the transaction commitment event after a submission. Unlike a buyer, which needs to wait for the BuyBid publication commitment to request the BuyBid validation, sensors simply push data to the chain and do not require knowing when the data is committed. Also, considering that the sensors' gateways might have processing constraints, listening to events should be disabled. We achieve this behavior by creating a specific network file description in which all peers are not set as event sources, as presented in Code 6.4.

Code 6.4 – Non-blocking *conection-tls.json*

```
"peers": {
    "peer1-org1": {
        ...
        "eventSource": false,
    },
    "peer1-org2": {
        ...
        "eventSource": false,
    }
}
```

6.2.5 Discarding HTTP servers to improve experiment reliability

The applications simulating the utility company's and payment company's HTTP server were discarded in our experiments. As we intended to focus on measuring the blockchain performance, the risk of the HTTP servers becoming the network bottleneck had to be eliminated. It is true that the discards also free some blockchain resources which would be required to respond to the companies' queries, making our experiments divert a little from a real deployed network. Nevertheless, we are sure that Hyperledger Fabric is the only one responsible for the measured performance limits.

The initially used Java HTTP library (*com.sun.net.httpserver.HttpServer*) is not very scalable, as it blocks threads to perform chaincode calls. Considering this behavior, we did not want to increase this work's complexity by finding and analyzing another Java HTTP library. However, we address joining back properly configured scalable HTTP servers as future work.

Instead of the payment company validating the buy bid, the buyer application validates its published buy bids in our tests to bypass the companies' absence. Also, all sensors are considered trusted by all organizations to avoid the necessity of setting the trusted sensors for each organization in the experiment context.

Observation: even without the companies, we maintained the buyers and sellers auction event listeners that normally would trigger discount and payment requests. Since there are no companies in our tests, the buyers and sellers only query the chaincode to check if their bids were fully matched after receiving the auction event notification. Therefore, auction events and bid queries still represent a chaincode load.

6.2.6 Measuring chaincode execution

When a chaincode function is called, the execution time is measured and sent to a *goroutine* exclusively responsible for incrementally calculating the average time of each function to avoid overflow. Due to the same function being called simultaneously by multiple endorsing requests, this approach using a *goroutine* ensures that the writes to the averages are coordinated and prevents the endorsing requests from blocking until the average calculation.

Code 6.5 shows the thread responsible for executing the transaction simulation. After the return, the *defer* function calculates the execution time and sends it to the goroutine ("thread") presented in Code 6.6. The calculation goroutine continuously listens to the channel *channelAverageCalculator* and recalculates the function execution time average when a new *FunctionAndDuration* is published.

Code 6.5 – Main execution “thread”

```
now := time.Now()
defer func() {
    elapsed := time.Since(now)
    channelAverageCalculator <- &FunctionAndDuration{functionName, elapsed}
}()
return functionPointer(stub, args)
```

Code 6.6 – Average calculation goroutine (“thread”)

```
for {
    functionNameAndDuration := <-channelAverageCalculator
    /* recalculate the average */
}
```

6.2.7 Limiting the number of sensors during validation

In some cases, we increase the number of sensors to more than 1000. Since, in our experiments, we keep all sellers within an influence radius of the sensors, we established a maximum limit of **5** sensors per organization to validate an energy generation claim. This avoids the validation complexity to grow proportionally to the *number of sellers* times the *number of sensors*, which would be unrealistic in a real-life scenario.

At the first experiments, every sensor participated in the validation of every energy generation claim. However, we noticed that such settings started requiring increasingly powerful instances in tests with more than 1000 buyers, 1000 sellers, and 1000 sensors. Therefore, scalability tests containing more than 10000 seemed infeasible. The **5** sensor maximum made our experiments closer to a real-life scenario while keeping the sensors’ load related to *SmartData* publishing and allowing further scalability tests.

6.3 EXPERIMENT ROUNDS

We structured our scripts to execute experiments in rounds, which are defined by a set of configurations and results. The configurations encompass blockchain configurations, like the number of orderers, and application configurations, like the sensors quantity. Results describe metrics and statistics within a single experiment round.

6.3.1 Experiment round configuration

Each experiment round has its own set of configurations regarding the blockchain - peers, orderers, organizations - and the applications - buyers, sellers, sensors, transaction frequency. The following configurations are determined before network creation:

1. Organizations.
2. Number of peers.
3. Number of orderers.
4. Number of application instances.
5. Peer concurrency limits.
6. Peers AWS instance type.
7. Orderers AWS instance type.
8. Applications AWS instance type.
9. Batch size.
10. Batch timeout.

After the network creation, multiple experiments might be performed, but the only possible configuration changes are related to the applications, displayed in Code 6.7. Experiments with different blockchain configurations require full network recreation.

Code 6.7 – Application test configuration

```
#quantity per cli-application
sensors:
  quantity: 1000
  unit: 3834792229
  #Interval in ms
  publishinterval: 5000
  publishquantity: 20
  msp: UFSC

sellers:
  quantity: 1000
  #Interval in ms
  publishinterval: 5000
  publishquantity: 5
  msp: UFSC


buyers:
```

```
  quantity: 1000
  #Interval in ms
  publishinterval: 5000
  publishquantity: 10
  msp: IDEMIXORG

#Interval in ms
auctioninterval: 30000
```

6.3.2 Experiment round results

During an experiment round, the peers', chaincodes', orderers', and applications' metrics are continuously fetched through Secure Shell (SSH) and dumped into files. Code 6.8 displays an example of how we do that. The *sshCmdBg* function connects to an instance and executes the "docker stats" command in the background until all applications finish their transactions. These CPU, memory, network, and disk stats are later processed by a python script that generates plots from the data.

Code 6.8 – Fetching orderers' stats

```
sshCmdBg ${hosts[orderer$i-$orgName]} docker stats --format
    "{{.CPUPerc}}:{{.MemUsage}}:{{.NetIO}}:{{.BlockIO}}" orderer$i-$orgName
    > $testFolder/stats-orderer$i-$orgName.txt
```

The initial and final file system sizes of peers and orderers are also measured to identify how much data an experiment round generated. We periodically get the average execution time of each chaincode function. In the end, we retrieve the peers', orderers', and applications' logs to verify if any abnormal behavior happened. Eventually, when stress testing the network, some transactions might be rejected, or connection timeouts might appear in the logging files.

6.4 EXPERIMENTS WITH DIFFERENT AWS INSTANCES

We evaluated the AWS instances' performance in a context with **one orderer**, **one peer**, and **one or two application instances**. Algorithm 1 provides a high-level idea about how we performed these experiment tests. We started with the limited arm instance *t4g.micro* for the peer, orderer, and application. The test load was constantly increased by growing the number of sellers, buyers, and sensors until the log indicated failures.

When the logs presented failures, we interpreted the test result to find what instance - peer, orderer, or application - needed upgrade to support the test load or if some Hyperledger Fabric's configuration should be changed. After the instance upgrade or configuration change, the experiment round was run again with the same load and was expected to work.

Algorithm 1 Experiment tests

```
procedure TestInstancesLimit
    peerInstance ← "t4g.micro"
    ordererInstance ← "t4g.micro"
    applicationsInstance ← "t4g.micro"
    testConfiguration ← getSmallLoadConfiguration
    while weWantToPerformAnotherRound do
        logs ← runExperimentRound(testConfiguration)
        if logsPresentsErrors(logs) then
            configurationNeedsChange ← identifyConfigChangeNeed(logs)
            if configurationNeedsChange then
                testConfiguration ← changeSomeFabricConfig(testConfiguration)
                continue
            end if
            entityToBeUpgraded ← identifyWhoNeedsUpgrade(logs)
            if entityToBeUpgraded = "peer" then
                peerInstance ← upgradeInstance(peerInstance)
            else if entityToBeUpgraded = "orderer" then
                ordererInstance ← upgradeInstance(ordererInstance)
            else
                applicationsInstance ← upgradeInstance(applicationsInstance)
            end if
        else
            testConfiguration ← increaseLoad(testConfiguration)
        end if
    end while
end procedure
```

6.4.1 Phase 1 experiment

The first phase of the experiments ran with all sensors participating in all energy validations. It raised our awareness about limiting the number of sensors, as discussed in Section 6.2.7 after analyzing the steep average time increase of the energy validation function execution. This phase's results are discussed in Section 7.2.1.

In Phase 1, the transaction publication quantities and intervals of each entity type were equivalent to the displayed in Code 6.7. Nevertheless, the numbers of sellers, sensors, and buyers were varied. This phase ended with a very specific failure regarding the **peer endorsing concurrency limit**, set by default to **2500** concurrent endorsing requests.

6.4.2 Phase 2 experiment

In the Phase 2 experiments, we increased the peer's concurrency limit and avoid this failure, provided that the peer instance has the required computing power.

The limit was set to 1 million concurrent requests to practically eliminate any concurrency restrictions, even though we never experiment with 1 million concurrent transactions. As in Phase 1, besides the number of buyers, sellers, and sensors, the Phase 2 configuration was the same as presented by Code 6.7.

We also increased the chaincode container RAM allocation since the chaincode started to respond to more simultaneous requests after the concurrency limit increase. The number of sensors participating in each energy generation validation was limited to **5** to represent a more realistic scenario. This phase's results are presented in Table 6 in Section 7.2.2.

6.4.3 Phase 3 experiment

The authors of (MOON et al., 2019) recommend a larger block size in Hyperledger Fabric environments with high throughput and lower latency demands. Considering that, in Phase 3, we performed tests with different block sizes and block timeout values. The results are presented in Table 7 and discussed in Chapter 7.

These configurations are set in the "configtx.yaml" file of the network being deployed. Code 6.9 presents the parameters that deal with block configurations, containing brief documentation explaining each field. The term "batch" can be considered equivalent to "block." The results of this phase are presented and discussed in Section 7.2.3

Code 6.9 – Hyperledger block configuration parameters in "configtx.yaml"

```
Orderer: &OrdererDefaults
    #...
    BatchTimeout: 2s
    BatchSize:

        # Max Message Count: The maximum number of messages to permit in a
        # batch.  No block will contain more than this number of messages.
        MaxMessageCount: 500

        # Absolute Max Bytes: The absolute maximum number of bytes allowed for
        # the serialized messages in a batch. The maximum block size is this value
        # plus the size of the associated metadata (usually a few KB depending
        # upon the size of the signing identities). Any transaction larger than
        # this value will be rejected by ordering...
        AbsoluteMaxBytes: 10 MB

        # Preferred Max Bytes: The preferred maximum number of bytes allowed
        # for the serialized messages in a batch. Roughly, this field may be considered
        # the best effort maximum size of a batch. A batch will fill with messages
        # until this size is reached (or the max message count, or batch timeout is
        # exceeded).  If adding a new message to the batch would cause the batch to
        # exceed the preferred max bytes, then the current batch is closed and written
        # to a block, and a new batch containing the new message is created.  If a
        # message larger than the preferred max bytes is received, then its batch
        # will contain only that message.  Because messages may be larger than
        # preferred max bytes (up to AbsoluteMaxBytes), some batches may exceed
```

```
        # the preferred max bytes, but will always contain exactly one transaction.
        PreferredMaxBytes: 2 MB
```

6.5 DATA GENERATION RATE EXPERIMENTS

Sensors continuously sending SmartData to the blockchain coupled with energy bid submissions might generate a huge amount of data, and knowing this data generation rate can point to network limitations. For that reason, we evaluated the data quantity generated in Phases 2 and 3 - with sensors, buyers, and sellers.

The peer’s and orderer’s root (“/”) file systems’ sizes were measured before and after the experiment rounds. We intended to draw conclusions discussing whether or not the data rate could be considered a problem with the results.

7 RESULTS AND DISCUSSION

In this Chapter, we present the results of our experiments. The preliminary metrics were taken during the chaincode development to maximize the database queries speed. We compare CouchDB and LevelDB after running some of our chaincode's functions with each one of them. LevelDB presented an overall better performance for the queries executed. The results from the three experiment phases are displayed, discussed, and compared with the related work solutions and proposals. We estimate the data generation rate and the cost of our implementation.

7.1 PRELIMINARY METRICS

The preliminary metrics provided base metrics to decide on the best performance chaincode design. We compared the time to retrieve a set of ledger States between CouchDB and LevelDB queries. The time measures were taken in a single machine with two orderers, two peers, and two CouchDBs for each peer, all running in docker containers. The preliminary metrics' intent is not to estimate the World State's access time in a real deployment context but to find the differences between the two databases, which will likely be proportional in a real deployment.

Our chaincode final version serializes the structs using gRPC to take advantage of more efficient storage. However, at the time of these preliminary experiments, the chaincode serialized structs to **JSON** format, which is required for using CouchDB with Hyperledger Fabric.

7.1.1 CouchDB vs. Go LevelDB

Hyperledger Fabric supports CouchDB and LevelDB as database solutions to the ledger and World State. In this context, we measured some preliminary database query times to decide the most appropriate database. Our initial metrics involved querying *SmartData* in a determined timestamp range from the World State. The other two queries regarded retrieving *SellerInfo* by its *SmartMeter* ID and fetching *SellBids*/*BuyBids* to perform the double auction. Based on the metrics, LevelDB was considered more appropriate for our chaincode than CouchDB.

At first sight, it might seem obvious that a CouchDB running as an extra docker container, communicating with the peer using the network, and performing JSON queries would be slower than a LevelDB peer-local with only *Key-Value* queries. However, we considered that JSON queries could filter more data at the database, optimize network usage, and avoid data filtering on the chaincode. Also, we created *indexes* for the JSON queries, as recommended in Hyperledger Fabric's documentation, for optimization (TEAM, F. D., 2020b).

Since we performed these preliminary experiments in a single machine, the network delay effect was almost irrelevant on the communications between peers and its CouchDB instance and between the peer and its chaincode instance. Code 7.1 presents how the network delay was calculated by performing near 100 round trips with the Linux *ping* command. The CouchDB settings in our experiments are presented in Code 7.2.

Code 7.1 – Round trip peer to CouchDB and peer to chaincode

```
$ ping "couchdb-address"
...
round-trip peer<-->couchdb min/avg/max = 0.063/0.077/0.156 ms

$ ping "chaincode-address"
...
round-trip peer<-->chaincode min/avg/max = 0.050/0.073/0.165 ms
```

Code 7.2 – CouchDB configuration in each peer configuration file *core.yaml*

```
ledger:
  ...
  state:
    ...
    totalQueryLimit: 100000

    couchDBConfig:
       ...
       maxRetries: 3
       maxRetriesOnStartup: 10
       requestTimeout: 35s
       internalQueryLimit: 1000
       maxBatchUpdateSize: 1000
       ...
```

7.1.1.1 Querying SmartData by timestamp range

The *SmartData* by timestamp query is part of the energy validation process. The trusted near sensors are selected, and their published *SmartData* serve as references to validate the alleged energy production. The seller informs a timestamp range, representing the period when they generated the energy. We evaluated the performance of three ways to retrieve the smart data:

1. Using CouchDB, with a JSON query and function *shim.ChaincodeStubInterface.-GetQueryResult()* with the operator *$in*
2. Using CouchDB, with a JSON query and function *shim.ChaincodeStubInterface.-GetQueryResult()* without the operator *$in*
3. Using CouchDB, with the function *shim.ChaincodeStubInterface.GetStateByRange()*
4. Using LevelDB, with the function *shim.ChaincodeStubInterface.GetStateByRange()*

Codes 7.3 and 7.4 contain the measured queries available only with CouchDB, one with the *$in* operator, which requires only one query, including all sensors IDs. The other query is performed per near trusted sensor. Code 7.5 shows the index for the *SmartData* stored in the World State, containing the fields *timestamp* and *assetid*, corresponding to the same fields in our JSON query.

Code 7.3 – Query with *GetQueryResult()* using the *$in* operator, possible only with CouchDB

```
assetsIDs := "["
for _, nearTrustedActiveSensor := range *nearTrustedActiveSensors {
    assetsIDs += ‘"‘ + nearTrustedActiveSensor.MspID + nearTrustedActiveSensor.SensorID + ‘",‘
}
assetsIDs = assetsIDs[:len(assetsIDs)-1] + "]"

queryString := fmt.Sprintf(‘{"selector":{"timestamp":{"$gt": %d},"timestamp":{"$lt": %d},"assetid":{ "$in": %s }}}‘, t0, t1, assetsIDs)
queryIterator, err := stub.GetQueryResult(queryString)
...
}
```

Code 7.4 – Query with *GetQueryResult()* for each near trusted sensor, possible only with CouchDB

```
for _, nearTrustedActiveSensor := range *nearTrustedActiveSensors {
    assetID := nearTrustedActiveSensor.MspID + nearTrustedActiveSensor.SensorID
    queryString := fmt.Sprintf(‘{"selector":{"timestamp":{"$gt": %d},"timestamp":{"$lt": %d},"assetid":"%s"}}‘, t0, t1, assetID)

```

```
    queryIterator, err := stub.GetQueryResult(queryString)
    ...
}
```

Code 7.5 – SmartData CouchDB index

```
{
    "index":{
        "fields":["timestamp","assetid"]
    },
        "ddoc":"indexSmartDataDoc",
        "name":"indexSmartData",
        "type":"json"
}
```

The queries performed with the function *GetStateByRange()*, as shown in Code 7.6, return all states with keys in the range [*startKey*, *endKey*[, considering the lexicographical order. Since the *SmartData* keys end with a 20 character timestamp, it is possible to use this function to fetch the sensor published *SmartData* in the interval [*t0*, *t1*[.

Code 7.6 – Query with *GetStateByRange()*, possible with both CouchDB and LevelDB

```
objectType := "SmartData"
for _, nearTrustedActiveSensor := range *nearTrustedActiveSensors {
    startKey := objectType + nearTrustedActiveSensor.MspID +
       nearTrustedActiveSensor.SensorID + getMaxUint64CharsStrTimestamp(t0)
    endKey := objectType + nearTrustedActiveSensor.MspID +
       nearTrustedActiveSensor.SensorID + getMaxUint64CharsStrTimestamp(t1)
    queryIterator, err := stub.GetStateByRange(startKey, endKey)
    ...
}
```

We repeated the queries 100 times for each of the two near trusted sensors. There were 2000 *SmartData* stored in the World State, all of them sent by one sensor. Table 2 exhibits the settings and time to perform a query quantity for each query presented in Codes 7.3, 7.4, and 7.6. In the specific case of Table 2 first line, we performed 100 queries total because both *sensor IDs* were placed in the same query.

Settings and time Database - query method	#Near trusted sensors	#SmartData in World State	Number of queries	Time to perform all queries	Avarage time/query
CouchDB-JSON *GetQueryResult* with ($in)	2	2000	100	5m23.6821498s	3.23s
CouchDB-JSON *GetQueryResult* without ($in)	2	2000	200	15m19.0253594s	4.59s
CouchDB-*GetStateByRange*	2	2000	200	1m23.1892231s	0.41s
LevelDB-*GetStateByRange*	2	2000	200	606.9276ms	0.003s

Table 2 – Time and settings to fetch *SmartData* in a timestamp range by different methods

7.1.1.2 Querying SellerInfo

Every time a new seller is registered, the chaincode checks if their *SmartMeterID* is not already associated with another seller. When a smart meter or its gateway publishes an energy generation claim, the *SellerInfo* related to that smart meter must be fetched so that the energy generated can be linked to the seller. There are two possible ways to accomplish both searches, one is presented in Code 7.7, with a JSON query, and the other is presented in Code 7.8, with the *GetState()* function.

Code 7.7 – Query with *GetQueryResult()*, possible only with CouchDB

```
queryString := fmt.Sprintf(‘{"selector":{"mspsmartmeter":"%s","smartmeterid":"%s"}}‘, meterMspID, meterID)
queryIterator, err := stub.GetQueryResult(queryString)
...
    if queryIterator.HasNext() {
        queryResult, _ := queryIterator.Next()
        sellerInfoBytes = queryResult.Value
    }
```

Code 7.8 – Query with *GetState()*, possible with both CouchDB and LevelDB

```
objectType := "MeterSeller"
key, err := stub.CreateCompositeKey(objectType, []string{meterMspID, meterID})
meterSellerBytes, err := stub.GetState(key)
if meterSellerBytes == nil {
```

```
    return sellerInfo, fmt.Errorf("No meter of MSP %s and ID %s",
       meterMspID, meterID)
}
err = json.Unmarshal(meterSellerBytes, &meterSeller)

objectType = "SellerInfo"
key, err = stub.CreateCompositeKey(objectType,
   []string{meterSeller.MspIDSeller, meterSeller.SellerID})
sellerInfoBytes, err = stub.GetState(key)
    if sellerInfoBytes == nil {
        return sellerInfo, fmt.Errorf("No seller related to the meter of
           MSP %s and of Smart Meter ID %s", meterMspID, meterID)
    }
```

Table 3 presents the time to perform 100 queries, presents the time to perform 100 queries, given a certain query method and *SellerInfo* quantity stored in the World State. Based on the time column, it is possible to conclude that, besides LevelDB having the best performance, the *SellerInfo* quantity stored in the World State does not significantly influence when its full key fetches speed.

Settings and time / Database - query method	#SellerInfo in World State	Number of queries	Time to perform all queries
CouchDB-JSON *GetQueryResult*	1	100	1.3593487s
CouchDB-JSON *GetQueryResult*	2000	100	1.1544551s
CouchDB-*GetState*	1	100	233.0843ms
CouchDB-*GetState*	2000	100	220.8931ms
LevelDB-*GetState*	1	100	235.5689ms
LevelDB-*GetState*	2000	100	212.9267ms

Table 3 – Time and settings to fetch *SellerInfo* by different methods

7.1.1.3 Querying sorted buy/sell bids to perform the auction

During the auction, the validated *BuyBids* and the *SellBids* must be sorted in the, respectively, descending order and ascending order. CouchDB JSON queries provide a mechanism to request the sorted list of a struct, requiring the declaration about what field should determine the order. Code 7.9 displays an example of this type of query, showing a query to fetch the *SellBids* in the ascending ("asc") order and the *BuyBids* in the descending ("desc") order, based on the field *priceperkwh*.

Another possible form to obtain the sorted lists is presented in Code 7.10. The *SellBids* and validated *BuyBids* are fetched by their partial key and sorted using a *golang sort* library during the chaincode execution. The *sort.Slice()* function calls quicksort.

Code 7.9 – Querying price-sorted SellBids and validated BuyBids to CouchDB through a JSON query

```
//get SellBids
queryString := fmt.Sprintf(‘{"selector":{"issellbid":true}, "sort":
   [{"priceperkwh": "asc"}]}‘)
sellBidsIterator, err := stub.GetQueryResult(queryString)

//get VALIDATED BuyBids
queryString = fmt.Sprintf(‘{"selector":{"validated":true}, "sort":
   [{"priceperkwh": "desc"}]}‘)
buyBidsIterator, err := stub.GetQueryResult(queryString)
```

Code 7.10 – Querying SellBids and validated BuyBids using the function *GetStateByPartialCompositeKey()* and sorting them. Possible with both CouchDB and LevelDB

```
//get SellBids
objectType := "SellBid"
sellBidsIterator, err := stub.GetStateByPartialCompositeKey(objectType,
   []string{})
//get VALIDATED BuyBids
objectType = "BuyBid"
buyBidsIterator, err := stub.GetStateByPartialCompositeKey(objectType,
   []string{"true"})

...
// Sorting part (Quicksort)
//sort SellBids in ASCENDING order
sort.Slice(sellBids[:], func(i, j int) bool {
```

```
    return sellBids[i].PricePerKWH < sellBids[j].PricePerKWH
})
//sort BuyBids in DESCENDING order
sort.Slice(buyBids[:], func(i, j int) bool {
    return buyBids[i].PricePerKWH > buyBids[j].PricePerKWH
})
```

We measured the times to perform a single auction and a hundred auctions sequence for the solutions presented in Codes 7.9 and 7.10 - exhibited in Table 4. We captured the single auction measure to have a clean time reference since the 100 auction sequence was performed in a single transaction, which we considered could suffer some optimization due to the query repetition.

Settings and time Database - query method	Number of SellBids	Number of BuyBids	Number of auctions	Time to perform the auction	Avarage time/auction
CouchDB-JSON *GetQueryResult* sorted by CouchDB	5000	1000	1	22.36s	22.36s
CouchDB-JSON *GetQueryResult* sorted by CouchDB	5000	1000	100	39m52.44s	23.92s
CouchDB-*GetStateByPartialCompositeKey* sorted in chaincode	5000	1000	1	8.65s	8.65s
CouchDB-*GetStateByPartialCompositeKey* sorted in chaincode	5000	1000	100	12m20.06s	7.4s
LevelDB-*GetStateByPartialCompositeKey* sorted in chaincode	5000	1000	1	2.56s	2.56s
LevelDB-*GetStateByPartialCompositeKey* sorted in chaincode	5000	1000	100	3m26.04s	2.06s

Table 4 – Time and settings to perform auctions, executing different methods to fetch sorted *SellBids* and sorted validated *BuyBids*

7.2 EXPERIMENT RESULTS

This section presents and discusses our main 3-phase experiments with our proposal's implementation. With the experiment metrics we analyze the transaction throughput, data generation rate, and estimated deployment costs. Then, the model's viability is addressed, followed by a comparison with the related work solutions.

7.2.1 Phase 1 experiment results

The first phase of the experiments ran with all sensors participating in all energy validations. It raised our awareness about limiting the number of sensors, as discussed in Section 6.2.7. Some configurations and results of the first phase are presented in Table 5. Each row represents a configuration that failed in an experiment round. The first column contains which entity - application, orderer, or peer - indicated failure, with the AWS instance name highlighted in red. An instance upgrade happened after every failure, usually related to not properly supporting the experiment processing demands.

The rightmost columns of Table 5 present the numbers of sellers, sensors, and buyers for each round that presented a failure. We had to upgrade the instances to continuously increase the network participant capacity and go from 600 to 3000 sellers, sensors, and buyers.

Table 5 last line shows a very specific failure regarding the **peer endorsing concurrency limit**, which is set by default to **2500** concurrent endorsing requests. In the Phase 2 experiments described in Section 7.2.2, we increase this limit and avoid such failure, provided that the peer instance has the required computing power.

Settings Who failed	Round number	App instance	Orderer instance	Peer instance	Number of sellers	Number of sensors	Number of buyers
application	1	t4g.micro	t4g.micro	t4g.micro	200	200	200
peer	2	t4g.xlarge	t4g.micro	t4g.micro	400	400	400
orderer	3	t4g.xlarge	t4g.micro	t4g.xlarge	600	600	600
peer	4	t4g.xlarge	t4g.xlarge	t4g.xlarge	600	600	600
application	5	t4g.xlarge	t4g.xlarge	t4g.2xlarge	600	600	600
orderer	6	t4g.2xlarge	t4g.xlarge	t4g.2xlarge	800	800	800
peer (concurrency limit)	7	t4g.xlarge	t4g.2xlarge	t4g.2xlarge	1000	1000	1000

Table 5 – Configurations that lead to failure in Phase 1

7.2.2 Phase 2 experiment results

Table 6 presents the results of the Phase 2 experiment. We could scale the network capacity from 2000 sellers, 2000 sensors, and 2000 buyers, in the first round, to 3500 sellers, 3500 sensors, and 3500 buyers in the last one. The orderer had to be upgraded up to *c6g.4xlarge* AWS instance, with 8 cores, 32 GiB, and 10 Gbps network capacity. The peer had to be scaled to the *c6g.8xlarge* instance, with 16 cores, 64 GiB of RAM, and up to 25 Gbps network.

Only **5** sensors were selected to validate energy generation claims in this phase, different from Phase 1. This was probably the main reason for the increase in the network participant capacity. In round 4, the chaincode had a memory limit failure, as demonstrated by Figures 22 and 23.

Both figures are graphs of chaincode container instantaneous memory usage in GiB. In the failure round, corresponding to Figure 22, the chaincode memory abruptly increased to *1.75* GiB and then went negative - indicating container failure. The memory allocated for the chaincode was 2 GiB.

After the chaincode container memory was upgraded to 4 GiB, the chaincode presented a memory usage of a little over 2 GiB, in Figure 23, indicating that the chaincode crashed before due to lack of available memory. We even allocated 12 GiB to the chaincode to prevent this failure from happening in the following experiment rounds.

The last three experiment rounds demonstrate the difficulty of increasing sellers, sensors, and buyers. Despite three consecutive upgrades, the logs always presented some type of failure indication regarding network capacity. These results made us end Phase 2 and start Phase 3.

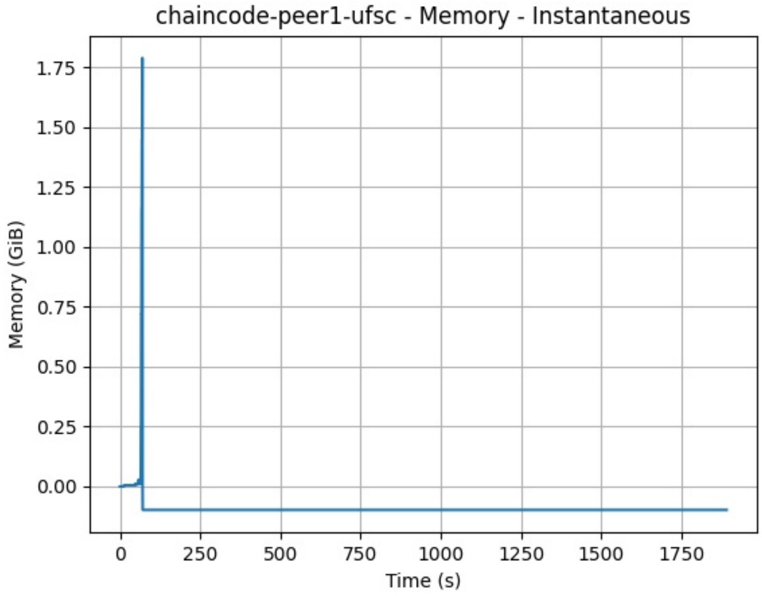
Settings Who failed	Round number	App instance	Orderer instance	Peer instance	Number of sellers	Number of sensors	Number of buyers
application	1	t4g.2xlarge	t4g.2xlarge	t4g.2xlarge	2000	2000	2000
orderer	2	c6g.4xlarge	t4g.2xlarge	t4g.2xlarge	2000	2000	2000
peer	3	c6g.4xlarge	c6g.4xlarge	t4g.2xlarge	2700	2700	2700
peer (chaincode mem limit 2 GiB)	4	c6g.4xlarge	c6g.4xlarge	c6g.4xlarge	3000	3000	3000
application/peer	5	c6g.4xlarge	c6g.4xlarge	c6g.4xlarge	3500	3500	3500
application	6	c6g.8xlarge	c6g.4xlarge	c6g.8xlarge	3500	3500	3500
orderer	7	c6g.16xlarge	c6g.4xlarge	c6g.8xlarge	3500	3500	3500

Table 6 – Configurations that lead to failure in Phase 2

7.2.3 Phase 3 experiment results

The two previous experiment phases had the default maximum block interval of **2s**, with at most **500** transactions in a block or the maximum block preferred size of **2MB**. The last three lines of Table 6 indicate difficulties in scaling beyond 3500 sellers, 3500 sensors, and 3500 buyers. We tested different block intervals, maximum transaction numbers, and maximum preferred block sizes in this phase, expecting to increase the scalability limits achieved in Phase 2.

Figure 22 – Chaincode memory data in failure round (round 4)



Designed by the author

We considered the average transaction size of 4 Kilobyte (KB) to keep always a *Block preferred size* with the capacity to fit *Block transaction limit* transactions. Otherwise, the blocks' creation would be triggered by the surpass of the *Block preferred size* and would never group *Block transaction limit* transactions.

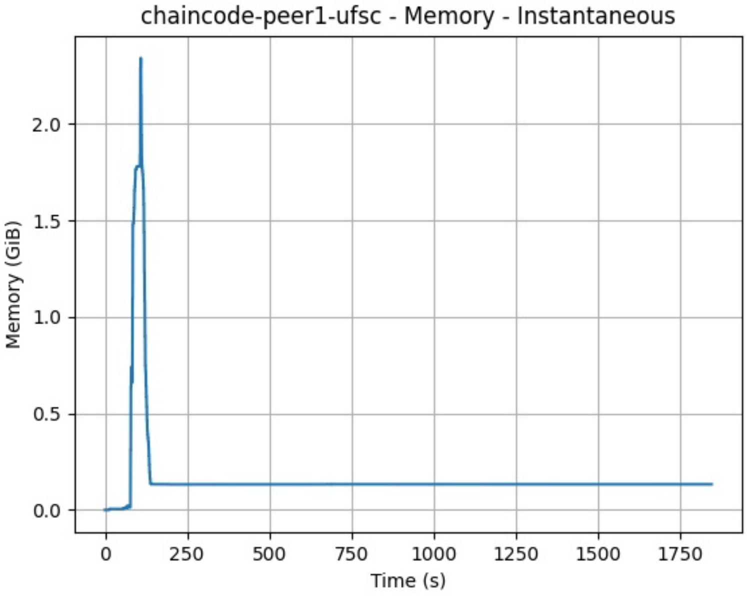
Round **1** of this phase, in Table 7, presented failures in the application's instance, indicating resource exhaustion. Thus, after round 1, we added a second application instance to divide the sensors, sellers, and buyers simulation load. For example, instead of making a single application instance simulate 3500 buyers, two instances simulate 1750 buyers each.

Unlike the previous phases, in this one, we decided to include the successful experiment rounds to explicit the network configurations that worked properly. In round 2, all logs indicated a successful execution when dealing with 3500 sensors, 3500 sellers, and 3500 buyers simultaneously.

In the third round, we tried to increase the quantity of each entity type to 5000 and failed, but round 4 made clear that the *Block transaction limit* should be increased from 2000 to 10000 maximum transactions in a single block to run without errors. After performing tests with 6000 sellers, 6000 sensors, and 6000 buyers in round 5, the chaincode container ran out of memory, and we allocated 12 GiB for the following rounds.

Rounds 6 through 8 failed due to excessive reporting of orderer timeout in the

Figure 23 – Chaincode memory usage post increase



Designed by the author

applications' logs, which was **60 seconds**. This means that after sending transactions to the orderer, the applications repeatedly did not receive a response after 60 seconds. Thus, the Hyperledger Fabric network could not absorb such high throughput with the configurations displayed in Table 7.

In round 7, we upgraded the orderer instance to a *c6g.16xlarge* AWS instance and increased the *Block transaction limit* to 20000. After it failed, our final experiment round had a 20 second *Block interval* and 30000 maximum messages per block. However, the final round also presented excessive orderer timeouts and was classified as a failure.

It is important to emphasize that if the applications' orderer timeout limit was set to a value greater than **60 seconds**, rounds like the 7th would probably run without error indications. Still, that was **our** criteria for an acceptable orderer timeout, even though we recognize that it depends on the application scenario.

.

Settings Who failed	Round number	App instance	Orderer instance	Peer instance	Number of sellers	Number of sensors	Number of buyers	Block interval	Block transaction limit	Block preferred size
application (add new app instance)	1	c6g.16xlarge	c6g.8xlarge	c6g.8xlarge	3500	3500	3500	10s	2000	60 MB
success	2	c6g.16xlarge	c6g.8xlarge	c6g.8xlarge	3500	3500	3500	10s	2000	60 MB
orderer (Low block transaction limit)	3	c6g.16xlarge	c6g.8xlarge	c6g.8xlarge	5000	5000	5000	10s	2000	60 MB
success	4	c6g.16xlarge	c6g.8xlarge	c6g.8xlarge	**5000**	**5000**	**5000**	10s	10000	60 MB
peer (chaincode mem limit 8 GiB)	5	c6g.16xlarge	c6g.8xlarge	c6g.8xlarge	6000	6000	6000	10s	10000	60 MB
orderer	6	c6g.16xlarge	c6g.8xlarge	c6g.8xlarge	6000	6000	6000	10s	10000	60 MB
orderer	7	c6g.16xlarge	c6g.16xlarge	c6g.8xlarge	6000	6000	6000	10s	20000	90 MB
orderer	8	c6g.16xlarge	c6g.16xlarge	c6g.8xlarge	6000	6000	6000	20s	30000	150 MB

Table 7 – Round configurations in Phase 3

Observation: We identified that the peer auction events were not sent to the applications in this phase. This prevented the auction transactions from being called, and that sell bids were matched to buy bids. However, the auction transactions represent an irrelevant share of all transactions in a throughput perspective - lower than 0.02%. The cause of such failure was not identified. It could have been caused by the higher peer demand or by the larger batch interval. Thus, regardless of this small failure, the orderer could handle the transaction throughput indicated as “success” by Table 7.

7.3 DATA GENERATION RATE

Every experiment round in Phases 1, 2, and 3 measured the orderer’s and peer’s file system size to identify the proportion of data generation rate. The file systems were measured at two distinct moments. First, at the round beginning before any application issued any transaction to the network. Second, after all the applications published all their transactions.

Tables 8 and 9 present, respectively, the orderer’s and peer’s file system sizes corresponding to the experiment rounds performed in Phase 3. The file systems’ sizes presented in Tables 8 and 9 are proportional to the number of sellers, sensors, and buyers. Tables 10 and 11 present the data generated in a successful round of experiment Phase 2.

Settings Phase 3 round	Number of sellers	Number of sensors	Number of buyers	Initial size	Final size	Data generated
2 (success)	3500	3500	3500	289 MB	897 MB	608 MB
4 (success)	5000	5000	5000	289 MB	1178 MB	889 MB

Table 8 – Orderer data generation in Phase 3

.

Settings Phase 3 round	Number of sellers	Number of sensors	Number of buyers	Initial size	Final size	Data generated
2 (success)	3500	3500	3500	53 MB	597 MB	544 MB
4 (success)	5000	5000	5000	53 MB	833 MB	780 MB

Table 9 – Peer data generation in Phase 3

.

Settings Phase 2 round	Number of sellers	Number of sensors	Number of buyers	Initial size	Final size	Data generated
success	3000	3000	3000	1564 MB	2146 MB	582 MB

Table 10 – Orderer data generation in successful Phase 2 round

.

Settings / Phase 2 round	Number of sellers	Number of sensors	Number of buyers	Initial size	Final size	Data generated
success	3000	3000	3000	1318 MB	1928 MB	610 MB

Table 11 – Peer data generation in successful Phase 2 round

Different from Phase 3, in Phase 2, the auction happened periodically, generating proportionally more data. This becomes evident comparing the data generation in round **2** of Phase 3 (Tables 8 and 9) with the **successful** round of Phase 2 (Tables 10 and 11). Even with fewer network participants, the round in Phase 2 generated just 26 MB less in the orderer and 76 MB more in the peer.

Therefore, to achieve better precision, we utilize the Phase 2 data generation and execution time to make estimates for longer periods. As stated in Code 6.7, each sensor published **20** transactions plus one declaring itself active. Each seller performed **5** energy generation and **5** sell bid publication transactions, while the buyers submitted **10** buy bids and **10** buy bid validation transactions. **Twenty-nine** auctions were executed in the successful Phase 2 round.

Considering that the round took 27 minutes to complete and generate the data quantity presented by Tables 10 and 11, we could estimate how much storage would be required to support the network execution with the same configurations for longer periods. The estimates for the data generated in a day, month, and year are presented in Table 12. Such estimates were calculated lineary, as the applications and the chaincode generate data linearly.

The estimated transaction numbers are also presented in the Table’s 12 rightmost column, excluding the first row, since it consists of a measure and not an estimate. Even though the Ethereum blockchain has a quite different concept, it serves as a good anchor for comparing data generation and transaction quantity. In the first half of 2021, the Ethereum main network grew approximately 1 Gigabyte (GB) per day, with around 1.2 million transactions and the rate of 1 GB per million transactions.

Generation period	Orderer generated	Peer generated	Orderer + Peer	Transactions
27 minutes	582 MB	610 MB	1192 MB	150 K
1 day	31 GB	32.5 GB	63.5 GB	8 M
1 month	930 GB	975 GB	1.8 TB	240 M
1 year	11 TB	11.7 TB	22.7 TB	2.8 B

Table 12 – Data generation and transaction estimates based on the successful Phase 2 round

Meanwhile, our network generated data at the rate of 8 GB per million transactions. Only a single endorser signed the transactions on our experiments. Therefore, this rate could increase proportionally to the endorsers' quantity in other scenarios. Solutions to deal with such characteristics would enhance the proposed model and increase its adoption chance.

We consider the idea of multi-layer chains (or channels) as a possible solution. The upper-level chains could perform some digest on the lower chains' transactions and store it. The raw data could last for a specific time interval and, after that, be digested, referenced in the upper chain, and erased from the lower chain.

In our experiments' context, the lower chain is equivalent to the network proposed and implemented by this work. The upper chain could be developed by future work. This architecture fits well with an energy trading scenario that does not require high granularity for long periods. As a result, the data generation rate could be lowered.

7.4 ENERGY NETWORK BASELINE COST ANALYSIS

Evaluating the costs of a proposal is crucial for judging its feasibility. For that reason, we estimate the cost to deploy our model. Based on the *c6g.8xlarge* AWS instances costs, we estimate the funds needed to run a network equivalent to the 4^{th} round of Phase 3 in terms of execution cost. Nevertheless, since we focused our data generation analysis on the Phase 2 round, it will serve as a reference to estimate the storage costs.

The *c6g.8xlarge* instance *on-demand* costs 0.6816 USD hourly. Regarding storage, we consider the pricing for an AWS General Purpose Solid-State Drive (SSD) (gp2) Elastic Block Store (EBS), which is 0.114 USD per GB-Month. All costs are related to the Mumbai region and displayed in Table 13. In the first three rows, we assumed that the storage space was **fully** provisioned at the beginning. However, we considered that the storage increases on a month-by-month demand basis for the *year* cost row. Equation 2 expresses the formula to calculate it as a 12 term sum of the monthly cost arithmetic progression.

$$Ysc = \frac{(GBMc * Yg * 12 + GBMc * Yg) * 12}{2} \tag{2}$$

.

Execution period	Orderer/Peer single instance cost (USD)	Instances total cost (USD)	EBS cost for generated data USD	Total cost (USD)	Transactions
27 minutes	0.3	0.6	0.114	0.714	150 K
1 day	16.35	32.70	7.24	39.94	8 M
1 month	490.5	981	205.2	1186.2	240 M
1 year	5886	11772	16005	27777	2.8 B

Table 13 – Cost estimate based on round 4 of Phase 3 execution, but Phase 2 data generation and transaction rate

Based on the yearly total cost and transaction rates, our model with one peer and one orderer has a cost (USD) per transaction ratio of $9.92 * 10^{-6}$. Considering an average Ethereum transaction fee of 5 USD while disregarding the 2021 transaction fee volatility (YCHARTS, 2021), our network presents significantly lower costs. Even if compared against Ethereum's lowest historical transaction fee of 0.5 USD, the comparison holds.

Unlike our experiments, a real Hyperledger Fabric network would have more than one peer and one orderer. Presuming that the real network would have 20 peers and 20 orderers to process the same 2.8 B, the cost per transaction would be around the value of $1.94 * 10^{-4}$, which still surpasses Ethereum (this is a rough estimate without considering that the transaction size and performance would be affected with more peers and orderers).

If the solution mentioned in Section 7.3 about reducing the data generation rate is implemented, the costs could drop. Furthermore, other storage solutions like AWS S3 or EFS should be analyzed in our model's context. The EBS has a maximum capacity of 16 Tebibyte (TiB) per volume, and changing the storage tools would change network prices.

7.5 ENERGY NETWORK VIABILITY

We achieved a successful throughput of 5000 sellers, 5000 sensors, and 5000 buyers simultaneously. At these metrics, our proposed model suits a **small neighborhood**. The proportion between sensors and buyers/sellers in our experiments may differ in a real environment since much more buyers/sellers are expected than sensors.

The energy validation transactions consume a considerable amount of chaincode processing. For that reason, blockchain trading models without energy validation based on sensors' data might attain greater performance. However, this decision depends on the network architect's objectives of applying blockchain in the energy context.

We assumed that validating energy before the sale would prevent frauds and increase the trust in the energy generation type, serving as a useful feature. Buyers anonymization might be a regulators requirement to protect users according to data privacy laws, and our implementation covers it.

Our analysis' intent consists in providing **computational** and **cost** perspectives of blockchain use in energy trading. Energy engineering researchers might consider our findings and judge if blockchain fits this area due to their greater knowledge in the field.

7.6 RELATED WORK COMPARISON

The related work's proposals are heterogeneous, with different market designs, experiment complexity, and focus. However, this work contributed to the research done by their authors in many diverse aspects like privacy, scalability, experiment depth, experiment procedure, and empirical data. We compare our work's implementation and results with the related work based on their proposals, experiments, and future directions.

The authors of (PEE et al., 2019), (HUSSAIN et al., 2019), (KODALI et al., 2018) only proposed or implemented simple models regarding blockchain in energy markets, and we brought clarity to a topic that lacks experimental data. As suggested by (HUSSAIN et al., 2019), our work did not use PoW consensus. Even though our model was implemented with a single chain, different from (LU et al., 2019) and (PEE et al., 2019), we consider a multi-chain approach for dealing with the data quantity due to our model's data generation rate.

We implemented a solution with pseudonymity, as suggested by (AHL et al., 2020) future directions, and with off-chain payment to enable the pseudonymity, guaranteeing the funds, as mentioned by (LU et al., 2019). We did not implement day-ahead and real-time market similar to (WANG, S. et al., 2019) because we focused on validating the energy before exchanging.

The most interesting comparison is with the thesis of (BLOM, 2018). They implemented an energy market with an Ethereum smart contract using PoW consensus, which consumes more power than the alternatives and should be avoided in a clean energy context. Furthermore, the authors implemented a model with off-chain market clearing. With these characteristics, they simulated their implementation with 600 entities transacting simultaneously.

Our implementation keeps the market clearing in the chain and adds the energy validation process based on sensors' data. Despite these smart contract processing increases, we could handle 15000 entities transacting simultaneously. Unlike (BLOM, 2018) - with day-ahead, real-time, and load curtailment markets - our model only lets energy generated in the past be exchanged. Therefore, our exchange options implied a lower chaincode complexity in this aspect, perhaps helping with the higher throughput.

In terms of cost, the (BLOM, 2018) required 8 billion Ethereum *gas* for a network with 600 entities and a 15-minute market clearing. Considering a *gas* price of 15.8 Gwei and an Ether price of 2031 USD, their proposal would cost around 250 000 USD per day if it ran in the Ethereum main net. Meanwhile, if the yearly costs of our experiment are divided by the days in a year, our model costs can be estimated to 76 USD per day for each pair of peers and orderers.

Accomplishing the (BLOM, 2018) future directions, our model could achieve higher scalability and better privacy. It was tested with real computers, even though we did not use real smart meters. The data in Table 13 points to the best throughput of 93 transactions per second by our implementation. (DORRI et al., 2019) had a throughput of 6 transactions per minute, while (BLOM, 2018) mentions the need for 52 transactions per second throughput, both in an Ethereum energy trading scenario.

While (HUANG et al., 2019) focused on the IoT part of blockchain energy trading, we did not take our model and experiments that far. Future work could research lighter interactions between restricted IoT devices and blockchain networks. The Hyperledger communication stack, including the Fabric SDKs, seems too heavy for lightweight devices.

8 CONCLUSION

8.1 CONCLUSION AND CONTRIBUTIONS

In this work, we proposed, implemented, and analyzed a blockchain-based energy trading scheme with validation using IoT sensors data. The negotiated energy must have been generated in the past and has a significant guarantee of origin. This is accomplished by a decentralized multi-organizational chaincode which requires a minimum organizations quorum to validate energy generation claims.

In our implementation, to protect buyers' energy consumption patterns, they transact with the network through a k-TAA algorithm (idemix). Even though the energy cannot be bought on-demand in our model, the buyer anonymization facilitates the implementation of a future secure real-time blockchain energy market.

We analyzed our solution's performance, scalability, and costs, considering different quantities of sensors, buyers, sellers, and different hardware configurations for peers and orderers. Some Hyperledger configurations like the peer concurrency limit, the memory allocated for the chaincode, block size, and block interval also were changed and analyzed. Considering our results, our solution fits a small neighborhood context.

Hyperledger Fabric is more efficient computationally and monetarily than the Ethereum solutions presented by the related work, based on our solution's better throughput and estimated cost. With a single peer and a single orderer, we measured a cost per transaction outstandingly lower than the one charged by the Ethereum main net.

As secondary contributions, we developed scripts that easily deploy configurable Hyperledger networks, enabling that parameters like organizations, peer quantity, orderer quantity, chaincodes are easily defined. The certificates host fields are set according to the hosts attributed by the cloud service. These scripts contribute to future work that depends on deploying a Hyperledger network on a cloud infrastructure similar to AWS.

Our Fabric modifications contribute to previous and future research. We enabled idemix in the fabric-gateway-java by implementing the required interfaces and performing small alterations on the fabric-sdk-java. The transactions, with our modifications, have priorities that are set by the chaincode function return.

It is now possible to bypass the *phantom read* checks by setting the proper chaincode function return method. Future work with Hyperledger chaincodes might take advantage of this modification to avoid that time-costly transactions are wrongfully invalidated due to the *phantom read conflicts*. Thus, the support for more complex chaincodes is increased.

At last, we analyzed the impact of the Hyperledger Fabric database type choice.

With our chaincode, Go LevelDB presented a significantly better performance than CouchDB. While CouchDB supports enhanced queries, it is quicker to retrieve data and implement the sorting in the chaincode using LevelDB. However, the more limited *key* queries with LevelDB require the proper design of data *keys*, or it might not be easy to perform attribute-based queries.

8.2 FUTURE WORK

Since knowing the precise function to calculate the maximum possible energy generated by a specific solar panel type, or wind turbine, was out of our scope, we leave it as future work for researchers in electrical engineering. The network consensus on how much a specific solar panel model can generate given environment metrics could also be designed and implemented.

Hyperledger Fabric allows the change on multiple configuration parameters, and we did not explore the full extent of them. Further analysis on enhancing performance through better Fabric configuration would add more reliability to our work. Also, there is space for experiments with more organizations, peers, orderers, sensors, buyers, which might require model modifications to handle higher transaction throughput.

In our experiments, the utility and payment companies' HTTP servers were removed to increase the reliability of the blockchain performance metrics. To fully validate our solution, new experiments, including the HTTP servers, would be required. However, some challenges come with bringing them back.

Golang HTTP servers would fit much better in terms of scalability and concurrent requests handling. Still, at the moment, only fabric-java-sdk provides idemix support, and the utility company server performs idemix signature verifications. Idemix support for the fabric-sdk-go would facilitate setting scalable HTTP servers.

All sensors are retrieved from the database as possible participants of an energy validation claim process in our implementation. Then the chaincode calculates the sensor distance to the seller and judges if the sensor will participate or not. Instead, a geospatial database could take the seller's location as query input and only return the near sensors more efficiently, as we suppose.

The SmartData provides the *confidence* and *error* fields, but we do not evaluate them in our chaincode. Future work could consider these fields and give more weight to SmartData with bigger confidence and discard the SmartData with error. Such verification would increase the energy validation reliability.

Furthermore, SmartData version 1.2 supports data from moving sensors. At the current implementation, our chaincode considers all sensors as static data sources. However, extending it to moving ones could enhance the energy validation process, but it also would require more analysis.

The energy sold through our chaincode has to be generated in the past. Yet, the following work could use our current work as a base to experiment with a futures energy market with energy delivery verification. This would bring blockchains closer to the current energy markets, as discussed in Section 2.1.2.

In our previous work (WESTPHALL et al., 2020), we analyzed Constrained Application Protocol (CoAP) and Datagram Transport Layer Security (DTLS) on an IoT gateway, both using User Datagram Protocol (UDP), which is considered more efficient for constrained devices. Meanwhile, Hyperledger Fabric communicates through gRPC, which uses Transmission Control Protocol (TCP). An examination on sensors gateways running gRPC would be relevant. Perhaps, a solution considering IoT-friendly protocols could enhance the interaction between gateways and blockchains.

In the chaincode, sellers, sensors, and companies are uniquely identified by the Base64 encoding of the certificate Distinguished Names, usually generate a string sized around 176 bytes. A smaller unique identification would promote more efficient stores in Fabric's database, considering that, as an example, every SmartData record stores the sensor's identification string.

We could not find any energy consumption per instance type in AWS EC2 documentation. The energy spent on our models' execution would be an interesting metric to decide if our model is efficient from an energetic and environmental standpoint. The clean energy amount negotiated and incentivized by the blockchain market should be worth the energy spent on executing the market's infrastructure.

The authors of (GORENFLO et al., 2019) present some adaptations on Hyperledger Fabric source code that scale up its throughput to a rate of 20000 transactions per second. Since some of our chaincode's transactions require considerably more processing than regular Hyperledger Fabric transactions, future work could run our experiments with (GORENFLO et al., 2019) adaptations to verify if the throughput would increase.

REFERENCES

ABIDIN, Aysajan; ALY, Abdelrahaman; CLEEMPUT, Sara; MUSTAFA, Mustafa A. **Secure and Privacy-Friendly Local Electricity Trading and Billing in Smart Grid**. [S.l.: s.n.], 2018. arXiv: `1801.08354 [cs.CR]`.

AHL, A.; YARIME, M.; GOTO, M.; CHOPRA, Shauhrat S.; KUMAR, Nallapaneni Manoj.; TANAKA, K.; SAGAWA, D. Exploring blockchain for the energy transition: Opportunities and challenges based on a case study in Japan. **Renewable and Sustainable Energy Reviews**, v. 117, p. 109488, 2020. ISSN 1364-0321. DOI: `https://doi.org/10.1016/j.rser.2019.109488`. Available from: `http://www.sciencedirect.com/science/article/pii/S1364032119306963`.

ALABDULLATIF, Abdullah M.; GERDING, Enrico H.; PEREZ-DIAZ, Alvaro. Market Design and Trading Strategies for Community Energy Markets with Storage and Renewable Supply. **Energies**, v. 13, n. 4, 2020. ISSN 1996-1073. DOI: `10.3390/en13040972`. Available from: `https://www.mdpi.com/1996-1073/13/4/972`.

ALAM, A.; ISLAM, M. T.; FERDOUS, A. Towards Blockchain-based Electricity Trading System and Cyber Resilient Microgrids. In: 2019 International Conference on Electrical, Computer and Communication Engineering (ECCE). [S.l.: s.n.], 2019. P. 1–5.

ALAM, Asraful; FERDOUS, Arafa; ISLAM, Mohammad. Towards Blockchain-based Electricity Trading System and Cyber Resilient Microgrids. In: DOI: `10.1109/ECACE.2019.8679442`.

ANDONI, Merlinda; ROBU, Valentin; FLYNN, David; ABRAM, Simone; GEACH, Dale; JENKINS, David; MCCALLUM, Peter; PEACOCK, Andrew. Blockchain technology in the energy sector: A systematic review of challenges and opportunities. **Renewable and Sustainable Energy Reviews**, v. 100, p. 143–174, 2019. ISSN 1364-0321. DOI: `https://doi.org/10.1016/j.rser.2018.10.014`. Available from: `http://www.sciencedirect.com/science/article/pii/S1364032118307184`.

ANDROULAKI, Elli et al. Hyperledger fabric. **Proceedings of the Thirteenth EuroSys Conference**, ACM, Apr. 2018. DOI: `10.1145/3190508.3190538`. Available from: `http://dx.doi.org/10.1145/3190508.3190538`.

AU, Man Ho; SUSILO, Willy; MU, Yi. Constant-Size Dynamic k-TAA. In: DE PRISCO, Roberto; YUNG, Moti (Eds.). **Security and Cryptography for Networks**. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006. P. 111–125.

AUTHORS, Amazon. **User Guide for Linux Instances**. [S.l.: s.n.], 2021. Online; accessed April, 2021. Available from: `https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/concepts.html`.

AVANCINI, Danielly B.; RODRIGUES, Joel J.P.C.; MARTINS, Simion G.B.; RABÊLO, Ricardo A.L.; AL-MUHTADI, Jalal; SOLIC, Petar. Energy meters evolution in smart grids: A review. **Journal of Cleaner Production**, v. 217, p. 702–715, 2019. ISSN 0959-6526. DOI: `https://doi.org/10.1016/j.jclepro.2019.01.229`. Available from: `http://www.sciencedirect.com/science/article/pii/S0959652619302501`.

BEN-KIKI, O.; EVANS, C; DÖT NET, I. **YAML: YAML Ain't Markup Language**. [S.l.: s.n.], 2020. Online; accessed December, 2020. Available from: `https://yaml.org/`.

BERNAL BERNABE, J.; CANOVAS, J. L.; HERNANDEZ-RAMOS, J. L.; TORRES MORENO, R.; SKARMETA, A. Privacy-Preserving Solutions for Blockchain: Review and Challenges. **IEEE Access**, v. 7, p. 164908–164940, 2019.

BLOM, Frederik. **A Feasibility Study of Blockchain Technology As Local Energy Market Infrastructure**. 2018. Available from: `https://ntnuopen.ntnu.no/ntnu-xmlui/handle/11250/2502356`.

BLUMMER, Tamas et al. **An Introduction to Hyperledger**. [S.l.: s.n.], 2018. Online; accessed May, 2020. Available from: `https://www.hyperledger.org/wp-content/uploads/2018/08/HL_Whitepaper_IntroductiontoHyperledger.pdf`.

CALDARELLI, Giulio. Real-world blockchain applications under the lens of the oracle problem. A systematic literature review. In: 2020 IEEE International Conference on Technology Management, Operations and Decisions (ICTMOD). [S.l.: s.n.], 2020. P. 1–6. DOI: `10.1109/ICTMOD49425.2020.9380598`.

CAMENISCH, Jan; LYSYANSKAYA, Anna. Signature Schemes and Anonymous Credentials from Bilinear Maps. In: FRANKLIN, Matt (Ed.). **Advances in Cryptology – CRYPTO 2004**. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004. P. 56–72.

CAO, Ling; WAN, Zheyi. Anonymous scheme for blockchain atomic swap based on zero-knowledge proof. In: 2020 IEEE International Conference on Artificial Intelligence and Computer Applications (ICAICA). [S.l.: s.n.], 2020. P. 371–374. DOI: `10.1109/ICAICA50127.2020.9181875`.

COMMITTEE, Microprocessor Standards. IEEE Standard for Floating-Point Arithmetic. **IEEE Std 754-2019 (Revision of IEEE 754-2008)**, p. 1–84, 2019. DOI: `10.1109/IEEESTD.2019.8766229`.

LE-DANG, Q.; LE-NGOC, T. Scalable Blockchain-based Architecture for Massive IoT Reconfiguration. In: 2019 IEEE Canadian Conference of Electrical and Computer Engineering (CCECE). [S.l.: s.n.], 2019. P. 1–4.

DATA, The Secret Lives of. **Raft Understandable Distributed Consensus**. [S.l.: s.n.], 2020. Online; accessed October, 2020. Available from: `http://thesecretlivesofdata.com/raft/`.

DORRI, A.; HILL, A.; KANHERE, S.; JURDAK, R.; LUO, F.; DONG, Z. Y. Peer-to-Peer EnergyTrade: A Distributed Private Energy Trading Platform. In: 2019 IEEE International Conference on Blockchain and Cryptocurrency (ICBC). [S.l.: s.n.], 2019. P. 61–64.

ENERGY, Leaders in. **Utilities of the future**. [S.l.: s.n.]. Online; accessed May, 2020. Available from: `https://leadersinenergy.org/wp-content/uploads/2018/10/Caldwell-2018-Utilities-of-the-Future-Slide-Deck_FINAL-FOR-DISTRIBUTION.pdf`.

FOSCHINI, L.; GAVAGNA, A.; MARTUSCELLI, G.; MONTANARI, R. Hyperledger Fabric Blockchain: Chaincode Performance Analysis. In: ICC 2020 - 2020 IEEE International Conference on Communications (ICC). [S.l.: s.n.], 2020. P. 1–6. DOI: `10.1109/ICC40277.2020.9149080`.

FRANCIA, Steve. **Viper github**. [S.l.: s.n.], 2020. Online; accessed December, 2020. Available from: `https://github.com/spf13/viper`.

GOEL, Seep; SINGH, Abhishek; GARG, Rachit; VERMA, Mudit; JAYACHANDRAN, Praveen. Resource Fairness and Prioritization of Transactions in Permissioned Blockchain Systems (Industry Track). In: (Middleware ’18), p. 46–53.

DOI: `10.1145/3284028.3284035`. Available from:
`https://doi.org/10.1145/3284028.3284035`.

GORENFLO, Christian; LEE, Stephen; GOLAB, Lukasz; KESHAV, S. **FastFabric: Scaling Hyperledger Fabric to 20,000 Transactions per Second**. [S.l.: s.n.], 2019. arXiv: `1901.00910 [cs.DC]`.

HUANG, Z.; SUANKAEWMANEE, K.; KANG, J.; NIYATO, D.; SIN, N. P. Development of Reliable Wireless Communication System for Secure Blockchain-based Energy Trading. In: 2019 16th International Joint Conference on Computer Science and Software Engineering (JCSSE). [S.l.: s.n.], 2019. P. 126–130.

HUSSAIN, S. M. S.; FAROOQ, S. M.; USTUN, T. S. Implementation of Blockchain technology for Energy Trading with Smart Meters. In: 2019 Innovations in Power and Advanced Computing Technologies (i-PACT). [S.l.: s.n.], 2019. P. 1–5.

INC., Docker. **Welcome to Docker Hub**. [S.l.: s.n.], 2020. Online; accessed December, 2020. Available from: `https://hub.docker.com/`.

JEON, J. M.; HONG, C. S. A Study on Utilization of Hybrid Blockchain for Energy Sharing in Micro-Grid. In: 2019 20th Asia-Pacific Network Operations and Management Symposium (APNOMS). [S.l.: s.n.], 2019. P. 1–4.

JOGUNOLA, O.; HAMMOUDEH, M.; ADEBISI, B.; ANOH, K. Demonstrating Blockchain-Enabled Peer-to-Peer Energy Trading and Sharing. In: 2019 IEEE Canadian Conference of Electrical and Computer Engineering (CCECE). [S.l.: s.n.], 2019. P. 1–4.

JOHANNING, S.; BRUCKNER, T. Blockchain-based Peer-to-Peer Energy Trade: A Critical Review of Disruptive Potential. In: 2019 16th International Conference on the European Energy Market (EEM). [S.l.: s.n.], 2019. P. 1–8.

KAMAL, M.; TARIQ, M. Light-Weight Security and Blockchain Based Provenance for Advanced Metering Infrastructure. **IEEE Access**, v. 7, p. 87345–87356, 2019.

KANG, E. S.; PEE, S. J.; SONG, J. G.; JANG, J. W. A Blockchain-Based Energy Trading Platform for Smart Homes in a Microgrid. In: 2018 3rd International Conference on Computer and Communication Systems (ICCCS). [S.l.: s.n.], 2018. P. 472–476.

KAUR, Amanpreet; NONNENMACHER, Lukas; PEDRO, Hugo T.C.; COIMBRA, Carlos F.M. Benefits of solar forecasting for energy imbalance markets. **Renewable Energy**, v. 86, p. 819–830, 2016. ISSN 0960-1481. DOI: `https://doi.org/10.1016/j.renene.2015.09.011`. Available from: `http://www.sciencedirect.com/science/article/pii/S0960148115302901`.

KODALI, R. K.; YERROJU, S.; YOGI, B. Y. K. Blockchain Based Energy Trading. In: TENCON 2018 - 2018 IEEE Region 10 Conference. [S.l.: s.n.], 2018. P. 1778–1783.

LISHA. **EPOS 2 User Guide**. [S.l.: s.n.], 2020a. Online; accessed February, 2021. Available from: `https://epos.lisha.ufsc.br/IoT+Platform#SmartData`.

LISHA. **EPOS 2 User Guide**. [S.l.: s.n.], 2020b. Online; accessed February, 2021. Available from: `https://epos.lisha.ufsc.br/EPOS+2+User+Guide#SmartData`.

LU, X.; GUAN, Z.; ZHOU, X.; DU, X.; WU, L.; GUIZANI, M. A Secure and Efficient Renewable Energy Trading Scheme Based on Blockchain in Smart Grid. In: 2019 IEEE 21st International Conference on High Performance Computing and Communications; IEEE 17th International Conference on Smart City; IEEE 5th International Conference on Data Science and Systems (HPCC/SmartCity/DSS). [S.l.: s.n.], 2019. P. 1839–1844.

MEDEIROS FRÖHLICH, A. A. SmartData: an IoT-ready API for sensor networks. **International Journal of Sensor Networks (IJSNET)**, v. 28, p. 202–210, 2018. ISSN 0959-6526. DOI: `https://doi.org/10.1504/IJSNET.2018.096264`. Available from: `https://www.inderscienceonline.com/doi/abs/10.1504/IJSNET.2018.096264`.

MENGELKAMP, Esther; GÄRTTNER, Johannes; ROCK, Kerstin; KESSLER, Scott; ORSINI, Lawrence; WEINHARDT, Christof. Designing microgrid energy markets: A case study: The Brooklyn Microgrid. **Applied Energy**, v. 210, p. 870–880, 2018. ISSN 0306-2619. DOI: `https://doi.org/10.1016/j.apenergy.2017.06.054`. Available from: `http://www.sciencedirect.com/science/article/pii/S030626191730805X`.

MOON, Sung Jun; PARK, In Hwan; LEE, Beom Suk; JU WOOK, Jang. A Hyperledger-based P2P Energy Trading Scheme using Cloud Computing with Low Capabillity Devices. In: 2019 IEEE International Conference on Smart Cloud (SmartCloud). [S.l.: s.n.], 2019. P. 190–192. DOI: `10.1109/SmartCloud.2019.00039`.

ORSINI, L.; KEMENADE, C.; WEB, M.; HEITMANN, P. Transactive Energy, 2019. Unavailable; accessed May, 2020. Available from: `https://exergy.energy/wp-content/uploads/2019/03/TransactiveEnergy-PolicyPaper-v2-2.pdf`.

PEE, S. J.; KANG, E. S.; SONG, J. G.; JANG, J. W. Blockchain based smart energy trading platform using smart contract. In: 2019 International Conference on Artificial Intelligence in Information and Communication (ICAIIC). [S.l.: s.n.], 2019. P. 322–325.

PINSON, Pierre. **Renewables in Electricity Markets**. [S.l.]: Technical University of Denmark, 2018. Online; accessed May, 2020. Available from: `http://pierrepinson.com/index.php/teaching/`.

PODGORELEC, B.; KERŠIČ, V.; TURKANOVIĆ, M. Analysis of Fault Tolerance in Permissioned Blockchain Networks. In: 2019 XXVII International Conference on Information, Communication and Automation Technologies (ICAT). [S.l.: s.n.], 2019. P. 1–6. DOI: `10.1109/ICAT47117.2019.8938836`.

POOL, Nord. **See what Nord Pool can offer you**. [S.l.: s.n.], 2020. Online; accessed May, 2020. Available from: `https://www.nordpoolgroup.com/`.

PROJECT, Hyperledger. **Gossip data dissemination protocol**. [S.l.: s.n.], 2020. Online; accessed March, 2021. Available from: `https://hyperledger-fabric.readthedocs.io/en/release-2.3/gossip.html`.

RAHOUTI, M.; XIONG, K.; GHANI, N. Bitcoin Concepts, Threats, and Machine-Learning Security Solutions. **IEEE Access**, v. 6, p. 67189–67205, 2018.

TEAM, Fabric. **Client Identity Chaincode Library**. [S.l.: s.n.], 2021. Online; accessed February, 2021. Available from: `https://github.com/hyperledger/fabric-chaincode-go/tree/master/pkg/cid`.

TEAM, Fabric. **Hyperledger Fabric Gateway SDK for Java**. [S.l.: s.n.], 2020a. Online; accessed December, 2020. Available from: `https://github.com/hyperledger/fabric-gateway-java`.

TEAM, Fabric. **Hyperledger Fabric SDK for Java**. [S.l.: s.n.], 2020b. Online; accessed December, 2020. Available from: `https://github.com/hyperledger/fabric-sdk-java`.

TEAM, Fabric Doc. **A Blockchain Platform for the Enterprise**. [S.l.: s.n.], 2020a. Online; accessed September, 2020. Available from: `https://hyperledger-fabric.readthedocs.io/en/release-2.3/`.

TEAM, Fabric Doc. **CouchDB as the State Database**. [S.l.: s.n.], 2020b. Online; accessed February, 2021. Available from: `https://hyperledger-fabric.readthedocs.io/en/latest/couchdb_as_state_database.html`.

TEAM, Fabric Doc. **Read-Write set semantics**. [S.l.: s.n.], 2020c. Online; accessed April, 2021. Available from: `https://hyperledger-fabric.readthedocs.io/en/release-2.3/readwrite.html`.

TEAM, Fabric Doc. **The Ordering Service**. [S.l.: s.n.], 2020d. Online; accessed September, 2020. Available from: `https://hyperledger-fabric.readthedocs.io/en/release-2.3/orderer/ordering_service.html`.

TRÓN, Viktor; JAMESON, Hydson. **Ethereum Homestead Documentation**. [S.l.: s.n.], 2020. Online; accessed May, 2020. Available from: `https://ethdocs.org/en/latest/`.

VUJIČIĆ, D.; JAGODIĆ, D.; RANĐIĆ, S. Blockchain technology, bitcoin, and Ethereum: A brief overview. In: 2018 17th International Symposium INFOTEH-JAHORINA (INFOTEH). [S.l.: s.n.], 2018. P. 1–6.

WANG, Naiyu; ZHOU, Xiao; LU, Xin; GUAN, Zhitao; WU, Longfei; DU, Xiaojiang; GUIZANI, Mohsen. When Energy Trading Meets Blockchain in Electrical Power System: The State of the Art. **Applied Sciences**, v. 9, p. 1561, Apr. 2019. DOI: `10.3390/app9081561`.

WANG, S.; TAHA, A. F.; WANG, J.; KVATERNIK, K.; HAHN, A. Energy Crowdsourcing and Peer-to-Peer Energy Trading in Blockchain-Enabled Smart Grids. **IEEE Transactions on Systems, Man, and Cybernetics: Systems**, v. 49, n. 8, p. 1612–1623, 2019.

WESTPHALL, Johann. **Energy Network - Developed in Hyperledger Fabric**. [S.l.: s.n.], 2021. Online; accessed April, 2021. Available from: `https://github.com/johannww/EnergyNetwork`.

WESTPHALL, Johann; LOFFI, Leandro; WESTPHALL, Carla Merkle; EVERSON MARTINA, Jean. CoAP + DTLS: A Comprehensive Overview of

Cryptographic Performance on an IOT Scenario. In: 2020 IEEE Sensors Applications Symposium (SAS). [S.l.: s.n.], 2020. P. 1–6. DOI: `10.1109/SAS48726.2020.9220033`.

XU, X.; WANG, X.; LI, Z.; YU, H.; SUN, G.; MAHARJAN, S.; ZHANG, Y. Mitigating Conflicting Transactions in Hyperledger Fabric Permissioned Blockchain for Delay-sensitive IoT Applications. **IEEE Internet of Things Journal**, p. 1–1, 2021. DOI: `10.1109/JIOT.2021.3050244`.

YAGA, Dylan; MELL, Peter; ROBY, Nik; SCARFONE, Karen. Blockchain technology overview. National Institute of Standards and Technology, Oct. 2018. DOI: `10.6028/nist.ir.8202`. Available from: `https://nvlpubs.nist.gov/nistpubs/ir/2018/NIST.IR.8202.pdf`.

YCHARTS. **Ethereum Average Transaction Fee**. [S.l.: s.n.], 2021. Online; accessed July, 2021. Available from: `https://ycharts.com/indicators/ethereum_average_transaction_fee`.

ZIEGLER, M. H.; GROBMANN, M.; KRIEGER, U. R. Integration of Fog Computing and Blockchain Technology Using the Plasma Framework. In: 2019 IEEE International Conference on Blockchain and Cryptocurrency (ICBC). [S.l.: s.n.], 2019. P. 120–123.

APPENDIX A – ENERGY VALIDATION CODE

Code A.1 – Solar energy validation function based on Candela *SmartData*

```
func getMaxPossibleGeneratedSolarEnergyInInterval(stub
   shim.ChaincodeStubInterface, nearTrustedSensorsSmartData
   *[]st.SmartData, solarPanelsNumber uint64) float64 {
  sensorSmartDataQuantity := make(map[string]int)
  sensorSmartDataSum := make(map[string]float64)
  sensorSmartDataMean := make(map[string]float64)

  for _, smartData := range *nearTrustedSensorsSmartData {
    si := smartData.Unit >> 31
    num := smartData.Unit >> 29 & 3
    mod := smartData.Unit >> 27 & 3
    if si == 1 && mod == 0 {
      isCandelaUnit := (smartData.Unit & smartDataUnitMask) ==
         smartDataCandelaUnitPart
      if isCandelaUnit {
        sensorSmartDataQuantity[smartData.AssetID]++
        if num < 2 {
          //consider float64 bytes as int
          sensorSmartDataSum[smartData.AssetID] +=
             float64(math.Float64bits(smartData.Value))
        } else {
          sensorSmartDataSum[smartData.AssetID] += smartData.Value
        }
        ...
  }
  for sensorID, sum := range sensorSmartDataSum {
    sensorSmartDataMean[sensorID] = sum /
       float64(sensorSmartDataQuantity[sensorID])
  }
  luminosityMean := 0.0
  nSensors := 0.0
  for _, sensorLuminosityMean := range sensorSmartDataMean {
    luminosityMean = (luminosityMean*nSensors + sensorLuminosityMean) /
       (nSensors + 1)
    nSensors++
  }
  return luminosityMean * float64(solarPanelsNumber) * 10000000
}
```