AMERICAN UNIVERSITY OF BEIRUT

Securing Smart Grid Communication Using Ethereum Smart Contracts

by
Raphaelle Maria Akhras

A thesis
submitted in partial fulfillment of the requirements
for the degree of Master of Science
to the Department of Computer Science
of the Faculty of Arts and Sciences
at the American University of Beirut

Beirut, Lebanon
June 2020

AMERICAN UNIVERSITY OF BEIRUT

Securing Smart Grid Communication Using Ethereum Smart Contracts

by

Raphaelle Maria Akhras

Approved by:

Dr. Wassim El-Hajj, Associate Professor Computer Science	Advisor
Dr. Haidar Safa, Professor Computer Science	Member of Committee
Dr. Mohamed Nassar, Assistant Professor Computer Science	Member of Committee

Date of thesis defense: June 5, 2020

AMERICAN UNIVERSITY OF BEIRUT

THESIS, DISSERTATION, PROJECT RELEASE FORM

Student Name: Akhras Raphaelle Maria
Last First Middle

☒ Master's Thesis ○ Master's Project ○ Doctoral Dissertation

☒ I authorize the American University of Beirut to: (a) reproduce hard or electronic copies of my thesis, dissertation, or project; (b) include such copies in the archives and digital repositories of the University; and (c) make freely available such copies to third parties for research or educational purposes.

☐ I authorize the American University of Beirut, to: (a) reproduce hard or electronic copies of it; (b) include such copies in the archives and digital repositories of the University; and (c) make freely available such copies to third parties for research or educational purposes after:
One ___ year from the date of submission of my thesis, dissertation or project.
Two ___ years from the date of submission of my thesis , dissertation or project.
Three ___ years from the date of submission of my thesis , dissertation or project.

Raphaelle 6/15/2020
Signature Date

This form is signed when submitting the thesis, dissertation, or project to the University Libraries

Acknowledgements

I would like to acknowledge my advisor, committee, and the NPRP group. Thank you for long meetings and answered questions. Thank you for short discussions in hallways and hours of material revision. Thank you for wild goose chases and fruitful pursuits.

This work was made possible by NPRP11S-1202-170052 grant from Qatar National Research Fund (a member of Qatar Foundation) and the American University of Beirut Research Board (AUB IRB). The statements made herein are solely the responsibility of the authors.

Dedication

I owe many people a debt of gratitude. I would like to thank my family for bearing with me through the thesis process. I thank you for the countless trips down to AUB, rain or shine. I thank you for the home I come back to, loving and warm. I thank you for the support you provide me, continuous and unconditional. I would also like to thank my friends. You have allowed me to smile through times of exhaustion, to prosper through times of despair, to laugh through times of anger, and most importantly to love through times of hate. Thank you to my second home.

My fantastic parents, my favorite twin[1], my loving godmother and grandparents, my treasured aunties, my favorite most fantastic boo, my best friend and sister, my best and truest undergraduate friends, and my pineapple-on-pizza-loving graduate friends; you are my family. I love you all.

[1] twin/researcher/writer/editor/commentator/greatest sister in the world

An Abstract of the Thesis of

Raphaelle Maria Akhras for Master of Science
Major: Computer Science

Title: Securing Smart Grid Communication Using Ethereum Smart Contracts

Smart grids are being continually adopted as a replacement of the traditional power grid systems to ensure safe, efficient, and cost-effective power distribution. The smart grid is a heterogeneous communication network made up of various devices and components such as smart meters, automation, and emerging technologies interacting with each other. As a result, the smart grid inherits most of the security vulnerabilities of cyber systems, putting the smart grid at risk of cyber-attacks. To secure the communication between smart grid entities, namely the smart meters and the utility, we propose in this thesis a communication infrastructure built on top of a blockchain network, specifically Ethereum. All two-way communication between the smart meters and the utility is assumed to be transactions governed by smart contracts. Smart contracts are designed in such a way to ensure that each smart meter is authentic and each smart meter reading is reported securely and privately. We present a simulation of a sample smart grid and report the costs incurred from building such a grid. Each architecture discussed will contain a solution to a problem previously faced and will come with trade-offs that are analyzed in terms of certain metrics. The simulations illustrate the feasibility and security of the proposed architectures.

Contents

Acknowledgements v

Dedication vi

Abstract vii

1 Introduction 1

1.1 Motivation . . . 2
1.2 Problem Definition . . . 3
1.3 Objectives . . . 4
1.4 Organization . . . 5

2 Background 6

2.1 Traditional Grids . . . 7
2.2 Smart Grids . . . 8
2.2.1 Meter Data Management System and Utility . . . 8
2.2.2 Smart Meter . . . 8
2.2.3 Communication Network . . . 9
2.2.4 Advanced Metering Infrastructure . . . 9
2.3 Blockchain . . . 11
2.3.1 Network . . . 11
2.3.2 Ledger . . . 11
2.3.3 Consensus . . . 11
2.3.4 Cryptographic Techniques . . . 12
2.3.5 Platforms . . . 14
2.3.6 Types . . . 15
2.3.7 Security Properties . . . 16
2.4 Ethereum . . . 18
2.4.1 Account . . . 18
2.4.2 Smart Contracts . . . 18
2.4.3 Decentralized Application . . . 20
2.4.4 Ethereum Virtual Machine . . . 20
2.4.5 Transaction . . . 21

2.4.6 Storage . . . 22
2.4.7 Gas and Gas Limit . . . 23
2.4.8 Event . . . 24
2.4.9 Fee . . . 25
2.4.10 Security Properties . . . 25
2.5 Cloud . . . 27
2.5.1 Database . . . 27
2.5.2 Security Properties . . . 27

3 Literature Review 28
3.1 Advanced Meter Infrastructure Attacks . . . 29
3.2 False Data Injection Attacks . . . 31
3.3 Miscellaneous Attacks . . . 33
3.4 Various Solutions . . . 35
3.5 Using Blockchain in the Energy Field . . . 37
3.6 Using Blockchain for Security in Various Fields . . . 42
3.6.1 Performance . . . 42
3.6.2 IoT and Communication . . . 43
3.6.3 Services . . . 44
3.7 Cloud Solutions in the Energy Field . . . 49

4 Architecture Preface 50
4.1 Blockchain . . . 51
4.1.1 Why a Public Blockchain? . . . 52
4.1.2 Why Ethereum? . . . 54
4.2 Smart Grid Communication Network . . . 55

5 Simulation Environment 57
5.1 JavaScript-based Tools . . . 58
5.2 Metamask . . . 59
5.2.1 Main Ethereum Network . . . 59
5.2.2 Rinkeby Test Network . . . 59
5.3 Compiling Smart Contracts . . . 61
5.4 Testing Smart Contracts . . . 62
5.5 Deploying Smart Contracts . . . 65
5.6 Observing Smart Contracts . . . 68
5.7 Subscription to Smart Contract Events . . . 69
5.7.1 Subscribed Events in the Front-End . . . 70
5.8 Public Key Management . . . 71
5.8.1 Key Management for Electrical Data (Utility Keys) . . . 72
5.8.2 Key Management for Load Balancing Data (Consumer Keys) 74

6 Architecture 1 **75**
6.1 Scenario . . . 76
6.1.1 Initial Contract Deployment . . . 76
6.1.2 Contract Interaction . . . 77
6.2 Smart Contract Details . . . 80
6.2.1 First Smart Contract - Joining the Network . . . 80
6.2.2 Second Smart Contract - Setting Up the Communication . 87
6.2.3 Third Smart Contract - Communicating . . . 90
6.2.4 Summary . . . 94
6.3 User Interface . . . 95
6.3.1 First Smart Contract - Joining the Network . . . 95
6.3.2 Second Smart Contract - Setting Up the Communication . 101
6.3.3 Third Smart Contract - Communicating . . . 103
6.4 Results . . . 106
6.4.1 First Smart Contract - Joining the Network . . . 107
6.4.2 Second Smart Contract - Setting Up the Communication & Third Smart Contract - Communicating . . . 109
6.4.3 Summary . . . 111
6.5 Security Properties . . . 112
6.5.1 Smart Contract Properties . . . 112
6.5.2 Blockchain Properties . . . 117
6.6 Limitations . . . 122
6.6.1 Privacy using Pseudo-anonymity . . . 122
6.6.2 Cost of Smart Contracts . . . 122
6.6.3 Scalability . . . 123

7 Architecture 2 (Part A) **124**
7.1 Scenario . . . 125
7.1.1 Initial Contract Deployment . . . 125
7.1.2 Contract Interaction . . . 126
7.2 Smart Contract Details . . . 128
7.2.1 First Smart Contract - Joining the Network . . . 128
7.2.2 Second Smart Contract - Communicating . . . 134
7.2.3 Summary . . . 138
7.3 User Interface . . . 139
7.3.1 First Smart Contract - Joining the Network . . . 139
7.3.2 Second Smart Contract - Communicating . . . 146
7.4 Results . . . 156
7.4.1 First Smart Contract - Joining the Network . . . 157
7.4.2 Second Smart Contract - Communicating . . . 159
7.4.3 Summary . . . 161
7.5 Security Properties . . . 162
7.5.1 Smart Contract Properties . . . 162

7.5.2 Blockchain Properties 166
7.6 Limitations 168
7.6.1 Privacy using Pseudo-anonymity 168
7.6.2 Cost of Smart Contracts 168
7.6.3 Scalability 168

8 Architecture 2 (Part B) 169
8.1 Scenario 170
8.1.1 Initial Contract Deployment 170
8.1.2 Contract Interaction 171
8.2 Smart Contract Details 174
8.2.1 First Smart Contract - Joining the Network 174
8.2.2 Second Smart Contract - Communicating 175
8.2.3 Summary 180
8.3 Cloud Details 181
8.3.1 Connecting to the Database 182
8.3.2 Updating the Tables 183
8.3.3 Querying from the Tables 184
8.4 User Interface 185
8.4.1 First Smart Contract - Joining the Network 185
8.4.2 Second Smart Contract - Communicating 186
8.5 Results 190
8.5.1 First Smart Contract - Joining the Network 191
8.5.2 Second Smart Contract - Communicating 193
8.5.3 Connecting to the Database 195
8.5.4 Updating the Tables 197
8.5.5 Querying from the Tables 199
8.5.6 Summary 201
8.6 Security Properties 202
8.6.1 Smart Contract Properties 202
8.6.2 Blockchain Properties 202
8.7 Limitations 204
8.7.1 Scalability 204

9 Conclusion 206

A Smart Contract Code 208
A.1 Architecture 1 208
A.1.1 First Smart Contract - Joining the Network 208
A.1.2 Second Smart Contract - Setting Up the Communication . 211
A.1.3 Third Smart Contract - Communicating 212
A.2 Architecture 2 (Part A) 213
A.2.1 First Smart Contract - Joining the Network 213

A.2.2 Second Smart Contract - Communicating 215
A.3 Architecture 2 (Part B) 216
A.3.1 First Smart Contract - Joining the Network 216
A.3.2 Second Smart Contract - Communicating 218

B Abbreviations **219**

List of Figures

2.1 AMI Subsystem Interaction 10

4.1 General Layout of Smart Grid Communication System 56

5.1 Simulation Tools Interaction 57
5.2 Testing Smart Contract Functionality 64

6.1 Architecture 1: Initial Contract Deployment 76
6.2 Architecture 1: Contract Interaction (Part 1) 78
6.3 Architecture 1: Contract Interaction (Part 2) 79
6.4 Architecture 1: Smart Contract #1 Distribution 81
6.5 Architecture 1: Smart Contract #2 Distribution 87
6.6 Architecture 1: Smart Contract #3 Distribution 90
6.7 Architecture 1: Join 95
6.8 Architecture 1: Accept Initial 96
6.9 Architecture 1: Accept 97
6.10 Architecture 1: Remove 99
6.11 Architecture 1: Check 100
6.12 Architecture 1: Send Electrical Data 103
6.13 Architecture 1: Get Electrical Data at Time 105

7.1 Architecture 2A: Initial Contract Deployment 125
7.2 Architecture 2A: Contract Interaction (Part 2) 127
7.3 Architecture 2A: Smart Contract #1 Distribution 129
7.4 Architecture 2A: Smart Contract #2 Distribution 134
7.5 Architecture 2A: Join 139
7.6 Architecture 2A: Accept Initial 141
7.7 Architecture 2A: Accept 141
7.8 Architecture 2A: URL - Account Address 141
7.9 Architecture 2A: URL - Contract Address 141
7.10 Architecture 2A: Remove 144
7.11 Architecture 2A: Consumer Interface 146
7.12 Architecture 2A: Send Electrical Data 146
7.13 Architecture 2A: Get Load Balancing Data Instantaneously 148

7.14 Architecture 2A: Utility Interface 150
7.15 Architecture 2A: Send Load Balancing Data 150
7.16 Architecture 2A: Get Electrical Data Instantaneously 152
7.17 Architecture 2A: Get Electrical Data at Time Initial 154
7.18 Architecture 2A: Get Electrical Data at Time 154

8.1 Architecture 2B: Contract Interaction (Part 2) 173
8.2 Architecture 2B: Smart Contract #2 Distribution 176
8.3 Architecture 2B: Consumer Interface 186
8.4 Architecture 2B: Send Electrical Data 186
8.5 Architecture 2B: Utility Interface 188
8.6 Architecture 2B: Get Electrical Data Instantaneously 188
8.7 Architecture 2B: Connection Time at Intervals 195
8.8 Architecture 2B: Connection Time on Average 196
8.9 Architecture 2B: Consumer Updating at Intervals 197
8.10 Architecture 2B: Consumer Updating on Average 198
8.11 Architecture 2B: Utility Querying at Intervals 199
8.12 Architecture 2B: Utility Querying on Average 200

List of Tables

2.1 Blockchain - Bitcoin vs. Ethereum vs. Hyperledger 14
2.2 Blockchain - Public vs. Private vs. Consortium vs. Hybrid 15

3.1 Using Blockchain in Energy Field: Paper - Blockchain - Type - Smart Contract - Simulation 39
3.2 Comparing Security Properties (Part 1): Paper - Confidentiality - Integrity - Availability - Privacy - Scalability 41
3.3 Comparing Concepts: Paper - Blockchain - Type - Smart Contract - Simulation - Other Components (Components) 41
3.4 Using Blockchain for Security – Properties (Part 1): Paper - Confidentiality (C) - Privacy (P) - Integrity (I) - Availability (Av)- Authenticity (Au) - Anonymity (An) - Scalability (S) 46
3.5 Using Blockchain for Security – Concepts (Part 1): Paper - Blockchain (BC) - Type - Smart Contract (SC) - Simulation - Other Components (Components) - Size - Sector 47
3.6 Using Blockchain for Security – Properties (Part 2): Paper - Confidentiality (C) - Privacy (P) - Integrity (I) - Availability (Av)- Authenticity (Au) - Anonymity (An) - Scalability (S) 47
3.7 Using Blockchain for Security – Concepts (Part 2): Paper - Blockchain (BC) - Type - Smart Contract (SC) - Simulation - Other Components (Components) - Size - Sector 48

6.1 Architecture 1: Joining the Smart Grid Communication Network - Smart Contract #1 107
6.2 Architecture 1: Cost of Transactions in Joining - Smart Contract #1 108
6.3 Architecture 1: Communication in the Smart Grid Communication Network - Smart Contract #2 & #3 109
6.4 Architecture 1: Cost of Transactions in Initializing the Communication Environment - Smart Contract #2 & #3 110
6.5 Architecture 1: Cost of Transactions in Sending Electrical Data - Smart Contract #3 110

7.1 Architecture 2A: Joining the Smart Grid Communication Network - Smart Contract #1 . . . 157
7.2 Architecture 2A: Cost of Transactions in Joining - Smart Contract #1 . . . 157
7.3 Architecture 2A: Communication in the Smart Grid Communication Network - Smart Contract #2 . . . 159
7.4 Architecture 2A: Cost of Transactions in Initializing the Communication Environment - Smart Contract #2 . . . 159
7.5 Architecture 2A: Cost of Transactions in Sending Electrical Data - Smart Contract #2 . . . 160
7.6 Architecture 2A: Cost of Transactions in Sending Load Balancing Data - Smart Contract #2 . . . 160

8.1 Architecture 2B: Joining the Smart Grid Communication Network - Smart Contract #1 . . . 191
8.2 Architecture 2B: Cost of Transactions in Joining - Smart Contract #1 . . . 191
8.3 Architecture 2B: Communication in the Smart Grid Communication Network - Smart Contract #2 . . . 193
8.4 Architecture 2B: Cost of Transactions in Initializing the Communication Environment - Smart Contract #2 . . . 193
8.5 Architecture 2B: Cost of Transactions in Sending Electrical Data - Smart Contract #2 . . . 194
8.6 Architecture 2B: Cost of Transactions in Sending Load Balancing Data - Smart Contract #2 . . . 194

Chapter 1

Introduction

The power system infrastructure faces an imbalance between the growing demand for electric power and the limited available supply. The emergence of smart grids came as a solution to a lot of the limitations that accompany the traditional electric grid system. The replacement of old technologies with new ones is not unheard of in this time and age, across all sectors such as the arts, economics, education, environment, governance, health, infrastructure, justice, media, and science. What makes these grids smart is the usage of smart meters, which are far more sophisticated than the currently used meters, at the consumer side. To understand how the different components in the energy sector work together, a network model is used. This model represents objects and their relationships with each other. The data retrieved from smart meters will be used to understand the relationship between the utility and the consumers in terms of energy needs. Thus, network modelling will be based on smart meter data. The shift from the traditional electric grid to the new smart grid brings with it a new set of possibilities for enhancement and improvement at the level of both efficiency and security.

Utility companies have already started enhancing the management of the energy sector while simultaneously increasing the system's reliability and reducing the environmental consequences. Additionally, various approaches have been proposed in order to increase the operational efficiency of many parts of the power network. A recently proposed improvement to the distribution networks highlights the involvement of customers in the demand side management (DSM) and demand response (DR) programs. The goal of DSM programs is to optimize consumer-side energy usage for the long-term whereas the goal of DR programs is to encourage end-users to reduce energy consumption during peak hours for a short-term. Most existing DSM and DR programs entail reducing the power consumption of customers according to predetermined policies of load priorities during peak times. Accordingly, optimized DSM and DR programs play a significant role in helping these programs reach their full potential. These programs cannot function properly or even exist if the data provided to the utility by the

customers is incorrect. If the data that is received at the utility server side contains incorrect information about the energy needs and loads during peak and off hours, the utility cannot make correct decisions or take the appropriate actions in DSM, DR, or other programs aimed at improving the power grid and maximizing its potential.

1.1 Motivation

The smart grid is considered to be a communication network with control devices, smart meters, automation, computers, and emerging technologies interacting. All of these entities must be able to communicate efficiently with one another, making the smart grid a large heterogeneous network. As a result, the smart grid inherits most of the security vulnerabilities of cyber systems, putting the electric grid at risk of cyber-attacks. Accordingly, security breaches in the smart grid can have detrimental consequences and may impact a country's entire infrastructure, its economy, and its citizens' lives. The smart grid's two most critical vulnerabilities are reporting false data and gaining private information about the user. The former vulnerability involves the active interception of data to drop or alter messages sent by the smart meter to the utility and causes the smart grid to act on demands that do not exist and take wrong shedding decisions and actions. The latter vulnerability encompasses the passive interception of messages, which could result in a unauthorized gain of private information about the user. This information could engender more severe attacks such as theft. The targeted data can also be sent from the utility to the smart meters. To address these vulnerabilities, we propose a secure-by-construction prevention mechanism that focuses on Blockchain as the underlying technology to secure the two-way communication between the smart meters and the utility servers.

1.2 Problem Definition

Due to the high risks associated with potential attacks targeting the two-way communication between the smart meters and the utility servers, it is vital to ensure confidentiality, integrity, availability, authorization, authenticity, and non-repudiation. Normal security measures, which are employed in traditional communication and network systems, fail to secure the complex network that composes the smart grid for various reasons which will be discussed in this work. The technique we focus on for resolving the risk associated with the smart grid is a prevention technique. It relies on Blockchain as a building block, where communicated data is treated as transactions that are encrypted and stored in a distributed fashion to ensure security and privacy. Blockchain prevents any alteration of the communicated data, and through cryptographic techniques, content and message authenticity are secured.

In order to demonstrate the prevention approach, a lab setup will be used to prove the practicality of the secure communication between smart meters and servers at the utility. We will first set up a group of nodes (computers) in order to create a Blockchain network. The Blockchain protocols and the security protocols will be configured on every node, while tailoring all cryptographic algorithms for a network that is supposedly composed of thousands of nodes.The smart meters and utility will act as nodes – generators of transactions. We will then perform actual testing on a Blockchain that will be implemented. Algorithm and protocol refinement will be carried out according to how resilient the network is. This setup aims to emulate an AMI and its interaction with the energy management system (EMS) which supports monitoring, controlling, and optimizing the performance of the electric grid. It will include multiple simulations of the various approaches suggested in this paper. These approaches differ in architecture but serve the same purpose. The goal here is to find the approach with the best outcomes and the fewest trade-offs.

1.3 Objectives

The two thematic priorities addressed here are cyber security and Blockchain. Cyber-security comes into play as a concept when dealing with securing the two-way communication between the smart meters and utility servers in smart grids. Blockchain is the tool to accomplish this security goal. The end goal is to have communication that satisfies the CIA triad: confidentiality which is the most important theme in this work in terms of dealing with securing the data from malicious entities (thieves, eavesdroppers, counterfeiters), integrity which is imperative for the communication to be successful (no power outages, theft of power), and availability which is the core necessity of the communication since without the relay of data (in both directions), nothing has been accomplished compared with traditional grids. The work aims to contribute to the improvement of power data distribution and proposes new approaches to secure it. These approaches will be implemented on smart grids; the encompassing theme being smart and customizable security solutions that ensure system protection.

1.4 Organization

This thesis is divided into several chapters beginning with this introduction in Chapter 1. Chapter 2 discusses relevant background information ranging from traditional and smart grids to blockchain and Ethereum specifics. Chapter 3 presents the work related to blockchain implementations for securing communication, smart grid current implementations for securing meter communication, and smart grid meter communication securing with a focus on blockchain proposed solutions. Chapter 4 details the proposed architecture concepts and chapter 5 describes the simulation setup for its various components. Chapters 6, 7, 8 describe the architectures in terms of the followed scenario, the smart contracts devised, the corresponding user interface created, the results recorded, the security properties achieved and the limitations inferred. Finally, Chapter 9 concludes this thesis and presents possible future work and prospects. Refer to Appendix B for any abbreviations made throughout this thesis. Refer to Appendix A for the smart contract code also found at this GitHub repository [1].

Chapter 2

Background

This thesis is composed of many different concepts combined into our architectures. The thesis discusses the securing of the smart grid metering infrastructure. The shift from the traditional electrical grid to the smart grid brought with it many issues which have been tackled in various ways.

In this chapter, we provide an overview of the topics related to our work starting with the traditional grid. We then shift towards the smart grid, discuss its limitations, explain the details of the blockchain platform, and finally move into the Ethereum blockchain platform we use in this thesis and the cloud platform. The limitations that plague the traditional grid are not discussed since this is not within the scope of the thesis. Moreover, the smart grid metering communication limitations, threats, and attacks are discussed in this chapter. These challenges lead to the use of the blockchain platform with all its advantages. In order to emphasize Blockchain's promising role in this thesis, we first aim to provide a comprehensive understanding of the technology by exploring its different aspects. Finally, we discuss Ethereum, the blockchain platform we have selected to provide secure communication between the consumers and the utility in the electric smart grid. The different sections in this chapter provide the necessary foundation for the concepts brought forward in chapters 4, 5, 6, 7, and 8.

2.1 Traditional Grids

The power system infrastructure faces an imbalance between the growing demand for electric power and the limited available supply. There exists a growing concern over suppliers' continued ability to meet new and increasing electric needs. The development of renewable energy sources (RES) is difficult to integrate and implement in the traditional outdated power system infrastructure. These new needs and developments engender myriad obstacles to the operation of generation, transmission, and distribution systems. Moreover, the distribution system is not equipped to deal with periods of intense power consumption. The excessively aged and thermally overloaded transformers can fail, causing interruption to the electric power supply. These concerns, coupled with the ongoing limited capital investment in new transmission systems, necessitate the development of new cost-effective intelligent power systems and infrastructure.

The traditional power grid connects various power system elements,in order to allow electricity to be transferred from the point of generation to the end consumers. It employs electromechanical infrastructure, one-way distribution, and sensors. It is centralized in the sense that power is generated from a central location. These characteristics prevent the traditional power grid from acquiring real time data about the health of the grid's equipment and the exact consumption rates. Thus, it is unfit to meet the increasing demand and is not suitable to incorporate renewable energy sources, such as wind and solar. The smart grid, on the other hand, is becoming a natural replacement to the traditional power grid where digital infrastructure is being used to enhance communication. The emergence of smart grids came as a solution to a lot of the limitations that accompany the traditional electric grid system.

2.2 Smart Grids

The smart grid is not a singular technology. It is a combination of communication, management, and engineering. It was created to save energy and, in turn, to save the environment. Smart grids provide increased power reliability. Unlike in traditional grids, consumers can control their energy usage. They can view their energy consumption through in-home displays.

Smart grids are composed of several entities: the utility company, transmission and distribution entities, and consumers. Many sensors are deployed to pinpoint problems and reroute power as needed. An integral part of the smart grid is the use of sophisticated smart meters, which allow data to be tracked more efficiently, at the consumer side. The meter data management system, promoted by the utility, the smart meter, the communication network, and the advanced metering infrastructure, are discussed in this section since these are the components that are covered in this thesis.

2.2.1 Meter Data Management System and Utility

At the utility side, there is the meter data management system which handles the smart meter data, collected from the Advanced Metering Infrastructure (AMI) and delivered by smart metering systems. The MDMS deals with data management and long-term data storage. It collects the data, cleans it, stores it, and processes it to be analyzed for Demand Side Management, Demand Response, and billing. These storage facilities have to be disaster proof since the data is important to the utility's load or outage management programs. Given the high cost of securing this data, concepts such as virtualization and cloud computing are implemented.

2.2.2 Smart Meter

Smart meters or energy consumption meters are deployed as gateways to the smart grid. Smart meters have the ability to report the electrical data back to the utility at different preset intervals. Smart meters/devices measure and gather electrical consumption data. This data is timestamped to show the time at which it was measured. The data also contains the unique meter identifier. A smart meter is designed to be able to communicate with the MDMS and to receive communication from the MDMS. The data received from the MDMS, which we refer to as load balancing data, is accepted by the smart meter and is acted upon. By replacing the old analogue meters with the new smart meters, the data measurements provided by the consumers are more precise. Different billing mechanisms are set to allow adaptive billing. This can motivate consumers to shift their energy consumption based on the utility control data. The consumers are more in

control of their energy consumption due to the in-home display (IHD) units.

The properties that a smart meter should possess are placed in six categories [2].

- Quantitative measurement which encompasses the accurate measurement of the data.
- Control and calibration which encompasses the ability to adapt to variations in the network.
- Communication which encompasses the sending and receiving of relevant data.
- Power management which encompasses the ability of a system to maintain functionality despite power loss.
- Synchronization which encompasses the reliable transmission of data in a timely fashion.
- Display which encompasses the customers' ability to view their electrical consumption data.

2.2.3 Communication Network

The means through which the smart meters can send data to the utility and vice versa is called the communication channel. The communication channel can be wired or wireless. In our work, we focus on the wireless communication channel between the consumers and the utility, specifically the Internet. As will be mentioned in chapter 3, the communication network faces certain attacks and threats that can be mitigated through the use of our blockchain-based solution

2.2.4 Advanced Metering Infrastructure

The advanced metering infrastructure (AMI) consists of smart meters, a two-way communication network, and a utility. AMI has been used to facilitate the visualization, processing, and management of data for both the consumers and the utility. The AMI transmits the electrical data measured by the smart meters to the utility. The AMI communication network should provide the path for secure information flow.

The use of AMI enables demand side management and allows the integration of green technologies. Smart meter systems provide efficient power system control and monitoring in addition to operation decisions that can be performed in real-time and minimize outages and monetary and electrical loses. Conventional

energy metering systems required manual collection of data and manual billing whereas smart meter systems do not require any manual intervention; the smart meters and the utility communicate via the AMI. The collection of consumers' electrical data allows the utility to manage the electricity demand efficiently.

AMI is not limited to the distribution of energy and the communication of electrical data. It also encompasses the water and gas distribution networks. Figure 2.1 shows the interaction of the smart meters on the consumer end with the MDMS on the utility end via the communication network providing two-way communication (blue and orange arrows).

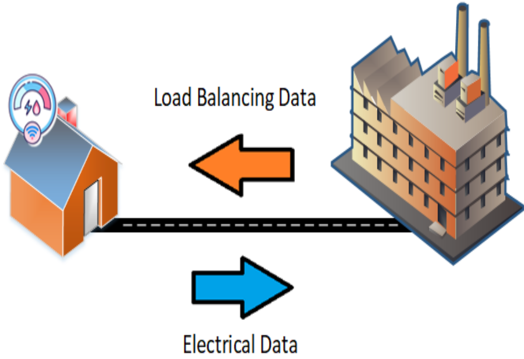

Figure 2.1: AMI Subsystem Interaction

2.3 Blockchain

Blockchain was invented by people/group of people under the name of Satoshi Nakamoto as a transaction ledger for the Bitcoin cryptocurrency [3]. A blockchain can be best described as a distributed computing architecture [4, 5, 6, 7]. Blockchain is a peer-to-peer network that achieves consensus to manage a secure network. It is a chain of blocks where each block contains transactions and a hash of the previous block. Blockchain creates trust through its decentralized nature where no node has control (autonomous) or the power to modify the distributed ledger. It enforces integrity through cryptographically secure transactions and the removal of trusted third parties. It is a decentralized, transparent, auditable, incorruptable chain of data. The traits that characterize blockchain are explained in the coming subsections.

2.3.1 Network

A centralized network is a network that has a central network owner which is a single point of contact for information sharing. A decentralized network is a network that has multiple central owners that have copies of the resources. This eliminates the problem of having a single point of failure that exists in the centralized network. Finally, a distributed network is a network that avoids centralization completely where everyone get full access. Blockchain is a distributed network where all nodes in the network have access to all the information available. The blockchain network is completely decentralized with no dependency on a trusted third party.

2.3.2 Ledger

A database is where data can be managed, stored, edited, and accessed at any time. A blockchain ledger is basically a database where data can be stored and accessed anytime but cannot be altered. Thus, this ledger is append-only and accordingly cannot be corrupted thereby guaranteeing immutability. Each node in the blockchain network will have a copy (distributed over the blockchain network as explained in subsection 2.3.1) of the synchronized ledger. This ledger contains the transactions that were created and signed by different blockchain nodes. These transactions are added to the ledger through the mining process described in subsection 2.3.3.

2.3.3 Consensus

The act of adding a block to the blockchain is known as mining. The miner completes the proof needed to add a block to receive a financial reward. The miner

takes transactions from the transaction pool and hashes them in a particular pattern, in addition to producing other hashes as explained in subsection 2.4.6. The hash of the previous block, nonce and timestamp are added to the block. There are various proofs that can be completed, each with varying computational cost and speed. The miner that has a higher stake, is trusted the most in completing the proof, or succeeds in cracking the puzzle will add the block after the other nodes verify that the proof is correct.

Proof of Work

Proof of Work requires miners to solve a hard math puzzle that changes frequently (to limit the rate at which new blocks can be generated by the network) and is agreed upon by all miners. Once the node validates the transactions and solves the puzzle, the block is submitted to the network which validates the block to guarantee that the submitter isn't falsifying. If it is valid, it is added to the distributed ledger, and the submitter gets a reward. The agreement is based on majority consensus which is impossible to fake unless attackers have control of more than 50 percent of the mining nodes. This needs high computational power.

Proof of Stake

Proof of Stake does not require solving computationally expensive problems. The miner is chosen pseudo-randomly depending on the node's wealth or stake. Attacking it would be expensive since attacking the network requires one to own the near majority; therefore, the attacker will suffer severely from their own attack.

Proof of Space

Proof of Space is similar to Proof of Work. However, the puzzle requires that the node has access to a lot of storage, so any node that has enough space to perform mining can create a new block. This needs high storage capacity.

2.3.4 Cryptographic Techniques

Cryptographic techniques have been developed to provide data security, which ensures that data transferred between parties has confidentiality, integrity, authenticity, privacy, and to prevent malicious users from accessing and misusing data. We mention a brief description of the concepts used in this thesis which are hash functions, digital signatures, and asymmetric cryptography.

Hash Function

A hash function is a cryptographic technique used to ensure the integrity of the data. Hashes of data are added to blocks that are chained together to produce

the blockchain, which ensures it is tamper-proof. Transactions are also hashed before being broadcast. This allows miners to verify the transaction's integrity.

Digital Signature

A digital signature is a cryptographic technique used to ensure the authenticity of the data. The user who is creating a transaction can sign the transaction using his/her private key. The recipient of the transaction can use the creator's public key to validate it before adding it to the blockchain.

Asymmetric Cryptography

Asymmetric cryptography is a cryptographic technique used to ensure the confidentiality of the data. The user creating a transaction can encrypt the data using his/her public key and the recipient of the transaction can decrypt the data using his/her private key.

2.3.5 Platforms

There are various blockchain platforms that can be used. There are many blockchain platforms currently in the ecosystem and table 2.1 shows three different options: Bitcoin, Ethereum, and Hyperledger.

Table 2.1: Blockchain - Bitcoin vs. Ethereum vs. Hyperledger

Platform	**Bitcoin**	**Ethereum**	**Hyperledger**
Currency	Bitcoin	Ether	None
TPS	7	15	3000
Transaction Mode	Manual	Programmable	Programmable
Transaction Speed	Minutes	Seconds	Milliseconds
Transaction Fee	Higher	Lower	None
Smart Contract	None	Yes	Yes
Decentralized Application	None	Yes	Yes
Type	Public	Public	Private
Consensus	PoW	PoW/PoS	PBFT
Model	Transaction-based	Account-based	Account-based
Goal	Digital Currency	DApp Platform	Platform to Create Private Blockchains

Bitcoin allows consumers to buy and sell items, property, or assets without being easily traced. This has made Bitcoin a popular platform for illegal and illicit activities. Ethereum allows entities or companies to build and deploy decentralized applications. These applications could belong to many different sectors. Hyperledger allows industry-wide collaboration in the process of developing blockchain solutions and distributed ledger-based technology.

2.3.6 Types

There are various blockchain platforms that can be used and each of these platforms has a certain type. There are four blockchain types currently in the ecosystem. Table 2.2 provides information on these four different options: public, private, consortium, or hybrid.

Table 2.2: Blockchain - Public vs. Private vs. Consortium vs. Hybrid

Type	Public	Private	Consortium	Hybrid
Centralization	No	Yes	Yes	Yes / No
Security	High	Low	Low	Depends
Trust Needed	No	Yes	Yes	Depends
Participation	Permissionless	Permissioned	Permissioned	Depends
Access	Everyone	Selected	Selected	Depends
User	Anonymous	Known	Known	Depends
Mining	Anyone	Selected	Selected	Depends
Consensus	Depends	Depends	Depends	Depends
Ledger Permissions	Append-only	Append-only	Append-only	Append-only
Performance	Slow	Fast	Fast	Depends
Scalability	Weak	Good	Good	Depends
Transparency	Full	Restricted	Restricted	Depends
Privacy	No	Yes	Yes	Depends
Examples	Bitcoin / Ethereum	Hyperledger / Ripple	Quorum / Corda	Depends

Public

Public blockchain is open, which means that anyone can read the distributed ledger and participate in the blockchain consensus mechanism. Accordingly, there are no regulations for who can participate in the blockchain network activities. The greater the number of nodes in the network, the more secure it is. Thus, nodes are provided incentives to participate in the blockchain activities (mining for instance). In order to mine, each participant in the whole blockchain network must partake in consensus (PoW, PoS, PoSpace as explained in subsection 2.3.3) leading to a low transactional throughput. Any transaction made can be viewed by everyone in the blockchain network. The public blockchain, as its name states, is public for everyone to use, and the distributed ledger is public for anyone to read. Therefore, transactions on the ledger are not private. Public blockchains are used in sectors that can afford for their data to be publicly viewed. Public

blockchains can be used for voting systems, currencies, betting, and even video games.

Private

Private blockchain is closed which means that anyone cannot read the distributed ledger or participate in the blockchain consensus mechanism. Accordingly, there are certain regulations for who can participate in the blockchain network activities. A node must be invited to join based on certain rules. The number of number of nodes in the network does not affect the security of the network. There are rules set up to decide what each node's role is. In order to mine, dedicated miners must partake in consensus (PoW, PoS, PoSpace as explained in subsection 2.3.3) leading to a high transactional throughput. Any transaction that is made can only be viewed by the participants who have knowledge about it in the blockchain network. The private blockchain, as its name states, is not public for everyone to use, and the distributed ledger is not public for anyone to read. Thus, transactions on the ledger are private to the nodes in the private blockchain network. Private blockchains are used in sectors that cannot afford for their data to be publicly viewed. Private blockchains can be used for military or national defense systems, tax return benefits, supply chains, and even government records.

2.3.7 Security Properties

Various security properties characterize the blockchain platform. These described security properties - such as integrity, availability, authenticity, transparency, auditability, accountability, and anonymity provided by blockchain - are integral to our thesis work.

Integrity

Blockchain's resistance to the modification of data can be accredited to their distributed ledgers. These ledgers are immutable: they cannot be changed (modified or removed). Blockchain is thus tamper-proof since any modification to the ledger will be detected.

Availability

Blockchain's decentralization, as explained in subsection 2.3.1, provides availability since each node has a full copy of the ledger. Therefore, attacks on the ledger at certain nodes cannot cause damage to the entire network. There is no single server that has complete control. In addition, transactions that already exist cannot be deleted which maintains data persistence.

Authenticity

Blockchain's use of asymmetric cryptography (explained in subsection 2.3.4) provides the tools needed to generate a digital signature (explained in subsection 2.3.4) for a transaction. This authenticates the transaction that is sent. This transaction is further verified for authenticity by nodes in the network.

Transparency

Blockchain's transparency of information is provided through a verified distributed ledger of transactions. It allows any of the nodes in the network (depending on the blockchain platform used) to view the transactions made.

Auditabiliy

Blockchain's transactions are traceable since every transaction made, added to a block, and broadcast to the network is documented in the distributed ledger. Tracking transactions is possible through the sue of the blockchain platform.

Accountability

Blockchain's transparency and auditability provide the necessary tools to accommodate accountability. The distributed ledger can be referenced to check for any needed data related to a certain blockchain address.

Anonymity

Blockchain's blockchain-specific account per user provides anonymity. This account is used for all of the user's transactions in the network. As long as there is no link between the user's blockchain address and his/her identity, the transaction remains anonymous.

2.4 Ethereum

Ethereum is an open source, public, blockchain-based distributed computing platform and operating system featuring smart contract functionality. Ethereum is a transaction-based state machine where every transaction changes the state of the blockchain. A transaction, after verification and placement in a block, becomes part of an immutable ledger. This ledger allows the Ethereum blockchain to be tamper-proof. Other qualities include the ability to audit data and trace it throughout its life-cycle. It is a programmatic platform that allows decentralized applications (DApps) to run on a decentralized network. These decentralized applications running on the Ethereum network use smart contracts. The concepts mentioned in this introduction will be described in detail in this section. All details mentioned in section 2.3 are relevant.

2.4.1 Account

There are two types of accounts in the Ethereum blockchain network. The first is the externally owned account (EOA) which is the user Ethereum account that contains a public and private key. The EOA can send and sign transactions as well as receive transactions. The second is the contract account which is the address of the deployed smart contract. The contract account can only receive transactions and execute smart contract code.

Each Ethereum externally owned account has a key pair: public key and private key. The account's private key should be kept safe. The account's public key is derived using the account's private key. The account's address is generated by taking part of the hash of the account's public key. The public key cannot be used to derive the public key.

The Ethereum account's private key is used to sign the transactions to prove authenticity. This prevents a transaction from being altered after it is issued.

2.4.2 Smart Contracts

A smart contract is very similar to a real contract which governs a certain agreement between two entities. Neither entity can break the contract. Similarly, a smart contract is a digital agreement between two entities which enforces the contract rules, negotiation, and performance. The code in the smart contract can enforce conditions in an automated manner. The code includes predetermined rules that if met, execute automatically. The transparency involved in these decentralized smart contract transactions as well as the elimination of the third party actors makes the smart contract a desirable conflict-free resource.

Many properties are ensured by the use of smart contracts. Transparency is ensured through the availability of these smart contracts for everyone to see. The smart contracts are thus trustworthy since any interaction with the smart contract is documented for everyone to see. Through processing transactions using smart contracts, the time taken to complete a certain task with third party entities is eliminated where if conditions are met, transactions will succeed otherwise they will fail. This process eliminates along the way any issues that come with the third party interaction including human error. The elimination of the third party entities eliminates any additional costs.

A smart contract may be written in the Solidity programming language. It is currently the most popular language in which to write smart contracts. Solidity is compiled into EVM bytecode know as ABI (Application Binary Interface), and can then be deployed to the platform. It is deployed onto the Ethereum network to ensure certain conditions are met between different parties. Ethereum is the most popular blockchain platform for creating and deploying smart contracts. Deploying a contract is an Ethereum transaction and results in a contract address. Just as an Ethereum account contains an Ethereum address, so does a smart contract. This address is used by other Ethereum nodes to interact with the smart contract.

Once deployed into the network, these contracts are stored safely in the state storage on the ledger of each node in the network, making the contract tamper-proof like any other transaction in the distributed ledger. Thus, when the smart contract is deployed onto the network, the contract may not be altered. The only way to edit a smart contract is to delete it, fix what is wrong, and re-deploy it. Also, once deployed, this contract will be accessible by any account in the blockchain network through the contract address mentioned above. All details concerning the smart contract are accessible including the senders and recipients of transactions, the transaction data, the bytecode executed, and the state of each variable stored in the contract.

The smart contract also contains a balance like other Ethereum addresses. A smart contract may receive a transaction that includes a transfer of ether (using a Solidity payable function). This adds to the contract balance. A smart contract may also send a transaction that includes a transfer of ether. This removes from the contract balance.

A smart contract must be deterministic: given the same input, the same output will be produced every time. A smart contract must be terminable: the smart contract function execution must end given a certain time limit. Finally, a smart contract must be isolated: the smart contract function execution will be separated from the entire ecosystem to stop negative side-effects.

2.4.3 Decentralized Application

A decentralized application is a new type of application that is not controlled by a single entity. It can be noted from its name that it is decentralized. Decentralized applications run on a peer-to-peer network of computers unlike centralized applications. The smart contract is the core logic of the DApp on the blockchain networks. Decentralized applications are not necessarily run on a blockchain network. However, the DApps we discuss are. These DApps that are built on top of the blockchain network ensure data may not be altered or deleted. The smart contract source code which is the core logic is open-source; it can be viewed by all blockchain users.

Developers can code smart contracts on Ethereum, which serves as a decentralized application blueprint. There are various categories of DApps that currently exist on the Ethereum blockchain including currency exchanges, identity management, energy management, health management and others. There are other platforms that allow the creation of decentralized applications such as EOS, Tron, Hyperledger and others.

Thus, to create a decentralized application, a developer must go through the process of creating, testing, and deploying a smart contract. The smart contract will govern how the DApp will function on the blockchain network. The smart contract is tamper-proof and thus so is the DApp. The smart contract is open source and consequently so is the DApp. Finally, the smart contract is not controlled by a trusted third party and accordingly neither is the DApp. After having dealt with the smart contract which can be seen as the back-end of the decentralized application, the development of the front-end of the decentralized application must be considered. The front-end of the application that can be coded using basic HTML/CSS/JS, or it can be developed using the React library, Angular, or any other framework. Using the web3 JavaScript library, which will be discussed later on in this thesis, the user can interact with the Ethereum blockchain and its existing smart contracts. The decentralized application thus consists of the smart contract as back-end, the web3 library as intermediary, and the front-end. It is important to note that the web3 library is not the only option in providing interaction with Ethereum.

2.4.4 Ethereum Virtual Machine

The Ethereum Virtual Machine (EVM) provides a run-time environment where the smart contract code can execute. The smart contract must first be compiled into bytecode that the EVM can read and execute. The EVM is hosted on each of the Ethereum nodes. Any program that is executed in the EVM can be solved due to the Turing Completeness of the EVM. The EVM is needed since smart

contracts have an important property - isolation property - that makes interacting with it dangerous. Subsection 2.4.2 discusses three properties. Anyone can deploy a smart contract and interact with a smart contract. The smart contract deployer could willingly or unknowingly introduce a virus or malicious software. If the execution of smart contract interactions is not isolated, then this could contaminate the whole system. This environment is sandboxed which means it is safe and isolated.

2.4.5 Transaction

A transaction can be anything from sending Ether to another account, deploying a smart contract, or communicating with a smart contract. When a transaction is created, data is written to the blockchain. This updates the Ethereum state. Transactions are available to any node in the network in real-time.

A smart contract contains functions that may be interacted with by different Ethereum blockchain nodes. A smart contract call does not consume any gas and does not update the Ethereum state. It is a call to a function that does not update any variables and only returns a certain value. This is done by using the "call" method. A smart contract can also create a transaction that modifies the Ethereum state and consumes gas. This is done by using the "send" method.

A transaction is composed of various fields. There is the "nonce" field which represents the previous transaction count for the account completing the transaction. There is the "to" field which represents the Ethereum address to which the account completing the transaction is sending their transaction. There is the "from" field which is the account completing the transaction's Ethereum address. There is the "value" field which is the amount of Ether that the account completing the transaction wants to send. There is the "gasLimit" field which is the maximum amount of gas that the transaction should consume. There is the "gasPrice" field which is the amount to pay per unit of gas. Finally, there is the optional "data" field. This field has multiple purposes. In the case of sending Ether from one account to another or sending Ether from an account to a smart contract or sending Ether from the smart contract to an account or sending Ether from one smart contract to another smart contract, it will contain the binary data (payload). In the case of smart contract deployment, it will contain the desired smart contract bytecode. In the case of smart contract function call, it will contain the hexadecimal representation of the desired function in addition to any parameters that are required.

A transaction that leads to the deployment of a smart contract will not contain a "to" field (this will be seen in subsections 6.4, 7.4, and 8.5) whereas any other transaction will have a "to" field containing the Ethereum address for which

the transaction is meant.

After the transaction is complete, the account creating the transaction will sign it with its private key thereby authenticating the transaction. The transaction can be validated by using the creator the transaction's public key.

2.4.6 Storage

Ethereum is a state machine meaning an Ethereum transaction can modify the state of the network. Ethereum stores and manages data using a trie. It is known as the Merkle Patricia Trie[1] which is a combination of the Merkle tree and the Patricia trie.

Merkle Tree

is a structure that is used to efficiently determine if nodes contain the same data through a tree of hashes. The top hash is the hash of their children nodes and the hash on the level below is the hash of their children nodes. This hashing occurs until the leaves that store data are reached. The hash tree structure is used in Ethereum to detect inconsistencies in hashes in the blockchain. If the top hash matches the hash contained, then the data stored preserves its integrity. If the top hash does not match the hash contained, the data has been tampered with and thus the whole path from the leaf to the root has been modified. This proof is faster than other techniques and the data broadcast onto the network to prove validity is small. The use of the Merkle tree structure in a distributed network allows nodes to quickly identify if the data has been tampered with.

Patricia Trie

is a structure that is used to find the common prefixes. Each path from each node contains a key that can be followed to the next node and so on building up the prefix.
Ethereum has four trie data structures that serve different purposes in the network. Account balance data is not stored in the blocks. Root hashes for the state trie, transaction trie, and receipt trie will exist in each block header where the block is mined into the blockchain network.

State Trie

This structure contains the Ethereum addresses. The root hash of this state trie exists in the block header. There is only one state trie in the whole Ethereum network, and it is constantly updated.

[1] https://medium.com/shyft-network-media/understanding-trie-databases-in-ethereum-9f03d2c3325d

Storage Trie

This structure contains the smart contract data specific to the Ethereum account. The root hash of this storage trie exists in the global state trie. There is a storage trie root hash for each Ethereum account.

Transaction Trie

This structure contains the records of the transactions made and will be added to a specific block. The root hash of this transaction trie exists in the block header. Once the block has been successfully mined, this structure cannot be updated. There is a transaction trie root hash for each block.

Receipt Trie

This structure contains the results of the transaction made and will be added to a specific block. The root hash of this receipt trie exists in the block header. Once the block has been successfully mined, this structure cannot be updated. There is a receipt trie root hash for each block.

2.4.7 Gas and Gas Limit

There exists two important fields to always consider in an Ethereum transaction: gas limit and gas used. The gas is the amount of fuel required for a transaction to be mined. The gas needed by the transaction is then multiplied by the cost of the gas-per-unit to get the total gas price. The amount of gas used is can be determined by looking at the function that is called. This function contains smart contract code that is composed of instructions internally. Each instruction in turn has a certain cost. Thus, the total cost, in terms of gas, for a called function is the summation of the individual instruction costs.

The gas limit is the upper boundary a user is willing to pay for the transaction to be successful. If the gas limit is high, the function is computationally complex and hence needs more work to be executed. A standard ether transfer needs a gas limit of 21,000 units of gas. This transfer does not include any restrictions or complicated expressions that would jack up the gas used. Gas is paid in ether, the cryptocurrency used in Ethereum.

The used gas per transaction differs drastically from deploying a contract, to calling the simplest smart contract function concerned with changing a state variable. In fact, in a smart contract, the amount of gas used differs if the function invoked is a call or an update. If the function is a call, this means that no gas will be consumed, where a value is simply returned from the local distributed ledger. If the function is an update, then gas will be consumed according to the

simplicity or complexity of the function, where a change in the Ethereum state will occur.

2.4.8 Event

An Ethereum event logs the change in state of the Ethereum blockchain which can easily be retrieved and filtered. Events contain data about the change of the state that occurred. They facilitate the communication between the user interface of an application and the smart contract. Events can also trigger data events in the user application and are a much cheaper form of storage. Storage is expensive; therefore, using an event can be multiple times cheaper.

When an event is fired/emitted, logs are written to the blockchain. Events are emitted through smart contract functions. When a user calls a smart contract function that contains an event, and this transaction is mined in a block, the event will be included with all parameters that it contains. The event is found in the transaction receipt. Events may also be filtered to tailor the user need. Applications that interact with Ethereum smart contracts can easily track changes to these smart contracts using events. Listing 2.1 shows an example of the structure of an event.

```
1 event LoadBalancingSent(address indexed _to, bytes32 _value);
```

Listing 2.1: Example of Event

Each event log consists of topics and data. There can be up to 4 topics in an event. The first topic will typically be the name of the event in addition to the types of its parameters hashed using keccak256 hash. This signature is not included if the sender emits an anonymous event. In this case it will be:
Keccak-256 Hash(LoadBalancingSent(address,bytes32)**)**.

If the argument is declared as indexed, then it is treated like an additional topic. So in this case, there are now two topics. One for the signature, and another for the indexed argument. If the argument is not declared as indexed, then it will be attached as data. Using data is a lot cheaper than including topics. Creating an event takes a minimum of 375 gas. Any additional topic will cost 375 gas as well, and each byte of data will cost 8 gas. This is the cost of emitting an event and not the total cost of the transaction.

To be able to retrieve events, the web3 library is used to subscribe to an event or just watch for an event. The web3 library will be explained in upcoming sections. Users do not have to query the distributed ledger for data which is cumbersome; they just have to watch for incoming events using the provided functions in the web3 library. They can also filter these events based on the topics discussed above to receive only certain events that matter to them.

Event sourcing is used to automatically update state and publish events. Instead of using the traditional approach and storing a variable in the smart contract to save its current state, event sourcing uses the sequence of events that alter the Ethereum state to recreate the current state.

2.4.9 Fee

A fee is an amount to be paid for services rendered. For every update function in the Ethereum smart contract, there is a fee that must be paid in order to process the transaction. This is due to the fact that an update function will change the state of the Ethereum blockchain. This fee is calculated based on the instructions in the function. The more complex the function, the more the transaction fee will be. It takes, as mentioned, a certain amount of gas per statement. For every call function in the Ethereum smart contract, there is no fee that must be paid since a call does not change the state of the Ethereum blockchain and consequently does not cost the user any money.

Ethereum is a Turing Complete system and can face the halting problem. Executing code depletes gas. Each function consumes a certain amount of gas. If this amount of gas consumed by the function is more than the gas given, the execution will exit before the code finishes running, and the transaction will be aborted. This mechanism keeps Ethereum safe from Denial of Service Attacks (DoS). Thus, if a function is called multiple times in an attempt to disrupt the network, the attacker will have a hefty fee to pay. This also applies to creating a function with a for loop that can consume all the gas allotted and lead to premature termination. The smart contracts are made in such a way that for loops are expensive. The fact that fees exist on transactions denies malicious users the ability to disrupt the network through infinitely sending transactions using a loop.

2.4.10 Security Properties

Various security properties characterize the Ethereum blockchain platform. The security properties described including integrity, availability, authenticity, auditability, anonymity, and transparency provided by blockchain are integral to our thesis work. The security against cyber attacks is mentioned in [2] concerning confidentiality, integrity, availability, and accountability. Confidentiality is compromised when unauthorized entities access stored meter data. Integrity is compromised when unauthorized users pretend they are authorized and issue illegal commands. Availability is compromised when a component fails. Finally, accountability is compromised when audit logs are tampered with.

Subsection 2.3.7 describes in detail these different security properties that the blockchain platforms achieve. The Ethereum blockchain achieves these same properties. The Ethereum blockchain maintains the integrity of data. It maintains the availability of data. It ensures the authenticity and auditability of data. The Ethereum blockchain ensures the pseudo-anonymity of the Ethereum accounts. Finally, it ensures the transparency of data.

2.5 Cloud

Cloud storage is a remote platform for scalable storage resources. The user can send files manually or in an automated fashion to data servers. This data is accessible through the cloud interface. Multiple servers exist and the fact that data is replicated on multiple servers ensures availability.

2.5.1 Database

A database is a structured collection of information. It can either be centralized, decentralized, or distributed. A cloud-native database is specifically built for the cloud, runs on the cloud, and will provide the best services. The way the data is stored improves flexibility and enhances clustering capabilities. Non-relational databases can be deployed easily on the cloud, and Relational databases can be used in the cloud but were not originally designed for distributed systems.

2.5.2 Security Properties

Various security properties characterize the cloud platform. The security properties described, including disaster recovery, accessibility and availability, scalability and the benefit of low cost provided by the cloud, are integral to our thesis work.

Disaster Recovery

Cloud storage provides data recovery for issues faced. There is no risk of losing data due to system failures since it is stored and backed up on external devices. Cloud storage also provides easy back-up of data.

Accessibility and Availability

Cloud storage provides accessible data through consistent replication. The data is stored on multiple data servers so it is accessible from any location with connectivity. The remote data locations allow easy access.

Scalability

Cloud storage consumers only have to pay for the storage needed. The cloud operator can accommodate growth or shrinkage in data storage needs.

Low Cost

Cloud storage is a cheaper alternative for local storage since it can be expensive. The cloud provides the infrastructure and man power needed. It also allows consumers to avoid the need to invest in expensive server infrastructure.

Chapter 3

Literature Review

The objective of this literature survey is to give a general overview about some important concepts related to the thesis work. We start by giving an overview about the different attacks that can compromise the smart grid network including Advanced Meter Infrastructure Attacks, False Data Injection Attacks, GPS Spoofing Attacks, Time Stamp Attacks, Topology Attacks, and Man in the Middle Attacks. We move on to explain how blockchain (our selected platform) is used in both the electrical field and various other fields to secure the communication between several entities. Our work depends on the wireless communication protocols as described in [8]. The Advanced Metering Infrastructure uses wireless systems for data communication in the smart grid.

3.1 Advanced Meter Infrastructure Attacks

The Advanced Meter Infrastructure (AMI) is composed of a smart meter, a communication link, and the Meter Data Management System (MDMS). These three components work together to provide bi-directional communication in the smart grid network. [9] proposes coupling the Public Key Infrastructure (PKI) mechanism and the ID-based authentication to offer authentication of smart meters. This provides a secure environment for smart meters through mutual authentication and reduces the message exchanges. However, this solution relies on a Trusted Third Party (TTP). [10] describes the need for authentication of the entities that are part of the smart grid network to determine whether or not they are authorized to interact in the network. These measures must ensure confidentiality, integrity, availability and accountability in the AMI systems. **Illegitimate control commands can be sent from the AMI headend to reset meters**. **Audit logs of grid interactions are also vulnerable to tampering**. [11] discusses the security concerns in the AMI and the paths that can be exploited to attack. A model is used to provide a metric to evaluate the security mechanisms and a simulation study is conducted. Vulnerability analysis, cyber security investment optimization, and cyber contingency analysis show the impact on information exposure. The papers mentioned tackle few security properties but none tackle confidentiality, integrity, availability, authentication, and accountability which our thesis's blockchain solution provides.

[12] discusses different aspects of the smart metering system by exploring the advantages and disadvantages in terms of both the utility and the customers. Smart meter communication networks are also presented in addition to the challenges in development and maintenance of smart meters. The communication technologies are important to secure for the delivery of data. [13] proposes that the meters can be Bluetooth based and will send the data wirelessly to the control center. This paper **proposes to change the current meters**. [14] discusses the Power Line Carrier (PLC) as being an effective technology for the power grid. [15] also describes the Broadband Power Line and the PLC communications that can transfer data using TCP/IP. This paper **is vulnerable to attacks from other TCP/IP devices**. [16] proposes an algorithm that combines the MAC algorithm and the IPv6 protocol to deliver the data. [17] proposes the use of the Session Initiation Protocol (SIP) to communication in the smart grid. **SIP, however, is vulnerable to eavesdropping.** [18] provides enhancements to the Distributed Network Protocol 3 (DNP3) protocol to create a more secure protocol. An enhancement is the addition of access rules for data to provide authorization. [19] suggests using The Zigbee protocol for data transfer and [20] proposes a peer-to-peer network over the internet that is cost-effective. General Packet Radio Service provides another communication medium [21]. **These three papers mention the use of a wireless communication which we**

use in our thesis but they do not address all the security measures ensured in our thesis. Data transmitted through wireless communication is usually encrypted and authenticated allowing a decrease in eavesdropping but the availability of the data is not ensured [22].

[2] and [23] are both surveys on the Advanced Metering Infrastructure. [23] is the journal from which [2] is extracted. [23] mentions encryption to secure the communication network since it must be reliable for transferring huge loads of data. Everything mentioned in [2] is contained in the journal. [2] mentions the different security aspects of the AMI. The privacy of the end user's information should be ensured. Detailed private data can be collected as shown in [24, 25]. Privacy can be ensured through load signature modernization [26]. Another procedure called minimization can be used to protect the consumer privacy through hiding, smoothing, and mystifying data [27]. [2] mentions the attacks on meter data through data collection or data transit, or from stored data. The details of these attacks can be seen in [28]'s attack tree. Moreover, spoofing can occur if encryption/authentication is not implemented properly. It can also lead to Man In the Middle (MIM).

[29] surveys the threats to the metering systems in the smart grids where the objectives are to maintain the availability of the power grid, provide legitimate power consumption and delivery, and provide privacy for consumer data. Solutions include authentication and encryption of communication to avoid tampering of messages and eavesdropping. In addition, key management must be secure. There could be attacks on the concentrator nodes where the data is aggregated. Solutions include online monitoring of network traffic and detection techniques. This paper does not tackle preventative measures related to data availability and auditability. [30] also surveys the metering and communication systems and identifies the possible threats/attacks and their solutions. [25] shows how fifteen minutes of consumer data is sufficient to understand consumer usage patterns. The threats and their solutions are documented in the miscellaneous attacks section 3.4.

[31] and [32] both discuss techniques to prevent malware propagation in the AMI. [31] proposes on-site investigation at certain time periods and monitoring of the system to investigate anomalies using the Markovian decision process. This paper implements a detection approach by checking deployed meters regularly. [32] proposes a policy engine to check for anomalies in AMI traffic. This paper also implements a detection approach by detecting a disturbance in certain properties in the data payload.

3.2 False Data Injection Attacks

A false data injection attack (FDIA) is when a malicious user compromises smart meter electrical data (collected measurements) to evade detection at the utility side. BY doing so, the malicious user causes the utility to act on incorrect data and send out load balancing commands that compromise the power system either physically or economically. Injecting incorrect data measurements into the smart meter must be studied and performed in a coordinated fashion. The attack can evade detection by allowing the estimation residual to fall below the test hypothesis threshold. [33] describes the 2015 Ukraine blackout which was caused by a cyber-attack. The attackers were able to covertly operate for extended periods of time to gather information about the smart grid. They were able to intercept from SCADA servers communication messages due to the **lack of a cryptographically secure communication protocol** in the SCADA communication layer.

To mitigate the threat of receiving incorrect data from consumers, the utility put in place the bad data detection (BDD) unit which gets rid of incorrect measurements. FDIA can still circumvent this unit and inject false data into the value of the estimated state. [34] provides a survey on FDIA literature. It states that measurements can be altered in two ways. The RTUs could be manipulated or **the meter could be compromised** leading to incorrect data reported to the utility. [35] provides a review of the FDIA literature as well. Measures that protect the power grid against FDIAs include securing physical power grid components [36], **enhancing the communication security**, or changing the structure of information flow in the grid [37]. Many papers explain the attack-based cyber topology capable of disrupting the electrical data through false data injection attacks. [38] discusses an attack that causes the customers in the market to pay higher electrical bills. This is done by small continuous attacks extending over a long period of time. This affects the customers and not the power system. This attack is done by attackers with full knowledge of the smart grid network topology and can **compromise several meters to send false information to the utility**. [39] and [40] state that full knowledge of the smart grid network is not necessary. Information about the region that will be attacked is sufficient for the attack [41]. [42] discusses the attacks where the malicious users control a set of smart meters and can alter the measurements of these meters. [43] discusses the influences of FDIAs on the power system's static security assessment (SSA). The attacker can manipulate the secure and insecure signals by injecting false data which would lead the utility to operate using false data. This is possible through the assumption that an **attacker can falsify meter measurements**. [44] shows the economic impact of the false data injection attacks on the power market. It displays how attacks can be constructed and executed without being detected and their economic effects. [45] discusses the relationship between false

data injection attacks and the electrical market operations. It proposes an online attack that does not need the network topology to launch an attack. It can **execute the attack in real-time through data streams of meter measurements**. To defend against this attack, a detection scheme is put in place. Unlike our work, this paper tends toward a detection scheme rather than a prevention scheme. **The papers in the section show the need for a secure communication channel for the cases where false data is injected in the network and not through compromised meters.**

3.3 Miscellaneous Attacks

Many other attacks exist on the smart grid network and operations. One of the various attacks is the time stamp attack (TSA). [46] shows the effectiveness of time stamp attacks which target the smart grid time information. The smart meters are equipped with a global positioning system to get precise times for the synchronous activities in the smart grid. GPS spoofing is performed. In the communication infrastructure of the smart grid, meter data is transferred with the time stamps. The utility uses this data for taking actions for DSM and analyzing the system state. As a result, a time stamp attack will have a negative impact on the utility's decisions for the grid. **The paper shows the need for a secure communication infrastructure that will not be vulnerable to time stamp attacks.** [47, 48, 49] describe other instances of GPS spoofing attacks leading to synchronization issues in the grid.

Topology attacks in smart grids are discussed in the next few papers. In [50], a proposed method to determine an attacking region based on minimal network information is discussed. They show the real-time topology of the grid's vulnerability. [51] describes the undetectable attacks on the smart grid network and how to prevent them. Man in the Middle attacks, which exploit the lack of an authentication system to allow a user to impersonate a legitimate user, are considered. **This allows the malicious user to replace authentic packets with fake packets.** This misleads the utility and can falsify network topology. To prevent such strong attacks, [51] proposes to secure a subset of meters as their solution. [52] describes different cyber-topology attacks: line-addition attack, line-removal attack, and line-switching attack. **The attacks mislead the utility in their decision-making process through introducing fake grid topology information.** These attacks lead to economic loss for both the utility and the consumers. Topology errors lead to erroneous state estimation [53] and influence optimal power flow [54].

The traditional communication systems are already vulnerable to attacks violating integrity, privacy, authorization, and accountability. The new smart communication systems integrate information and communication technology, leading to even more vulnerabilities. [22] provides a survey of the power system cyber security issues and their solutions. Unauthorized access to private networks can be solved by using firewalls.**Firewalls need to contain numerous rules to cover all aspects and this is not feasible. In addition, they cannot secure against spoofing attacks and other vulnerabilities.** Ensuring confidentiality and integrity will require cryptographic mechanisms. However, most communication networks (DNP3 and MODBUS) used to secure SCADA and others do not use these mechanisms [55, 56]. [57, 58] proposes a secure frame format in addition to a lightweight authentication mechanism in DNP3 to secure

the communication protocol, and [59, 60] proposes authentication mechanisms in MODBUS to secure the communication protocol. **Our thesis uses blockchain to secure the communication network since there limitations in firewalls and Intrusion Detection Systems that still lead to malicious attacks such as coordinated cyber attacks**. In addition, the SCADA system is a target for attacks that can later target AMI and DER systems. Information exchange through WANs is vulnerable [61].

[62] mentions different cybersecurity issues in the power system communication systems. One of these issues is the possibility of reading the consumers' consumption or the utility's control data remotely without being detected. [63] mentions customer information leakage due to networking intrusion. [64] reviews threats on the smart grid communication networks. Attacks are studied on network availability, data integrity, and information privacy all of which are our concerns. In data integrity and information privacy, attackers target consumer information with the goal of modifying it to corrupt data exchange (data integrity) or eavesdroping to acquire information (information privacy). Solutions for these types of attacks include authentication protocols [65, 66, 67], intrusion detection, and firewall/gateway design. Energy fraud, which could occur as shown in the Hack in Paris[1] because of re-injected old packets, could not occur with our solution due to the use of timestamps.

Another attack that will be discussed is the infamous attack on the Ukrainian power grid. [68] explains the steps taken during the attack and provides measures that could be taken to mitigate the resulting issues. It describes ways to deal with the SCADA defense. The Ukrainian grid experienced a set of coordinated cyber attacks where illegal third parties accessed the computers and SCADA systems. This attack lead to power outages affecting around 225,000 customers for several hours. One of the recommendations made to ensure the safety of the power grid system is to ensure that logging of actions on the grid is enabled. This guarantees accountability; in the event of a future attack, the malicious users will be identified. Other recommendations include event monitoring systems, configuring intrusion detection systems, and personnel training. This paper identifies techniques and procedures used in this attack. **It shows the need for a secure communication infrastructure that contains authentication standards in addition to data provenance.**

[1]https://hackinparis.com/archives/2014/

3.4 Various Solutions

[69] proposes an architecture that utilizes distributed intrusion detection to improve cybersecurity in the wireless mesh network by analyzing network traffic using machine learning and artificial immune systems. This paper does not target preventing malicious attacks. Instead, it aims to detect and identify these malicious attacks in order to prevent them from recurring. [66] discusses an encryption scheme used to secure electricity grids against adaptive ciphertext attacks. This paper tackles confidentiality of data but does not concern itself with the other properties we tackle. [70] discusses the Camenisch-Lysyanskaya signature that provides privacy, authenticity, and auditability.

[71] describes a privacy preserving mechanism where utilities consumers can collect measurement data. It introduces privacy preserving nodes that are controlled by independent parties. These parties could be a point of failure. [72] mentions attribute-based encryption as well as pseudonyms to ensure anonymity which in turn enhances privacy whereas [73] uses homomorphic encryption to ensure privacy. [74] introduced a privacy preserving range query scheme over encrypted metering data to provide privacy of data and authorization of consumers accessing data. These solutions achieve confidentiality and privacy but do not tackle other security properties in the metering and communication systems. [75] proposes a key management scheme that lacks scalability and suffers from desynchronization attack. [76] proposes a scalable key management scheme by using both an efficient key tree technique and an identity-based cryptosystem. It also mentions that key management schemes designed for IT systems are not compatible with the smart grid heterogeneous components. Authentication can occur using Diffie-Hellman for lightweight message authentication in smart grid communication in [77]. [78] shows that a proposed key management system protocol that mitigates man-in-the-middle attacks [79] is susceptible to man-in-the-middle attacks, and [78] proposes a key distribution protocol using trusted third parties for the smart grid. This paper depends on a trusted third party which we remove in our work for security purposes. [75] also discusses a key graph solution based on key management systems. [80] employs Physically Unclonable Function for key exchange and authentication in the smart grid's advanced metering infrastructure. It relies on a single centralized server to store all the smart meter data thus making it unscalable. These solutions achieve availability and integrity but do not tackle other security properties in the metering and communication systems. [81] mentions the mutual inspection strategy between the data aggregator and the smart meters. It does not consider other communications. [82] discuss communication protocols based on peer review strategy that provide accountability in the smart grid communication. These solutions achieve non-repudiation but do not tackle other security properties in the metering and communication systems. [83] surveys the cyber security threats on communication and their solutions for the smart grid.

For privacy, [84] proposes compressed meter reading for communicating the data where the randomness of the sequence provides privacy. [85] proposes encrypting the data using keys. [86] puts forward a method to securely anonymize the data communicated but requires a third party for authentication. For integrity, [87] proposes semantic checks, certificates, and trusted third parties. For availability, the International Standards Organization and International Electrotechnical Commission propose adding to the data transmission information that will be used to verify the authenticity. [88] proposes the use of digital signatures and timestamps to authenticate users. These solutions tackle each security property individually but do not handle them all at once.

3.5 Using Blockchain in the Energy Field

In most of these papers, blockchain technology is used on top of the smart grid to facilitate energy trading in its various forms. It is used to promote the buying and selling of electricity/power in local neighborhoods and microgrids. These papers assume the smart grid's advanced metering infrastructure's communication is secure.

LO3 Energy[2] uses the Ethereum blockchain to allow members of the microgrid to interact using transactions which provided more flexibility in the microgrid. It also allowed "prosumers" to sell their energy to other microgrid members. It enables transparency of the transactions. Brooklyn Microgrid[3] and the New York TransActive Grid[4] are blockchain-based energy marketplaces. Decentralised Energy Exchange[5] is a non-blockchain-based online marketplace that allows households to sell their excess renewable energy. Likewise, in Australia, AGL[6] also began implementing a blockchain-based solar trading program. WePower[7] offers a green marketplace using blockchain to buy and sell energy from producers. In Singapore, SP Group[8] created a blockchain-powered certificate marketplace platform that allows consumers to buy electricity from green energy providers. ACCIONA Energía[9] is using blockchain technology to trace renewable electricity generation. Volt Markets[10] is an energy tracking platform. Ethereum blockchain's smart contracts are used to trade and track energy. European utilities are also participating in the new trend of using blockchain technology in energy trading. Ponton[11] uses Enerchain, a blockchain-based energy trading market. Lition Energy Exchange[12] developed Ethereum smart contracts to provide buyers and sellers the ability to sign energy transactions for trading energy. EW[13] launched the blockchain-based Energy Web Chain to track the ownership of renewable energy production. In Texas, GridPlus[14] customers can trade electricity utilizing cryptopayments. Even Shell[15], invested in blockchain-based energy tracking startups.

[2]https://lo3energy.com/
[3]https://www.brooklyn.energy/
[4]http://www.solutionsandco.org/project/transactive-grid/
[5]https://arena.gov.au/projects/decentralised-energy-exchange-dex/
[6]https://www.agl.com.au/
[7]https://www.wepower.com/
[8]https://rec.spdigital.io/
[9]https://www.acciona.com/pressroom/news/2020/february/acciona-pioneering-application-blockchain-emissions-trading-alliance-climatetrade/
[10]https://voltmarkets.com/
[11]https://www.ponton.de/b2b-integration/blockchain/
[12]https://www.lition.io/
[13]https://www.energyweb.org/
[14]https://gridplus.io/
[15]https://www.shell.com/inside-energy/blockchain.html

[89] mentions the pros and cons of using blockchain in the energy system. It discusses the use of Ethereum smart contracts for making energy transactions and the issues regarding transaction limitations, the need for data manipulations at time, and the need for legal regulation. [90] mentions using blockchain technology to optimize distributed energy systems. [91] proposes the use of a private blockchain for distributed communication to provide energy within a decentralized local energy market. [92] proposes a game theory approach to provide a demand side management model using blockchain to ensure secure implementation. Blockchain provides a flexible peer-to-peer power trading system. It is used to communicate privately. [93] discusses using blockchain to set up a microgrid electric network where local microgrid consumers and producers can trade energy. This paper proposes the use of smart contracts to manage the energy trading transactions. [94] proposes the use of blockchain technology to exchange information for energy trading by utilizing the distributed ledger to improve quality of life. [95] proposes the use of blockchain, multi-signatures, and anonymously encrypted messaging streams to secure energy trading. This paper provides more privacy than centralized trading solutions. [96] proposes a framework using blockchain for the energy sector and explores the challenges of doing so. The framework incorporates institutional, environmental, social, economic, and technological aspects. [97] is a thesis dissertation expressing the advantages and challenges of using the blockchain in the smart grid. The use of blockchain is mentioned for trading energy in microgrids where "prosumers" can exchange their surplus energy in the local market.

The papers mentioned in this section have been placed in table 3.1 for a clearer understanding of the concepts we have implemented in this thesis. The occurrence or lack thereof of these concepts shows the difference between these papers and what we have implemented.

Table 3.1: Using Blockchain in Energy Field: Paper - Blockchain - Type - Smart Contract - Simulation

Paper	Blockchain	Type	Smart Contract	Simulation
[89]	Ethereum	-	Yes	No
[90]	-	Private	No	No
[91]	-	Private	No	Yes
[92]	ZigLedger	Consortium	Yes	Yes
[93]	-	-	Yes	No
[94]	-	-	No	Yes
[95]	-	-	No	Yes
[96]	-	-	No	No

In all the papers above, blockchain technology is used to promote the buying and selling of electricity/power in local neighborhoods and microgrids. These papers assume the smart grid's advanced metering infrastructure's communication is secure. In our approach, we do not adopt the assumption that the two-way communication is secure and propose an architecture that ensures the security of this two-way communication. Any breach in the two-way communication between the utility and the consumer entities rules any proposed application in the smart grid as useless. Hence, our architecture can act as a building block that augments the proposed approaches and secures all communication between any two points in the smart grid, using public Ethereum blockchain and cryptography functions. A few other papers have tackled the use of the blockchain network in the smart grid. We will discuss these papers and have formulated two tables for a clearer understanding of their concepts. The occurrence or lack thereof of these concepts shows the difference between these papers and what we have implemented.

[98] briefly discusses using the Ethereum blockchain for dealing with smart meter data. They show a simple, one function contract, that handles meter data. The paper is very brief in its description of what happens in the network, lacks many details for a proof-of-concept, and contains no mention of a simulation or security properties achieved. [99] discusses the use of blockchain in the smart grid. The authors in [99] propose a new distributed blockchain-based protection framework. This framework protects the power system in terms of self-defensive capabilities. This, in turn, increases the robustness of the modern power grid by implementing a private blockchain and re-configuring some blockchain concepts such as broadcasting transactions, verifying transactions, mining and block generation, and consensus, while using the smart meters as nodes in this private network. **This proposed framework does not implement smart contracts using the Ethereum network as we have done in our thesis. The use of smart**

contracts enhances security and provides reliability. This framework also could lead to information disclosure since the data is redundantly placed in distributed storage without considering data confidentiality or user privacy. [100] uses blockchain technology, including smart contracts, to mitigate the issue of compromising data and to provide consumers accessibility to their data. [100] uses a sovereign blockchain with side blocks. The architecture proposed depends on a registration and an authentication layer initially. This layer is accessed every time a node logs onto the system. Once in the system, nodes can communicate data to the smart grid network. This data will also be stored in a data center which causes redundancy. **In addition, certain details such as using the smart contract for creating private and public keys is not effective since everything done on the smart contract is public and documented on the ledger. Thus, all data encrypted using these keys is neither authentic nor confidential.** [101] suggests using Rainbowchain to secure the smart grid and energy exchange. Rainbowchain is a blockchain technology that contains seven authentication techniques. It is described in detail in [101], proving that it provides superior performance in terms of integrity and scalability. **However, the data collected by the smart meters may lead to the disclosure of the users' personal information.** [102] proposes the use of private blockchains and pseudonyms to preserve privacy. **As we discuss in this thesis, the use of pseudonyms is not enough to preserve the anonymity of the user and thus the users' privacy. In addition, the division of the users into small groups increases the risk of the 51% attack.**

The three works we discuss all revolve around providing integrity-first IoT communication using the Ethereum blockchain. [103, 104] describe the use of the Ethereum blockchain to provide integrity-first communication in the smart grid. This paper focuses on reducing the blockchain code size in addition to not storing blockchain data on the device. **The focus of the paper is just on the integrity of the data and does not deal with data privacy.** In the thesis which tackles IoT integrity-first communication, [105] discusses a light client that is an integrity first communication protocol for IoT devices based on the Ethereum blockchain. This ensures the data is not compromised and the framework is lightweight. **This proposed framework does not implement smart contracts using the Ethereum network as we have done in our thesis. The use of smart contracts enhances security and provides reliability.** The three works we've discussed revolve around IoT integrity in communication using the Ethereum blockchain. The works, however, do not tackle the transaction latency problem which is found in the Ethereum blockchain. The works describe light clients in the Ethereum network which leads to the need for trust in the network. The works also do not mention the use of smart contracts we discuss in our work.

In order to provide a clearer understanding of the papers we compare ourselves with respect to the usage of the blockchain in the smart grid, these papers have been organized in tables 3.2 and 3.3. The occurrence or lack thereof of these properties and concepts shows the difference between these papers and what we have implemented.

Table 3.2: Comparing Security Properties (Part 1): Paper - Confidentiality - Integrity - Availability - Privacy - Scalability

Paper	Confidentiality	Integrity	Availability	Privacy	Scalability
[98]	No	Yes	Yes	No	No
[99]	No	Yes	Yes	No	Yes
[100]	No	Yes	Yes	No	Yes
[101]	No	Yes	Yes	Yes	Yes
[102]	No	Yes	Yes	No	Yes
[103, 104]	No	Yes	Yes	No	No
[105]	No	Yes	Yes	No	No
This Thesis	Yes	Yes	Yes	Yes	Minimal

Table 3.3: Comparing Concepts: Paper - Blockchain - Type - Smart Contract - Simulation - Other Components (Components)

Paper	Blockchain	Type	Smart Contract	Simulation	Components
[98]	Ethereum	-	Yes	No	None
[99]	-	Private	No	Yes	Distributed Storage
[100]	Sovereign	Private	Yes	No	None
[101]	Rainbowchain	Private	Yes	Yes	None
[102]	-	Private	No	Yes	None
[103, 104]	Ethereum	Public	No	Yes	None
[105]	Ethereum	Public	No	Yes	None
This Thesis	Ethereum	Public	Yes	Yes	Cloud

3.6 Using Blockchain for Security in Various Fields

In section 3.5, blockchain technology is used on top of the smart grid to facilitate energy trading in its various forms. It is used to promote the buying and selling of electricity/power in local neighborhoods and microgrids. These papers assume the smart grid's advanced metering infrastructure's communication is secure. In this section, blockchain technology is used in various fields to secure communication and provide security features for applications, governments, protocols, and systems. The lack of use of blockchain to secure the two-way communication in the smart grid has led us to research the ways blockchain has secured communication in other fields and applications to compare the effectiveness.

3.6.1 Performance

Blockchain is discussed in terms of security issues, scalability problems, and other performance metrics. Its performance in various fields is discussed specifically in the area of the Internet of Things.

[106] surveys the security issues that IoT devices tackle. It also reports the different ways blockchain can mitigate certain issues using the blockchain properties such as secure communication, integrity assurance, authentication, authorization, and privacy. [107] describes the environments where blockchain mechanisms play an important role for the Internet of Things. It also describes how blockchain can be used for sensors that don't have encryption or have weak encryption and have weak communication protocols. Using blockchain technology can provide integrity for information transfer, store sensor data, and enable micro-payments.

Security issues in the Internet of Things can be solved using blockchain which will secure communication between devices. Two systems are created with and without blockchain to study the effects of blockchain on IoT security in [108]. It shows that implementing blockchain proves to provide a higher level of security than systems without it. Avalanche effects are observed in relation to the hash algorithm and encryption where blockchain system proved to be more secure.

[109] compares the performance and scalability of a web-based groupware communication application using blockchain and non-blockchain technologies. The system performance is tested using multiple cloud computing configurations. The blockchain implementation scales linearly until a certain threshold and high throughput and low response time can be achieved if enough servers are deployed.

3.6.2 IoT and Communication

Blockchain's communication mechanisms in various fields are discussed specifically in the area of the Internet of Things.

[110] describes how blockchain can be used to solve security and trust issues in the Internet of Things through the insertion of sensor data in blockchain transactions. This provides duplication of the sensors' data in public and distributed ledgers, timestamps, data authentication, and non-repudiation. [111] states that the solution to making IoT systems more secure is the use of blockchain. To provide secure communication, when IoT devices communicate, they will process a transaction and store it in the ledger. In addition, to ensure integrity and confidentiality, communication must be authenticated and encrypted efficiently. The use of a decentralized system provides a peer to peer network and removes the need for a central server. [112] provides a lightweight blockchain-based architecture for smart greenhouse farms to provide security and privacy; in this architecture, a local blockchain is centrally managed by the owner. By adopting blockchain technology, challenges such as decentralization, anonymity, and security in IoT are addressed because blockchain provides increased data transparency and immutability. [113] proposes a security framework for smart devices to provide secure communication using blockchain. [114] explores the opportunities that blockchain provides to secure the IoT communication. Various blockchain platforms are used to develop applications for IoT such as IoTify[16] and IOTA[17]. [115] proposes the use of blockchain's transparent and redundant nature to secure communication. The proposed solution uses smart contracts to verify users' identities and their public keys by mimicking the PKI mechanism. This paper's future work will focus on using blockchain's smart contracts to deal with certificate creation, validation, and storage. [116] proposes the use of blockchain technology and the cloud to secure drone communication for the collection and transmission of data. The system proves to be reliable for a large number of drones in addition to the securing of data integrity and cloud auditing. This thesis resembles work done in this paper to provide data integrity as well as scalability through the use of blockchain and cloud services in unison. [117] proposes the use of smart contracts in the blockchain for the payment of toll fees through vehicle to vehicle communication. [118] describes the use of a smart contract on a private blockchain to store, access, and monitor the data flow in the home. This helps amend privacy issues in smart home systems due to centralization.

[16] https://iotify.org/
[17] https://www.iota.org/

3.6.3 Services

Blockchain's services in various fields are discussed specifically in the area of the Internet of Things.

[119] also describes a blockchain-based PKI solution. It is a certificate system based on a permissioned blockchain. [120] shows how blockchain replaces the need for centralized applications. Blockchain resolves traditional challenges where it is fully distributed and provably secure. Blockchain-based Identity-Based Cryptography (IBC) solutions can use blockchain as a distributed database to resolve traditional IBC problems such as centralization and single point of failure. Blockchain-based Public Key Infrastructure (PKI) solutions are distributed so they don't have a central point of failure. [121] explains that interacting things can authenticate each other and communicate securely through the Ethereum blockchain based on principal of "the friend of my friend is my friend". It ensures that stored information is available for all participating nodes and protected from modifications using blockchain. [122] proposes a data sharing solution for healthcare that uses blockchain technology in addition to a channel formation scheme. Integrity is preserved by using proof of integrity validation which is found on the cloud. [123] proposes the use of blockchain for government services to remove the need for a trusted third party and to create a cheaper alternative while reducing fraud. Governmental activities that can be done on the blockchain include identity management, voting systems, record keeping, healthcare management, and smart cities. [124] also proposes the use of blockchain for ensuring the accountability and integrity of government operations specifically for military use. [125] discusses the use of blockchain for logging application events. [126] discusses the possibility of using blockchain for authentication requirements in distributed environments to overcome the issues encountered with Kerberos. It discusses the strong points that blockchain provides in a general sense. [127] presents a blockchain-based distributed access control system for IoT devices. This paper mentions the issue of scalability in such a system and discusses how it was handled. In this thesis, we deal with the issue of scalability and mitigate it similarly. [128] tackles the use of blockchain to replace the traditional client-server request-response mechanism. This provides the auditability needed for controlled resources.

[129] describes the use of the blockchain smart contracts to facilitate negotiations for trustless transactions. Role based access control is provided through the use of blockchain. [130] describes the use of the ledger to log the electrical market exchange data. [131] describes how to use a blockchain-based solution for electricity trading. The use of smart contracts increases the scale and security of this energy application. [132] proposes a blockchain-based anonymous reputation system to break the link between real identities and public keys. This proposed

method preserves privacy through the use of pseudonyms. The messages are saved on the distributed ledger. [133] proposes a blockchain-based solution for distributed access control systems for the Internet of Things. Smart contracts are used to control devices that post data by creating a transaction between IoT devices and the ethereum network. It is proposed to standardize communication and create a fault-tolerant infrastructure using the Ethereum blockchain. [134] is a decentralized DApp that uses smart contracts on the Ethereum blockchain to share objects. It is fully deterministic and publicly available on blockchain. [135] describes a full distributed access control system for the Internet of Things based on blockchain technology. The simulation results prove blockchain could be used as access management technology in specific scenarios in IoT. [136] proposes the use of blockchain for the election process. Encrypted votes are accumulated in the form of block on each polling booth. This method provides data provenance and ensures the infeasibility of tampering with the votes through the distributed ledger. [137] proposes the use of a blockchain-based architecture for sensory data produced and stored in semi-trusted data storage locations. Blockchain provides trust and a decentralized mechanism for IoT.

In order to ensure a clearer understanding of the security properties and concepts we have implemented in this thesis, the papers mentioned in this section have been organized into tables 3.4, 3.5, 3.6, and 3.7. The occurrence or lack thereof of these security properties and concepts show the difference between these papers and what we have implemented.

Table 3.4: Using Blockchain for Security – Properties (Part 1): Paper - Confidentiality (C) - Privacy (P) - Integrity (I) - Availability (Av)- Authenticity (Au) - Anonymity (An) - Scalability (S)

Paper	C	P	I	Av	Au	An	S
[117]	-	-	Yes	Yes	Yes	-	-
[133]	Yes	Yes	Yes	Yes	Yes	-	-
[134]	-	-	Yes	Yes	Yes	-	-
[118]	-	Yes	Yes	Yes	Yes	-	-
[121]	-	-	Yes	Yes	Yes	-	-
[111]	-	-	Yes	Yes	Yes	-	-
[112]	-	-	Yes	Yes	Yes	-	Yes
[131]	-	-	Yes	Yes	Yes	-	-
[109]	-	-	Yes	Yes	Yes	-	Yes
[135]	-	-	Yes	Yes	Yes	-	Yes
[130]	-	-	Yes	Yes	Yes	Yes	-
[99]	-	-	Yes	Yes	Yes	-	-
[136]	-	-	Yes	Yes	Yes	-	-
[137]	-	-	Yes	Yes	Yes	-	-

Table 3.5: Using Blockchain for Security – Concepts (Part 1): Paper - Blockchain (BC) - Type - Smart Contract (SC) - Simulation - Other Components (Components) - Size - Sector

Paper	BC	Type	SC	Simulation	Components	Size	Sector
[117]	-	Public	Yes	No	-	Big	Toll Payment
[133]	Ethereum	Private	Yes	No	-	Big	Secure IoT Communication
[134]	Ethereum	Public	Yes	No	-	Big	Sharing App
[118]	Ethereum	Public/Private	Yes	No	Cloud	Small	Secure Smart Home
[121]	Ethereum	Public	Yes	Yes	-	Big	Authentication System
[111]	-	-	No	No	Cloud	Big	Secure IoT Communication
[112]	-	Private	No	No	Cloud	Small	Greenhouse Farming
[131]	-	-	Yes	No	-	Big	Distributed Energy Markets
[109]	Tendermint	Private	No	Yes	Database	Big	Communication Protocols
[135]	Ethereum	Private	Yes	Yes	-	Big	IoT Access Management
[130]	-	-	No	No	Cloud	Small	Communication in Microgrid
[99]	-	Private	No	Yes	-	Big	Data Protection in Power
[136]	-	-	No	No	Database	Small	Securing Electoral Voting
[137]	Ethereum	Private	Yes	Yes	Cloud	Big	IoT Data Integrity

Table 3.6: Using Blockchain for Security – Properties (Part 2): Paper - Confidentiality (C) - Privacy (P) - Integrity (I) - Availability (Av)- Authenticity (Au) - Anonymity (An) - Scalability (S)

Paper	C	P	I	Av	Au	An	S
[113]	-	-	Yes	Yes	Yes	-	-
[122]	-	-	Yes	Yes	Yes	-	Yes
[123]	-	-	Yes	Yes	Yes	-	-
[98]	-	-	Yes	Yes	Yes	-	-
[115]	Yes	-	Yes	Yes	Yes	-	-
[100]	Yes	Yes	Yes	Yes	Yes	-	-
[125]	-	-	Yes	Yes	Yes	-	-
[116]	-	-	Yes	Yes	Yes	-	Yes
[127]	-	-	Yes	Yes	Yes	-	Yes
[124]	Yes	Yes	Yes	Yes	Yes	-	-
[128]	Yes	Yes	Yes	Yes	Yes	-	-
[101]	Yes	-	Yes	Yes	Yes	-	Yes

Table 3.7: Using Blockchain for Security – Concepts (Part 2): Paper - Blockchain (BC) - Type - Smart Contract (SC) - Simulation - Other Components (Components) - Size - Sector

Paper	BC	Type	SC	Simulation	Components	Size	Sector
[113]	Ethereum	Private	Yes	No	-	Big	Securing Smart Cities
[122]	Hyperledger	Private	No	Yes	Cloud	Big	Healthcare App
[123]	-	-	No	No	-	Big	Government Services
[98]	Ethereum	-	Yes	No	-	Big	Meter Management System
[115]	Ethereum	-	Yes	No	-	Big	Secure Communication
[100]	-	-	Yes	No	Database	Big	Secure Communication
[125]	-	-	Yes	No	-	Big	Secure Applications
[116]	-	Public	No	Yes	Cloud	Big	Drone communication
[127]	-	-	No	Yes	Cloud	Big	Access Control System
[124]	-	-	No	No	-	Big	Military Usage
[128]	-	-	No	Yes	-	Big	Audited Communication
[101]	Rainbowchain	-	Yes	Yes	Cloud	Big	Secure Smart Grid

3.7 Cloud Solutions in the Energy Field

The cloud computing technology can be used in the information infrastructure which is currently a centralized structure. Control activities done by the utility are also centralized. Power applications can benefit from the cloud computing technology.[138] discusses using cloud computing in the the power system. The **control center cannot handle the huge amounts of data generated by the grid thereby creating a storage bottleneck**. Dealing with computing in relation to big data will be an issue in a centralized infrastructure. In this paper's layered model the communication layer is secured using encryption and authentication.**This layer thus is secured in terms of confidentiality and authenticity but integrity, availability, and data provenance are not ensured.** Many benefits come from using the cloud infrastructure including distributed power data management, scalable and elastic virtualized resource provision, scalable high-performance computing, security and fault tolerance, cost reduction, and access anytime and anywhere as cited in [138]. **The paper shows the limitation of the centralized control center in storing the data. The need for a decentralized storage infrastructure is evident and it is what we have used in this thesis.**

The application of cloud computing to the power grid has been studied in various studies. [139] describes an algorithm using cloud computing and power grids. It is a dispatch algorithm (incorporated in the cloud dispatch infrastructure) that considers the computational resources on the power grid. [140] explores the limitations of the power grid related to the data management in the control center. This paper proposes a cloud framework for data management in the power grid. [141] describes a demand-response application for the smart grid based on the cloud. [142] describes multilayer services to be used in the smart grid. [143] explains the benefit of cloud computing technology in various applications in the smart grid including monitoring SCADA monitoring. The papers in this section describe the use of the cloud computing technology / cloud platform in different aspects of the smart grid. They do not mention the use of the cloud platform for the safe storage of meter measurement data in the smart grid. We use the cloud platform in this thesis to store the data from the advances metering infrastructure for availability of the data.

[116] mentioned above proposes the use of blockchain technology and the cloud to secure drone communication for the collection and transmission of data. This thesis resembles work done in [116] that aims to provide data integrity in addition to scalability through the usage of blockchain and cloud services in unison. [127] presents a blockchain-based distributed access control system for IoT devices with the use of cloud for storage. In this thesis, we use the cloud platform for a similar purpose.

Chapter 4

Architecture Preface

The two main actors in this work are the utility and the smart meters. The utility servers are on the utility side and the smart meters are on the consumer side. For the grid to be effective, the utility must receive correct smart meter data. If the data is not representative of the actual occurrences, this will result in the utility making the wrong decisions with respect to load balancing, load shedding, demand response, and other functionalities that the utility is in charge of providing and maintaining.

The security in the power system involves both the physical security and the cyber security of the power system. In this thesis, the aim is to secure the cyber security of the power system, specifically the two-way communication flow between the utility and the consumers. The constant flow of electrical information from the consumer to the utility provides malicious users/attackers the opportunity to monitor the information flow. Malicious actors intend to disrupt the system through sending fake loads which could have consequences such as nationwide power outages which may lead to fatalities. If fake loads are projected to the utility, it will make wrong decisions and take wrong actions, the consequences of which can be very severe and affect entire infrastructures. In addition, malicious actors can masquerade as legitimate users to send false data or as the MDMS to send false load balancing commands, access confidential information, and prevent legitimate consumers from accessing the network.

The approach we plan on taking in this thesis is implementing a prevention mechanism instead of detecting the issue once it has occurred and then taking effective action. Anomaly detection can be used to alarm the utility when there are abnormal meter readings reported and can only be done using large amounts of data for training and learning. As for the prevention approach, it depends on putting in place intuitive security measures to avoid any unwanted intrusion on the network. This ensures the security and privacy of the communicated data, and accordingly the network as a whole.

4.1 Blockchain

Using the advanced smart metering system (referred to in subsection 2.1) involves the two-way communication between the consumers and the utility. This communication network will be transferring a huge amount of data (electrical or load balancing). This data should represent the complete information sent or received by either party without any tampering. This data is sensitive and access to it should be by authorized users.

The goal of using the blockchain platform as the prevention approach in the energy sector is to create a communication link between the consumers of energy and the controllers of energy (MDMS). This direct link is secure and simplifies the current energy system. The transactions that occur between the utility and the consumers are related to electrical data consumption or load balancing directives. The use of smart contracts in blockchain allows prosumers[1] to sell their energy without the need for a third party. This smart contract could initiate transactions that would need manual assistance in a traditional grid. This greatly improves the flexibility, security, and cost of neighborhood selling energy. In addition, the blockchain platform provides secure storage space for electrical data and load balancing data on a distributed ledger. The data on this ledger is transparent and tamper proof.

Thus, to ensure the safe transfer of data, this thesis uses the Ethereum blockchain. The details of the blockchain platform and Ethereum specifically are found in sections 2.3 and 2.4 respectively.

[1] Electricity consumers - “sumers” - that produce - “pro” - electricity

4.1.1 Why a Public Blockchain?

In chapter 2, we discuss the different concepts used in this thesis. Section 2.3 discusses the network, ledger, consensus, cryptographic techniques, platforms, types, and security properties of the blockchain platform. Distinguishing differences between the different types of blockchains are noted in subsection 2.3.6. The question of why a public blockchain is chosen as opposed to a private or consortium blockchain comes to mind. To understand the need for a public blockchain, we will discuss the limitations of a private/consortium blockchain.

We discuss both the private and consortium blockchains's limitations, but we only refer to the private blockchain in our discussion.

The private blockchain is closed: a random individual cannot read the distributed ledger or participate in the blockchain consensus mechanism. As such, there are certain rules that determine who can participate in the blockchain network activities. This means that trust is needed. Therefore, the integrity of the private blockchain depends on the credibility of the authorized participants. The private network is built and maintained by specific users. This leads to a certain degree of centralization which we specifically try to avoid in our thesis.

A node must be invited to join based on certain rules. There are rules set up to decide each node's role. These nodes are also known to all in the network. Thus, the user is not anonymous in the private blockchain. Furthermore, with fewer nodes in the private blockchain network, it becomes easier for malicious users to gain control of the network. The private blockchain is more vulnerable to risks of data manipulation. To mine, dedicated miners must partake in consensus. These miners need to be trusted to verify and validate authentic transactions. The validity of records cannot be independently verified so external actors have to trust private blockchain without having any control over the various processes of verification. Any transaction that is made can only be viewed by the participants who know about it in the blockchain network. The private blockchain, as its name states, is not public for everyone to use, and the distributed ledger is not public for anyone to read. Thus, transparency in the private blockchain is restricted.

The public blockchain we use in this thesis encompasses the security properties we require from the blockchain platform. The public blockchain is open: anyone can read the distributed ledger and participate in the blockchain consensus mechanism. Accordingly, there are no rules that determine who can participate in the blockchain network activities. Consequently, no trust is needed. The integrity of the public blockchain does not depend on the credibility of the authorized participants or trusted parties. This leads to the decentralization that we want to

achieve in our thesis.

The greater the number of nodes in the network, the greater the decentralization and the more secure it is. Nodes are provided with incentives to participate in the blockchain activities (mining for instance) which makes the network trustless. Thus, with more nodes in the public blockchain network, it becomes harder for malicious users to gain control of the network. The public blockchain makes the risk of data manipulation almost impossible. To mine, each participant in the whole blockchain network must partake in consensus. This eliminates intermediaries and removes the need for trust. These miners do not need to be trusted to verify and validate authentic transactions since the whole network will be taking part in the consensus mechanism. Any transaction made can be viewed by everyone in the blockchain network. The public blockchain, as its name states, is public for everyone to use and the distributed ledger is public for anyone to read. Thus, transactions on the ledger are not private and transparency in ensured.

For more details about the public/private/consortium blockchains types refer to table 2.2 in subsection 2.3.6.

4.1.2 Why Ethereum?

In chapter 2, we discuss the different concepts used in this thesis. Section 2.3 discusses the network, ledger, consensus, cryptographic techniques, platforms, types, and security properties of the blockchain platform. Distinguishing differences between the different blockchain platforms are noted in subsection 2.3.5. The question of why the Ethereum blockchain is chosen as opposed to another blockchain platform like Bitcoin or Hyperledger comes to mind. To understand the need for the Ethereum blockchain, we will discuss the Bitcoin and Hyperledger blockchains.

Bitcoin allows consumers to buy and sell items, property, or assets without being easily traced. This has made Bitcoin a popular platform for illegal and illicit activities. Hyperledger allows industry-wide collaboration in the process of developing blockchain solutions and distributed ledger-based technology. Therefore, Bitcoin is a digital currency platform and Hyperledger is a blockchain implementation, testing, and deployment platform. Both of these platforms do not achieve this thesis's goal which is to secure the two-way communication between the utility and the smart meters.

Ethereum is an open source, public, blockchain-based distributed computing platform and operating system featuring smart contract functionality as discussed in section 2.4. It allows entities or companies to build and deploy decentralized applications. These applications could belong to many different sectors. Ethereum allows us to achieve secure two-way communication between the utility and the smart meters, according to the security properties it achieves which are discussed in subsection 2.4.10.

For more details about the Bitcoin/Ethereum/Hyperledger blockchains platforms refer to table 2.1 in subsection 2.3.5.

4.2 Smart Grid Communication Network

The smart grid is composed of various entities, as indicated in figure 4.1. As in the traditional grid, the electricity is generated, transmitted and distributed to the consumer. Electricity in the traditional grid is generated in traditional power plants (thermal or nuclear), transmitted, and distributed. In addition to the traditional generation methods, the smart grid includes the generation of electricity by renewable energy resources such as renewable power plants (solar, wind, hydraulic, or geothermal). Electricity may also be generated in the smart grid by consumers that use renewable energy resources. Consumers that produce electricity are known as prosumers. The integration of renewable energies allows the creation of microgrids - small electric networks that permit close-by consumers to produce and sell electricity. The electricity generated by different sources is, in turn, distributed to different consumers. The consumers that do not contain their own source of electricity (a.k.a renewable energy resources) need to have electricity delivered. Electric vehicles need to be charged. In all three modes of generation, the electricity produced is transmitted and distributed as seen by the red arrows in figure 4.1.

The traditional grid only deals with a flow of electricity in which it is usually generated in traditional power plants, transmitted, and distributed. The smart grid, on the other hand, deals with a flow of electricity in which it can be generated by various entities, transmitted, and distributed. It also contains two-way communication between entities. The entities we considered while formulating our approach are the smart meters, located at the consumer establishments, and the utility, the smart grid control center. Data is expected to flow both ways between the utility and the smart meters. The consumers send the utility electrical data for reporting purposes and the utility sends the consumers load balancing data in order to manage electricity demand. The electrical data sent to the utility and the load balancing data sent to the consumers encompass the two-way communication as seen by the green arrows in figure 4.1.

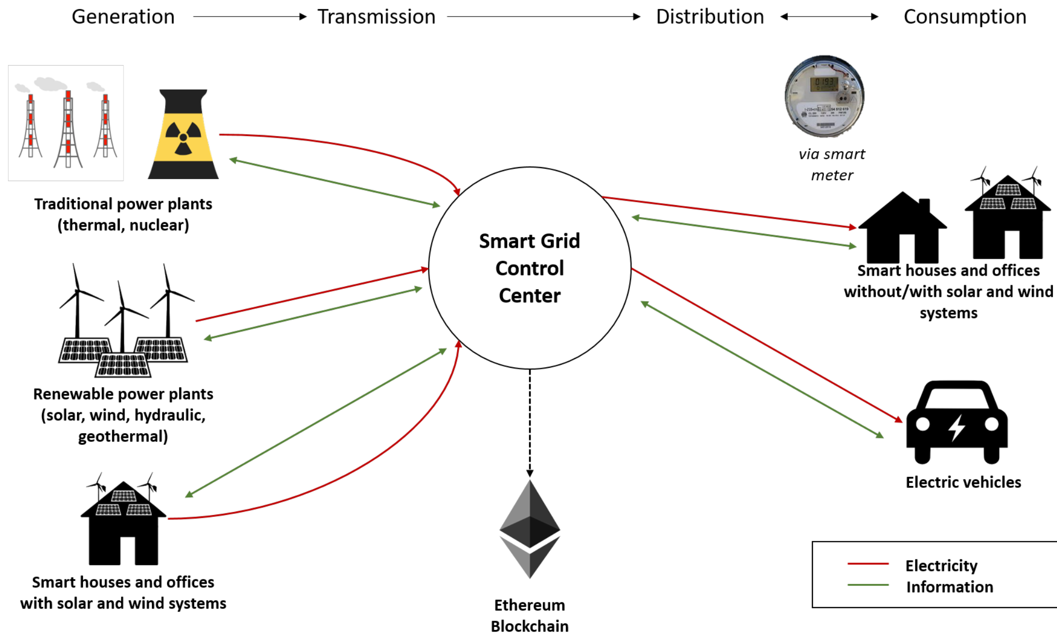

Figure 4.1: General Layout of Smart Grid Communication System

We propose the use of the Ethereum blockchain as the underlying technology in addition to some cryptographic techniques to secure the two-way communication between the utility and the smart meters. As shown in figure 4.1, the smart grid control center, otherwise referred to as the utility, manages the distribution of electricity and interacts with the Ethereum blockchain to monitor consumers' electric consumption and manage it accordingly. Ethereum implements the concept of smart contracts, which we will use extensively, in order to ensure that our suggested architectures behave securely and are able to mitigate threats on the smart grid metering communication. Architecture 1 in chapter 6 and architecture 2 in chapters 7 and 8 will deal with a different number of smart contracts interacting in various ways but will ultimately result in the same end-goal which is a secure smart grid communication network. In our work, many trade-offs will be taken into consideration and choices will be made based on the architecture that best suits the market and the current needs of new technologies in the smart grid. In chapters 6, 7, and 8 we discuss the architectures proposed to deal with securing the smart meter - utility communication.

Chapter 5

Simulation Environment

Simulation in this thesis will help show the interaction of the consumer and utility with the smart contracts deployed in our proposed system. We are using the Ethereum blockchain to deploy our smart contracts. The point of this simulation is to see the interaction between the consumer, the utility, and the smart contracts deployed to provide secure communication between them in order to relay electrical data consumption needed for DR, DSM, and load balancing (explained in section 2.2). We use Metamask in this thesis as well as JavaScript/React libraries to provide the user with an interface to interact with the created smart contracts. Figure 5.1 shows the interaction of the various tools in our simulation.

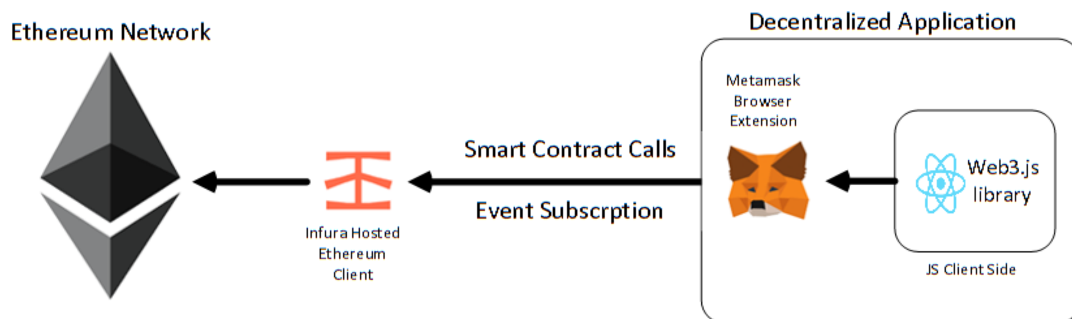

Figure 5.1: Simulation Tools Interaction

5.1 JavaScript-based Tools

Through the use of JavaScript, a high-level, just-in-time compiled, multi-paradigm language, we create the necessary files to compile, test, deploy, and interact with smart contracts. The JS files written for compiling and deploying the smart contracts will be explained in subsections 5.3 and 5.5 respectively, and the details concerning the various smart contracts can be found in chapters 6, 7, and 8. JS files are also written for testing smart contracts, which is explained in subsection 5.4 using a JS framework known as Mocha.

Web3[1] is an important collection of JS libraries we used to interact with remote or local Ethereum nodes. It allows the client to interact with the Ethereum blockchain. It also allows the client to send Ether, deploy smart contracts, and interact with smart contracts. Web3 must first be installed, and then an instance of it can be created. The instance of web3 will be injected into the browser using Metamask. Web3 will need a provider that is running a geth node that talks to the Ethereum network. In this thesis, the provider used is Infura[2] which is a hosted Ethereum node cluster, to run applications without the need for an Ethereum node. This provides the client access to a remote Ethereum node instead of having to run our own Ethereum node which requires a lot of data download and maintenance.

React[3], a JS library, is used to create interactive user interfaces with ease. React works with the concept of components that contain their own state updating and rendering the state of variables defined and used. Components may interact making the passing of data from one to the other simple. We use the React library to create components for the different smart contract interactions (calls and updates to the Ethereum Network).

Using Node.js[4], a cross-platform run-time environment, we can run JS code on a server. For our simulation, we run the code on localhost using port 3000. Using Node.js, we can host the front-end application created using React, which will be seen in sections 6.3, 7.3 and 8.4. We also use Next.js[5] for an intuitive page routing system, allowing dynamic page routing which in turn allows the creation of dynamic URLs.

[1] https://web3js.readthedocs.io/en/v1.2.6/
[2] https://infura.io/
[3] https://reactjs.org/
[4] https://nodejs.org/en/about/
[5] https://nextjs.org/docs

5.2 Metamask

MetaMask provides a secure and simple way to connect to decentralized applications on the blockchain network. DApps may be built using technologies that some browsers cannot support. Metamask handles this and provides the users with a set of accounts that are secured using a seed phrase. A seed phrase is a recovery phrase that contains the information that allows the generation of the private keys for the accounts generated by Metamask. This seed phrase must be backed up and stored securely since any external access to it compromises the accounts. In addition, this seed phrase can be used to restore the Metamask account at any time. Moreover, Metamask supports hardware wallets that protect the user's funds. Through Metamask, cryptocurrencies are managed in addition to tokens or collectibles.

Most importantly, Metamask allows the users to interact with decentralized applications through smart contract calls and updates. Metamask has access to the Main Ethereum Network and its various Test Networks. The user may generate multiple accounts, each with its own Ethereum address and associated private key, and use these to deposit or send Ether onto the network and interact with smart contracts.

5.2.1 Main Ethereum Network

The mainnet is a fully developed blockchain protocol. It is deployed onto the network where the cryptocurrency transactions go through the process of being broadcast, verified, and recorded on a distributed ledger. Users can send and receive transactions related to cryptocurrency or digital data. The mainnet network is used for real transactions containing real monetary value meaning the currency is real whereas testnets are used for testing smart contracts and DApps in an environment where the currency has no value.

5.2.2 Rinkeby Test Network

Testnets, a.k.a test networks, are used to test software. These testnets are usually used by developers to ensure that the software or application is working as desired. It functions the same way as the mainnet. By containing computers running nodes, DApps deployed onto it, and smart contracts running, it simulates the real-world environment. In Ethereum, there are several known testnets: Ropsten Test Network, Kovan Test Network, and Rinkeby Test Network, which we use in this thesis. The value of the currency on these testnets in non-existent. The ether used for the transactions on these testnets can be retrieved from ether faucets.

Rinkeby is an Ethereum test network. Rinkeby testnet is a proof-of-authority network unlike the Ethereum mainnet that uses proof-of work network. It is immune to spam attacks, unlike the Ropsten Test Network, making the Ropsten Test Network less stable. The ether needed for these transactions can be obtained, as mentioned, from a Rinkeby ether faucet[6]. As we know, ether is used to pay the transaction fees that come with sending transactions in the Ethereum network (as discussed in subsection 2.4.9).

[6] https://faucet.rinkeby.io/

5.3 Compiling Smart Contracts

The smart contracts written in the Solidity language are saved in a file with extension ".sol". These contracts need to be compiled, tested, and then deployed for use. To compile to a contract object and get the necessary JSON file for deployment, the solidity compiler must be installed as well as other steps shown in listing 5.1. Once compiled, the code will produce a JSON file per smart contract containing the interface and ABI (explained in subsection 2.4.2) needed for deployment. The "compile" file in listing 5.1 is that of the second architecture in chapters 7 or 8.

```
const path  = require("path"); //getting paths
const solc = require("solc"); //compiler for solidity
const fs = require("fs-extra"); //file manipulation

//if build folder exists delete it
//find the directory and delete it
const buildPath = path.resolve(__dirname, "build");
fs.removeSync(buildPath);

//find directory where contract is and get path
const simulationPath = path.resolve(__dirname, "contracts",
    "Simulation.sol");

//get the source code of file
const source  = fs.readFileSync(simulationPath, "utf8");

//provides the json for our contracts
const output = solc.compile(source, 1).contracts;

//create build directory if folder doesn't already exist
fs.ensureDirSync(buildPath);

//loop over all contracts and write to files
for(let contract in output) {
    fs.outputJSONSync(
        path.resolve(buildPath, contract.replace(":","")+".json"),
        output[contract]
    );
}
```

Listing 5.1: Compile Contract

5.4 Testing Smart Contracts

Remix is an IDE that allows users to write Solidity in the browser. It supports testing, debugging, and deploying smart contracts. Due to its simplicity, we used Remix to test the smart contracts we created. The contract is written out and saved in an ".sol" file. It is then deployed using one of the many Ethereum addresses Remix provides. Function calls and updates can be made using the easy-to-use interface. Events can be viewed by opening the logs and the gas usage can be seen as well. We also used Ganache, Mocha, and Web3 to test more extensively. Ganache provides us with a set of unlocked accounts; accounts that do not need to use public or private keys to access. Ganache is an Ethereum node Emulator. Mocha is a JS test framework that runs on Node.js. Finally, Web3 is used to interact with Ethereum nodes. So using Ganache, we can connect to a local node.

The smart contracts written in the Solidity language have now been compiled and are ready to be tested. To test to a contract, we use both methods, Remix and Ganache/Mocha/Web3. The code shown in listing 5.2 is using the latter. In the small example, two concepts are being tested. The first is whether or not the contract was deployed successfully - lines 28 to 30. The second is whether or not the user who deployed the contract is the utility - lines 32 to 35. Figure 5.2 is an example run of the code in listing 5.2. The "test" file in listing 5.2 is that of the second architecture in chapters 7 or 8.

```
1 const assert = require("assert"); //make assertions about deployed
    contract
2 const ganache = require("ganache-cli");
3 const Web3 = require("web3"); //constructor to create instance of
    web3 library
4 const web3 = new Web3(ganache.provider()); //create instance and
    try to connect to network
5 web3.currentProvider.setMaxListeners(300);
6
7 //get access to compiled files
8 const compiledSimulation =
    require("../ethereum/build/Simulation.json");
9
10 let accounts;
11 let simulation;
12 let utility;
13
14 //deploy contract
15 beforeEach(async() => {
16     accounts = await web3.eth.getAccounts();
17     utility = accounts[0];
18
19     //deploy factory contract
20     simulation = await new
        web3.eth.Contract(JSON.parse(compiledSimulation.interface))
21     .deploy({data: compiledSimulation.bytecode})
22     .send({from: utility, gas: "1100000"});
23 });
24
25 describe("Simulation", () => {
26
27     //checking if the contract is deployed successfully
28     it("deploys contract", () => {
29         assert.ok(simulation.options.address);
30     });
31
32     it("caller is utility", async () => {
33         const isUtility = await simulation.methods.utility().call();
34         assert.equal(utility, isUtility);
35     });
36
37 });
```

Listing 5.2: Test Contract

```
D:\AUB\Thesis\Ethereum and Solidity\simulation>npm run test

> simulation@1.0.0 test D:\AUB\Thesis\Ethereum and Solidity\simulation
> mocha



  Simulation
    √ deploys contract
    √ caller is utility


  2 passing (247ms)
```

Figure 5.2: Testing Smart Contract Functionality

5.5 Deploying Smart Contracts

The smart contracts have now been compiled and tested and are ready to be deployed. To deploy a contract, the JSON files produced in section 5.1 can be used by extracting the interface and ABI as shown in listing 5.3 in addition to other steps (smart contract concepts are explained in detail in subsection 2.4.2). Once deployed, the code will produce an Ethereum contract address per contract which is used to access the instance of the contract. In the example, there are two contracts being deployed by the utility. The "deploy" file in listing 5.3 is that of the second architecture in chapters 7 and 8.

```
1 const HDWalletProvider = require("truffle-hdwallet-provider");
2 const Web3 = require("web3");
3
4 //get access to compiled code
5 const compiledSimulation =
     require("../ethereum/build/Simulation.json");
6 const compiledCommunication =
     require("../ethereum/build/Communication.json");
7
8 //provide mnemonic and Rinkeby API
9 const provider = new HDWalletProvider("mnemonic", "rinkeby api");
10
11 //connect to provider and get an instance of web3 that is enabled
     for Rinkeby network (unlocked account)
12 const web3 = new Web3(provider);
13
14 //code
15 const deploy = async () => {
16     const accounts = await web3.eth.getAccounts();
17     const utility = accounts[0];
18     console.log("account used: ", utility);
19
20     //for simulation contract
21     const result1 = await new
         web3.eth.Contract(JSON.parse(compiledSimulation.interface))
22     .deploy({data: "0x" + compiledSimulation.bytecode})
23     .send({from: utility})
24
25     //address of the simulation contract
26     console.log("address: ",result1.options.address);
27
28     //for communication contract
29     const result2 = await new
         web3.eth.Contract(JSON.parse(compiledCommunication.interface))
30     .deploy({data: "0x" + compiledCommunication.bytecode,
         arguments: [result1.options.address]})
31     .send({from: utility})
32
33     //address of the communication contract
34     console.log("address: ",result2.options.address);
35 };
36
37 deploy();
```

Listing 5.3: Deploy Contract

The smart contracts have now been compiled, tested, and deployed. To get the instance of a contract, the JSON files produced in section 5.1 can be used by extracting the interface and ABI. The interface and contract address produced in listing 5.3 are used to refer to the Communication contract instance. This will be used in the rest of the user interface code implemented in sections 7.3 and 8.4. In the example, there is an instance of one of the deployed contracts. The "deploy" file instance in listing 5.4 is that of the second architecture in chapters 7 and 8.

```
1 import web3 from './web3';
2 import Communication from './build/Communication.json';
3
4 //contract instance
5 const instance = new web3.eth.Contract(
6     JSON.parse(Communication.interface),
7     'contract address'
8 );
9
10 export default instance;
```

Listing 5.4: Deployed Contract Instance

5.6 Observing Smart Contracts

To understand what occurs when transactions are submitted to the Ethereum network, Etherscan[7] is used. Etherscan is a block explorer. Users that are interacting with the Ethereum Blockchain Network can access Etherscan to find transactions that have occurred. It is a search engine that provides users with the ability to search for Ethereum addresses (contract or externally owned) and see their corresponding transactions. This fortifies the notion of transparency in blockchain. Etherscan has many uses including monitoring gas price and monitoring statistics, but in this thesis it is used to monitor the transaction fees and gas consumed; both of these concepts are discussed in subsections 2.4.9 and 2.4.7 respectively. We use Etherscan to track transactions and observe our smart contract interactions.

[7] https://etherscan.io/

5.7 Subscription to Smart Contract Events

A subscription is when a customer pays for a certain commodity at regular intervals. Subcriptions, using the Web3.j libraries explained in subsection 5.1, are possible for many different Ethereum blockchain functionalities. An Ethereum user can subscribe to events/logs, pending transactions, newly added block headers, and syncing events (in the process of being added). In this thesis, we deal with Ethereum events that are explained in subsection 2.4.8. The "web3.eth.subscribe" function allows us to subscribe to particular events in the Ethereum blockchain. The structure of the subscription is explained in detail to facilitate the work explained in subsections 7.3 and 8.4. Listing 5.5 shows an example of the structure of a subscription.

```
1 web3.eth.subscribe('logs', {
2     address: ....,
3     topics:
        [web3.utils.sha3("NameOfEvent(parameter1,parameter2)"),...]
4 }, function(error, result) {
5     if (!error)
6         console.log(result);
7 });
```

Listing 5.5: Subscription to Smart Contract Event

The subscription is on the logs - where the event data is stored. We can select from which blockchain block, using the "fromBlock" parameter. The function takes the optional "address" and "topics" parameters. The address array parameter is the only list of addresses from which events are to be accepted. The topics array parameter is the list of values that must be present to accept the event.

We only use the topics parameter, which was explained in subsection 2.4.8, when discussing events and the indexing of their parameters. When an event is created it will always contain one indexed parameter, which is the signature of the event called, and it can contain other parameters. Of these parameters, only two more can be indexed. Indexed parameters are parameters on which the event can be filtered. In the example in listing 5.5, the topics array contains the one default indexed parameter which is the signature of the event - line 2.

If an event is found with the correct filters applied and no error, the resulting subscription instance will be shown in the console - line 5. In this thesis, we work with the returned subscription instance, as is demonstrated in sections 7.3 and 8.4.

5.7.1 Subscribed Events in the Front-End

When the React component is loaded in the web browser, we begin listening for either of the two events emitted by our smart contracts. Events are explained in subsection 2.4.8. The utility will listen for the event containing the electrical data and the consumer will listen for the event containing the load balancing data as explained in subsection 7.2.2 listings 7.8 and 7.9 respectively. The details and code showing the interaction with the events and subscription to the events will be shown in chapters 7 and 8 subsections 7.3.2 and 8.4.2 respectively.

If an event is detected, the component will reload and render the respective tables showing the data the events contained. The whole process is known as event sourcing. The events will be filtered based on what each user needs. For the utility, the event containing the electrical data will be filtered by the name of the event: “DataSent” in addition to the optional desired timestamp. For the consumer, the event containing the load balancing data will be filtered by the name of the event: “LoadBalancingSent” in addition to the consumer’s Ethereum address.

5.8 Public Key Management

The confidentiality and privacy of the consumer's electrical data and the utility's load balancing data is of the utmost importance in this thesis. The combination of the Ethereum blockchain platform and cryptographic tools ensure this security property. Confidentiality is usually ensured through the use of symmetric or asymmetric encryption as seen in chapter 3. In this thesis, specifically chapters 7 and 8, we use asymmetric cryptography, combined with the Ethereum platform alone or with the Ethereum platform in addition to the cloud platform respectively.

In this thesis, we use Rivest-Shamir-Adelman (RSA) asymmetric encryption. With the help of a Node.js package named node-rsa[8], we produce the code shown in listing 5.6 that describes the steps taken to produce the public-private key pair needed for encryption in architecture 2. The PEM format or base64 format the keys are exported to is encoded in base64 with additional header and footer lines "—–BEGIN PUBLIC KEY—–" and the corresponding "—–END PUBLIC KEY—–" for the public keys and "—–BEGIN PRIVATE KEY—–" and the corresponding "—–END PRIVATE KEY—–" for the private keys. This PEM format allows us to add both the utility and consumers' public keys to the first smart contract as will be seen in sections 7.3 and 8.4.

```
1  //in class named getKeys
2  function keys () {
3      const NodeRSA = require('node-rsa');
4      const key = new NodeRSA({b:512});
5
6      //export the public and private keys from the key generated
7      //in PEM format so it is readable (base64)
8      this.publicK = key.exportKey('pkcs8-public-pem');
9      this.privateK = key.exportKey('pkcs8-private-pem');
10 };
11
12 //export keys
13 module.exports = keys
```

Listing 5.6: Public-Private Key Creation

[8]https://www.npmjs.com/package/node-rsa

5.8.1 Key Management for Electrical Data (Utility Keys)

The utility will need to create a public-private key pair. This will be done once and will occur before the smart contracts are deployed. The utility can then include its public key in the contract for all the nodes to use, as will be seen in sections 7.3 and 8.4. The private key will be kept locally and safeguarded in the utility's smart meter.

The consumer will use the utility's public key found in the first smart contract to encrypt the electrical data it sends to utility. The JS class, in listing 5.7, is used by the consumers to get the utility's public key in the appropriate RSA format[9] - line 17. The asynchronous[10] transaction - line 8 - calls the smart contract getter function for the constant public key variable "publicKey" (as discussed in subsection 7.2.1 listing 7.1). Once the promise is resolved - line 15 - the utility's public key will be retrieved in the PEM format.

```
1 const NodeRSA = require('node-rsa');
2 const key = new NodeRSA();
3
4 const get = async () => {
5
6     //call the smart contract getter of public key variable
7     //0 params
8     const publicKey =  await simulation.methods.publicKey().call();
9
10    //the asynchronous function call needs to be resolved
11    var promise1 = Promise.resolve(publicKey);
12
13    //we wait for the promise to be resolved
14    //when it is we get value of the public key
15    promise1.then(function(value) {
16        //import the public key in pkc8 format
17        key.importKey(publicKey, 'pkcs8-public');
18    });
19 }
20
21 get();
22 export default key;
```

Listing 5.7: Get Utility Public Key

[9]pkcs8: one of the standards in Public-Key Cryptography Standards (PKCS) created by RSA Laboratories

[10]The async keyword is used in the function definition to declare it is asynchronous. The await keyword used to show the action of waiting for a promise. The function is paused in a non-blocking way until the promise settles. The value is returned if the promise fulfills.

The utility will use the its private key found locally to decrypt the electrical data sent. The JS class in listing 5.8 is used by the utility to get their private key in the appropriate RSA format - line 8.

```
1 //found locally
2 const privateKey = "utility's private key";
3
4 const NodeRSA = require('node-rsa');
5 const key = new NodeRSA();
6
7 //import the private key in pkc8 format
8 key.importKey(privateKey, 'pkcs8-private');
9
10 export default key;
```

Listing 5.8: Get Utility Private Key

5.8.2 Key Management for Load Balancing Data (Consumer Keys)

Each consumer in the network will need to create a public-private key pair. This will be done once. Each consumer's public key will be distributed through the first smart contract as will be seen in sections 7.3 and 8.4. The private key will be kept locally and safeguarded in the consumer's smart meter.

The utility will use the consumer's public key found through the first smart contract to encrypt the load balancing data it sends to consumers. The JS class in listing 5.9 is used by the utility to get the consumer's public key in the appropriate RSA format - line 8. The passed argument is the PEM format of the key available through the first smart contract.

```
1 var keys = function (given) {
2     const NodeRSA = require('node-rsa');
3     const key = new NodeRSA();
4 
5     const publicKey = given;
6 
7     //import the public key in pkc8 format
8     key.importKey(publicKey, 'pkcs8-public');
9     this.key = key;
10 };
11 
12 module.exports = keys
```

Listing 5.9: Get Consumer Public Key

The consumer will use his/her private key found locally to decrypt the load balancing data sent. The JS class in listing 5.10 is used by the consumer to get their private key in the appropriate RSA format - line 8. The passed argument is the PEM format of the key available locally.

```
1 var keys = function (given) {
2     const NodeRSA = require('node-rsa');
3     const key = new NodeRSA();
4 
5     const privateKey = given;
6 
7     //import the private key in pkc8 format
8     key.importKey(privateKey, 'pkcs8-private');
9     this.key = key;
10 };
11 
12 module.exports = keys
```

Listing 5.10: Get Consumer Private Key

Chapter 6

Architecture 1

We propose to create three separate smart contracts each for a different functionality. The first smart contract is the one that allows consumers to join the smart grid communication network. It is called the simulation smart contract. The second smart contract is the contract deployed by the utility to allow consumers to create a customized smart contract that organizes consumer-utility communication. It is called the factory smart contract. The third smart contract is the contract that defines the protocols used in the transmission of data from the consumer to the utility and vice versa. It is called the data smart contract. These smart contracts have been carefully designed to reduce the cost of using them (to understand the concept of smart contract fees and transaction costs refer to subsections 2.4.2 and 2.4.9). This chapter shows the scenarios the architecture will follow in section 6.1, the smart contract details in section 6.2, the user interface used to interact with the smart contracts in section 6.3, the results in section 6.4, the security properties achieved in section 6.5, and the limitations of this architecture in section 6.6.

This proof of concept architecture, along with the corresponding simulations and results, have been accepted at the 16th International Wireless Communications and Mobile Computing Conference (IWCMC 2020), scheduled to be held at Limassol, Cyprus, June 15 - 19, 2020. Due to the COVID-19 pandemic, the conference will be held virtually this year.

> *R. Akhras, W. El-Hajj, M. Majdalani, H. Hajj, R. Jabr, and K. Shaban, "Securing Smart Grid Communication using Ethereum Smart Contracts", IEEE International Wireless Communications and Mobile Computing Conference (IWCMC 2020), Limassol, Cyprus, June 15-19, 2020.*

6.1 Scenario

Using a control flow graph (CFG), we display the order of execution of the calls and updates to the smart contract. The CFG[1] shows all possible steps that can be executed in our program.

6.1.1 Initial Contract Deployment

Figure 6.1 shows the steps taken by the utility to deploy the smart contract needed to join the smart grid communication network and the smart contract needed to deploy the data contract. The utility initially deploys the simulation contract that would lead to a transaction that would update the blockchain state (step 1). This contract deals with the consumers requesting entry and the utility accepting consumers into the network. The utility also deploys the factory contract that would lead to a transaction which would update the blockchain state (step 2). This contract allows consumers to deploy their own data smart contracts.

Deploying the simulation contract and the factory contract lead to changes in the Ethereum state.

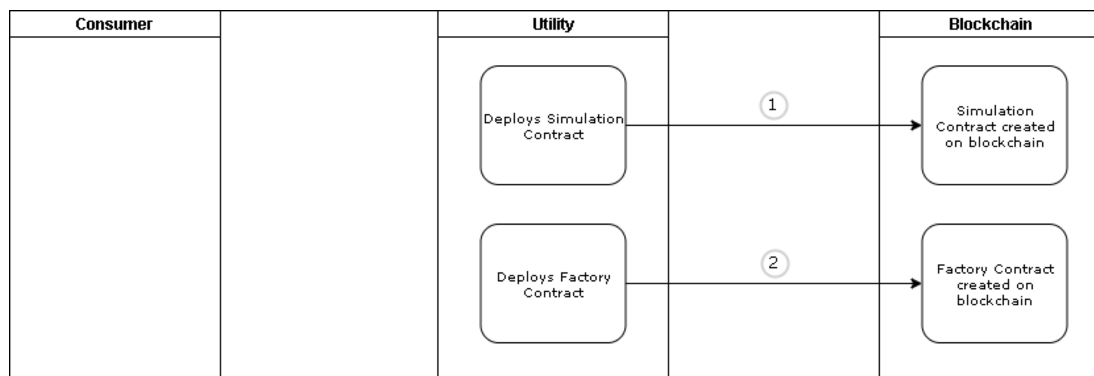

Figure 6.1: Architecture 1: Initial Contract Deployment

[1]https://en.wikipedia.org/wiki/Control-flow_graph

6.1.2 Contract Interaction

Figure 6.2 shows the steps taken by the consumer to try and enter the network. The consumer can request entry that would lead to a transaction that would update the blockchain state (step 1).

Figure 6.2 also shows the steps taken by the utility to accept or reject the consumer. The utility queries for requests from consumers. The utility finds a request (step 2). If there is a request and the utility wishes to reject the consumer, then it would lead back to a state of waiting for a request. For instance, the utility can choose to reject a consumer if they did not register with the utility beforehand. If there is a request and the utility wishes to accept the consumer, then it would lead to a transaction that would update the blockchain state (step 3). If the utility does not find a request, it waits 15 minutes before rechecking (step 4). The 15 minute wait can be updated to 30 minutes or an hour depending what the utility wants. The 15 minute intervals will recur throughout the architecture and can be updated to 30 minutes or an hour also depending on what the utility wants. The utility can also remove a consumer from the network that would lead to a transaction that would update the blockchain state (step 5).

Requesting, accepting and removing consumers lead to changes in the Ethereum state. Querying for requests does not lead to changes in the Ethereum state.

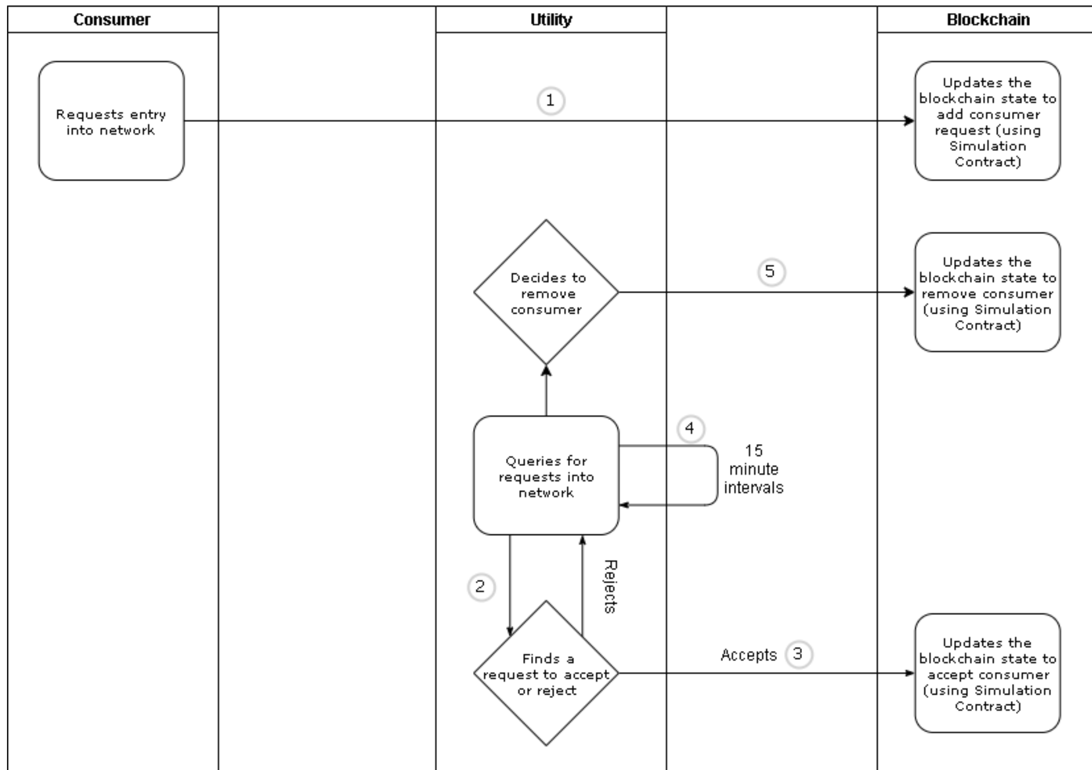

Figure 6.2: Architecture 1: Contract Interaction (Part 1)

Figure 6.3 shows the steps taken by both the utility and the consumer to send data and get data. Electrical data can be sent from the consumer to the utility. This electrical data can be received by the utility. The consumer may not send any data till it has deployed its individual data contract. The consumer must query at 15 minute intervals for acceptance into the network (step 4). If the consumer can send data (step 1), then the consumer would deploy its individual data contract that would lead to a transaction which would update the blockchain state (step 2). It would also lead to a transaction that would update the blockchain state (step 3).

Deploying the data contract and marking the consumer as having deployed the data contract lead to changes in the Ethereum state. Querying for ability to send data does not lead to changes in the Ethereum state.

Every 15 minutes the consumer sends electrical data (step 6) that would lead to a transaction which would update the blockchain state (step 5). Every 15 minutes the utility will check for electrical data (step 7).

The consumer sending data leads to changes in the Ethereum state. The utility querying for data (receiving) does not lead to changes in the Ethereum state.

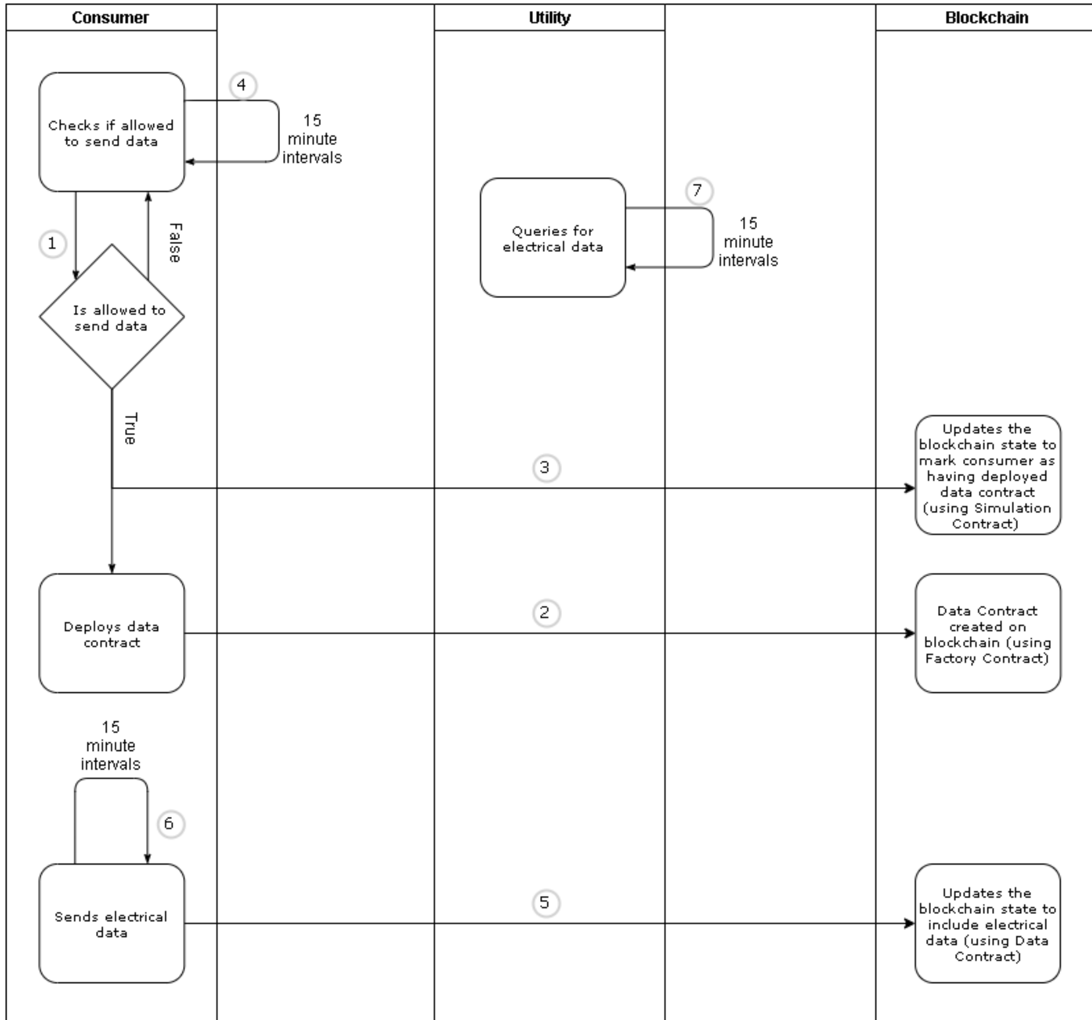

Figure 6.3: Architecture 1: Contract Interaction (Part 2)

6.2 Smart Contract Details

Now that we have gone through the control flow of the smart contracts' executions, we answer next the following questions for each smart contract: By who was the contract deployed? When was it deployed? For what purpose? What are the contracts' contents?

6.2.1 First Smart Contract - Joining the Network

This contract is deployed by the utility only once at inception. Figure 6.4 shows a high level view of the smart grid communication network in relation to the first smart contract. This figure illustrates a city in Qatar that contains 10 households/establishments (HHs) numbered 1 through 10 and the utility numbered 11. All of the establishments numbered 1 through 10 contain smart meters that measure their electrical consumption.

There are two distinguishable components in figure 6.4. The first component is the blockchain network. This network contains all the households and the utility. The HHs and the utility are part of the peer-to-peer Ethereum blockchain network as illustrated by the road which connects all the HHs and establishments. The HHs are nodes in the blockchain network and have access to the distributed blockchain ledger. Node 11, the utility, contains the first smart contract known as the simulation smart contract. Once deployed at the inception of the smart grid communication network, the smart contract becomes accessible by all other nodes on the blockchain since it is now part of the blockchain's ledger. This smart contract is deployed to allow the consumers to join the network, composed of the different consumers that contain legal smart meters registered at the utility.

The second component is the distributed ledger. This is the Ethereum ledger to which every node will have access. This ledger will contain all the transactions that have been mined and added to the blockchain. The transactions include all the updates to the Ethereum state (explained in subsection 2.4.5).

The symbol table explains that the different nodes 1 through 11 can send a transaction "T" at a certain time "t" that is added to the distributed ledger. The legend displays the objects that exist in the smart grid communication network.

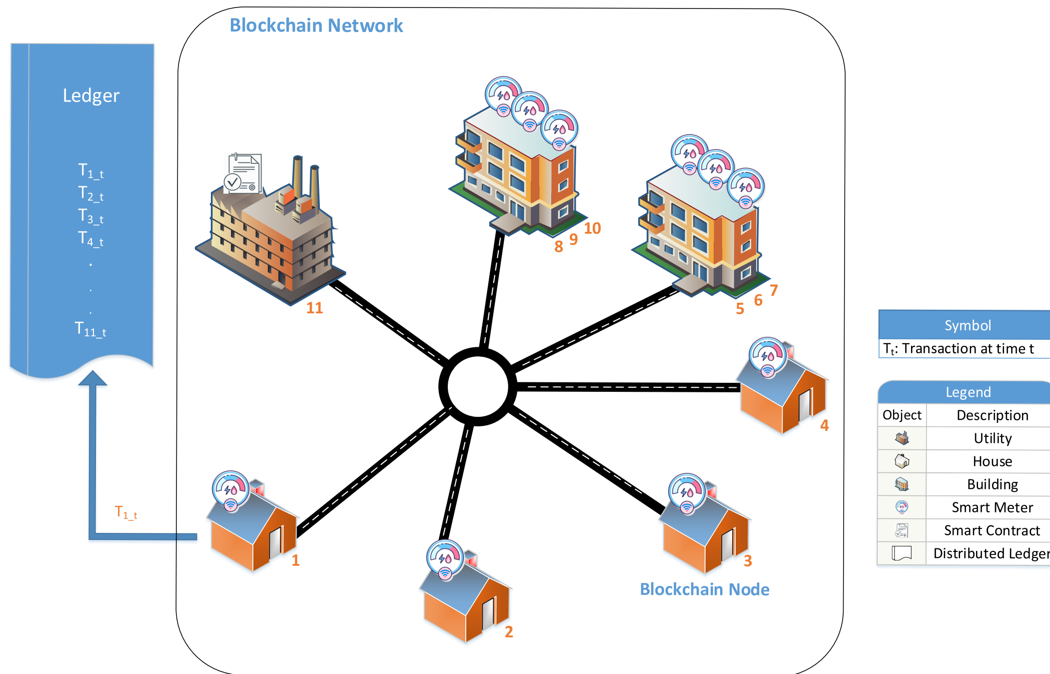

Figure 6.4: Architecture 1: Smart Contract #1 Distribution

As we've explained in subsection 2.4.2, variables are set up to be interacted with and functions exist to allow actions to occur on the blockchain. The detailed code of the first smart contract variables and functions is available in Appendix A listing A.1. Here, we will briefly explain the variables, functions, and their purpose in the smart contract.

Listing 6.1 displays the variable names and descriptions in this contract. These variables are used in functions we discuss below.

- **Line 1 - utility**: Utility's Ethereum address
- **Line 2 - countNodes**: Number of consumers accepted onto the network
- **Lines 5 till 10 - Node**: Node structure that contains properties such as smart meter id, mac address, Ethereum address, and contract completion attribute - lines 6, 7, 8, 9 respectively. It will be used to create requests by consumers
- **Line 11 - requestedNodes**: Array of requesters (represented by the node structure)
- **Line 13 - hasRequested**: Map of requested consumers which are consumers that want to enter the network

- **Line 14 - isDeployed**: Map of deployed consumers which are consumers that have been accepted onto the network
- **Line 15 - hasContract**: Map of consumers who have deployed their data contract (to be discussed in the subsection 6.2.3)

The array of requesters will be used by the utility to display the requesters in the user interface whereas the mapping of requesters will be used by the smart contract to ensure that the requester cannot request entry again (security measure).

```
address public utility;
uint public countNodes;

//request for node
struct Node {
    string smId;
    string mac;
    address add;
    bool complete;
}
Node[] public requestedNodes; //to be added to network by utility

mapping(address => bool) hasRequested;
mapping(address => bool) isDeployed;
mapping(address => bool) hasContract;
```

Listing 6.1: Variables

The functions included in this smart contract are listed below and will be explained in listings 6.2, 6.3, 6.4, 6.5, 6.6, 6.7, and 6.8 respectively.

- Constructor
- Request entry
- Accept node
- Remove node
- Get requesters count
- Can deploy
- Mark as done

We assume that the smart grid communication network is starting with no consumers (smart meters). The "constructor" function - line 3 - initializes the variables in listing 6.1 as seen in listing 6.2. The Ethereum utility address is set to the caller of the constructor function "msg.sender" which is the utility - line 4. The number of nodes is set to 0 since we assume the network is starting with no consumers - line 5.

```
//only called once by utility
//since deployed by utility
function Simulation() public {
    utility = msg.sender;
    countNodes = 0;
}
```

Listing 6.2: Constructor

Via this smart contract, the consumer (smart meter) must request entry into the network to become part of it. Using the "requestEntry" function - line 3 - shown in listing 6.3, the consumers can request to enter the network providing certain properties to be approved by utility. Before the function can be completed, certain rules must be checked. The caller of the function must not be the utility since this function is not used by the utility - line 5. The function must not be called by a consumer that has previously requested entry (called this function) - line 6 - and must also not be called by a consumer that is currently a consumer in the network (accepted by utility) - line 7. These restrictions force the function to be called only when necessary thereby alleviating any problems with overusing this function and stressing the network. If any of these conditions are not satisfied, the function will throw an exception which we will further discuss in subsection 6.3.1. If they are satisfied, a node structure is created with the passed properties - line 9. The consumer's node is now added to the mapping of requesters - line 10 - and array of requesters - line 11 - and waits for the utility to accept it.

```
//node requests access to network
//must be accepted by utility
function requestEntry(string smId, string mac) public {
    //shouldn't be the utility AND shouldn't be a requestor AND
        shouldn't already be a Node
    require(msg.sender != utility
    && !hasRequested[msg.sender]
    && !isDeployed[msg.sender]);

    Node memory r = Node(smId, mac, msg.sender, false);
    requestedNodes.push(r);
    hasRequested[msg.sender] = true;
}
```

Listing 6.3: Request entry

The utility can accept the entry request using the "acceptNode" function - line 3 - shown in listing 6.4 or reject it (by not accepting it). If the caller of the function is not the utility, the function will throw an exception which we will further discuss in subsection 6.3.1. The node is retrieved from the array of requesters - line 4. If the consumer is not already added to the network, the function continues execution - line 11 - otherwise, the function will throw an exception which we will further discuss in subsection 6.3.1. Once the utility accepts the consumer,

the consumer is added to the mapping of deployed consumers - line 14 - and the number of nodes is increased by one - line 15. The node is now marked as complete which means that it can be considered part of the network - line 20.

The utility also has the ability to accept an entry and then remove it later for any reason using the "removeNode" function - line 2 - in listing 6.5. If the caller of the function is not the utility, the function will throw an exception which we will further discuss in subsection 6.3.1. Removal reasons can be caused by an entity requesting to stop its power service due to relocation, or disciplinary actions taken by the utility against malicious consumers. The node is retrieved from the array of requesters - line 4. If the consumer is part of the network (can be found in mapping of deployed consumers), the function continues execution - line 7, otherwise, the function will throw an exception which we will further discuss in subsection 6.3.1. Once the utility removes the consumer, the number of nodes is decreased by one - line 9 - and the consumer is removed from all the mappings (requesters, deployed consumers, and consumers that have data contract - lines 10, 11, 12 respectively) thus clearing it from the network.

```
1 //can only add node if the utility accepts
2 function acceptNode(uint index) public restrictedUtility {
3     //the node requested
4     Node storage r = requestedNodes[index];
5
6     //check if this node is valid
7     //based on smId db at utility
8     //done manually by calling verify function -> comparing values
          to values they have -> making the call
9
10    //shouldn't already be a Node
11    require(r.complete != true);
12
13    //add node to list that needs to create a contract
14    isDeployed[r.add] = true;
15    countNodes ++;
16
17    //mark as complete in request
18    //for ui:
19    //when complete true the row is disabled
20    r.complete = true;
21 }
```

Listing 6.4: Accept node

```
//utility can remove a node from network
function removeNode(uint index) public restrictedUtility {
    //the node requested
    Node storage r = requestedNodes[index];

    //should already be a Node
    require(isDeployed[r.add]);

    countNodes --;
    hasRequested[r.add] = false;
    isDeployed[r.add] = false;
    hasContract[r.add] = false;
}
```

Listing 6.5: Remove node

In addition, the contract includes a function that allows the utility to view how many requesters there are in the array of requesters using "getRequestsCount" function - line 3 - provided in listing 6.6. It returns the length of the array of requesters which is the number of consumers' nodes that have been added previously upon requesting entry - line 4.

```
//number of requests
//for ui
function getRequestsCount() public view returns (uint) {
    return requestedNodes.length;
}
```

Listing 6.6: Get requests count

Finally, the consumer may call the "canDeploy" function - line 4 - in listing 6.7 that allows the consumer to check if it is allowed to deploy its data contract (to be discussed in subsection 6.2.2). It returns if the consumer is in the deployed consumers mapping and is not in the deployed data contract mapping - line 5. The deployed consumer mapping shows if the utility has accepted it into the smart grid communication network. The deployed data contract mapping shows if the consumer has deployed its data contract (to be discussed in subsection 6.2.3). If the consumer is not in this mapping, it means it is allowed to deploy its data contract.

If it can deploy, it calls the "markDone" function in listing 6.8 after having called the "createDataContract" function in listing 6.11 found in subsection 6.2.2. Calling the "markDone" function - line 2 - makes sure the consumer can no longer request to create another data contract (for security purposes) by adding it to the deployed data contract mapping - line 5. The caller of this function must be the one updating their own record - line 4.

```
1 //is node in network but does not have deployed contract
2 //evaluates to true when the node is in the network (accepted by
     the utility)
3 //and when the node has not deployed a contract yet
4 function canDeploy(address add) public view returns (bool) {
5     return isDeployed[add] && !hasContract[add];
6 }
```

Listing 6.7: Can deploy

```
1 //mark as not able to deploy data contract anymore
2 function markDone(address add) public {
3     //marking should be done by node not any other node
4     require(msg.sender == add);
5     hasContract[add] = true;
6 }
```

Listing 6.8: Mark done

6.2.2 Second Smart Contract - Setting Up the Communication

This contract is deployed by the utility only once at inception. Figure 6.5 shows a high level view of the smart grid communication network in relation to the second smart contract. The specifics of the city, households, and utility can be found in subsection 6.2.1.

Node 11, the utility, contains the second smart contract known as the factory smart contract. Once deployed at the inception of the smart grid communication network, the smart contract becomes accessible by all other nodes on the blockchain since it is now part of the blockchain's ledger. This smart contract is deployed to allow the consumers to create their data smart contracts needed for the data exchange between the consumer with the utility.

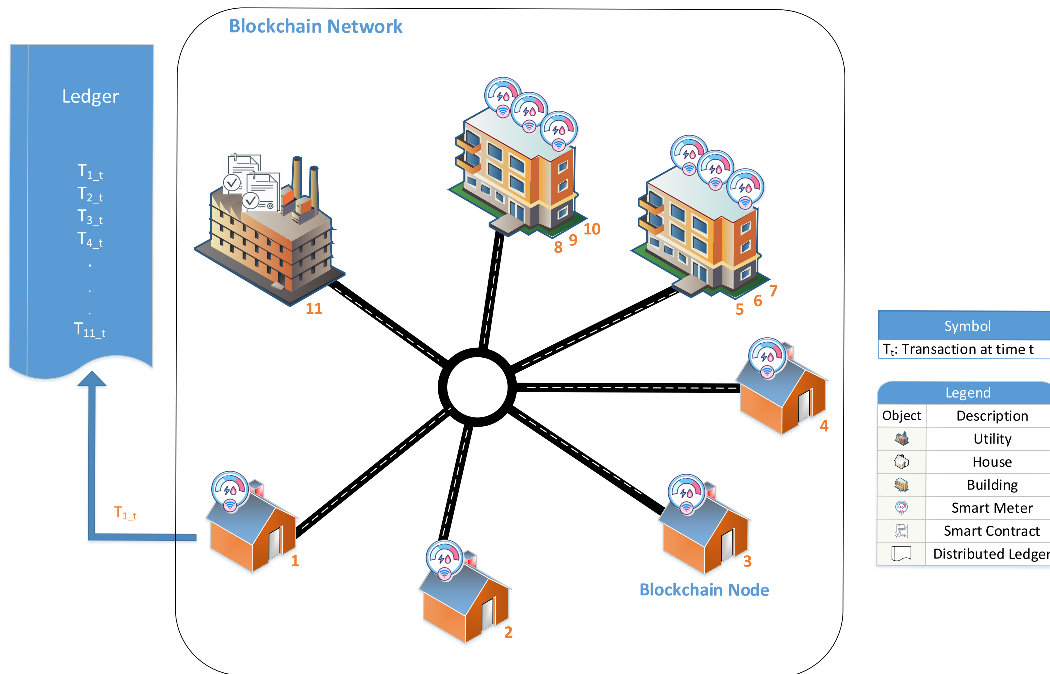

Figure 6.5: Architecture 1: Smart Contract #2 Distribution

As we've explained in subsection 2.4.2, variables are set up with which to interact and functions exist to allow actions to occur on the blockchain. The detailed code of the second smart contract variables and functions is available in Appendix A listing A.2. Here, we will explain the variables and functions briefly and their purpose in the smart contract.

Listing 6.9 displays the variable names and descriptions in this contract. These

variables are used in functions we discuss below.

- **Line 1 - deployedDataContracts**: Array of all the deployed data contracts
- **Line 2 - utility**: Utility's Ethereum address
- **Line 3 - numberContracts**: Number of deployed data contracts
- **Line 5 - Deployed**: Event declaration for contract deployment that returns the deployed contract's address (the concept of smart contract events is discussed extensively in subsection 2.4.8)

```
address[] public deployedDataContracts;
address public utility;
uint public numberContracts;

event Deployed(address indexed _add);
```

Listing 6.9: Variables

The functions included in this smart contract are listed below and will be explained in listings 6.10 and 6.11 respectively.

- Constructor
- Create data contract

The "constructor" function - line 3 - initializes the variables above as seen in listing 6.10. The Ethereum utility address is set to the caller of the constructor function "msg.sender" which is the utility - line 4.

```
//only called once by utility
//since deployed by utility
function Factory() public {
  utility = msg.sender;
}
```

Listing 6.10: Constructor

Via this smart contract, the consumer will be able to create his/her data smart contract which they will use to communicate smart meter electrical data to the utility and where the utility may send load balancing data to the consumer. Using the "createDataContract" function - line 2 - in listing 6.11, the consumer may complete the task of creating the data contract. This deploys the data smart contract by calling the data smart contract constructor - line 3 - discussed in subsection 6.2.3. This call returns the address of the deployed data contract that can be accessed by both the consumer and the utility - line 3. This address

will be added to the array of deployed data contracts - line 4 - and the counter of deployed contracts is incremented - line 5. This address is also sent as an event so that the consumer can log it and access the smart contract - line 7. The concept of smart contract events is discussed extensively in subsection 2.4.8.

```
//create the factory contract
function createDataContract() public returns (address){
  address c = new Data(utility, msg.sender);
  deployedDataContracts.push(c);
  numberContracts++;

  Deployed(c);
  return c;
}
```

Listing 6.11: Create data contract

In addition, the contract includes a function that allows the nodes in the network to get the array of deployed data smart contracts using the "getDeployedDataContract" function - line 2 - shown in listing 6.12. It returns the length of the array of all the deployed data contracts which is the number of deployed data contracts in the smart grid communication network - line 3.

```
//get addresses for all contracts
function getDeployedDataContracts() public view returns(address[]) {
  return deployedDataContracts;
}
```

Listing 6.12: Get deployed data contracts

6.2.3 Third Smart Contract - Communicating

This contract is deployed by the consumer only once through the factory contract described in subsection 6.2.2. Figure 6.6 shows a high level view of the smart grid communication network in relation to the third smart contract. The specifics of the city, households, and utility can be found in subsection 6.2.1.

Nodes 1 through 10, the consumers, contain an instance of the third smart contract known as the data smart contract. Once deployed by each consumer, the smart contract becomes accessible by all other nodes on the blockchain because it is now part of the blockchain's ledger. This smart contract is deployed to allow the consumers to communicate their energy data securely and the utility to send load balancing data back to the consumer (demand-response). This preserves the two-way data flow for which the smart grid is known.

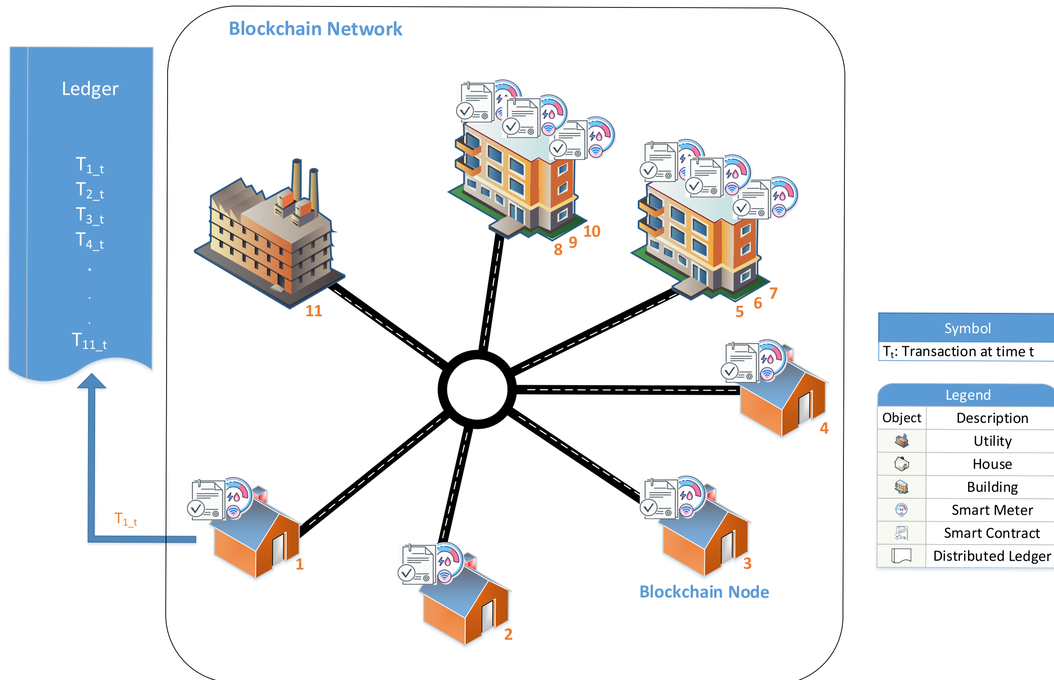

Figure 6.6: Architecture 1: Smart Contract #3 Distribution

As we've explained in subsection 2.4.2, variables are set up with which to interact and functions exist to allow actions to occur on the blockchain. The detailed code of the third smart contract variables and functions is available in Appendix A listing A.3. Here, we will explain the variables and functions briefly and their purpose in the smart contract.

Listing 6.13 displays the variable names and descriptions in this contract. These variables are used in functions we discuss below.

- **Line 1 - owner**: Owner's Ethereum address. The owner of this contract is the consumer that has deployed it
- **Line 2 - utility**: Utility's Ethereum address
- **Line 3 - entries**: Number of entries that counts the number of electrical data entries
- **Line 4 - loadBalancing**: Load balancing data that is received from the utility
- **Line 5 - electricity**: Map of electrical data that is sent from the consumer to the utility

```
1 address public owner;
2 address public utility;
3 uint public entries;
4 uint public loadBalancing;
5 mapping(string => uint) electricity;
```

Listing 6.13: Variables

The functions included in this smart contract are listed below and will be explained in listings 6.14, 6.15, 6.16, and 6.17 respectively.

- Constructor
- Send data
- Check data
- Fix data

We assume that the consumers using this smart contract have been accepted onto the network and are accessing this contract to send electrical data to the utility and to get load balancing data from the utility. The "constructor" function - line 1 - initializes the variables above as seen in listing 6.14. The Ethereum consumer address is set to "creator" - line 2 - which, as we have mentioned in subsection 6.2.2, is called by the consumer. The Ethereum utility address is set to the address of the utility - line 3 - which is previously known through the simulation smart contract discussed in subsection 6.2.1. The load balancing data is set to 0 since we assume the consumers do not need to receive any load balancing data upon entry - line 4.

```
1 function Data(address utilityAdd, address creator) public {
2     owner = creator;
3     utility = utilityAdd;
4     loadBalancing = 0;
5 }
```

Listing 6.14: Constructor

Via this smart contract, the consumer can send electrical data to the utility. Sending data using the "sendData" function - line 5 - in listing 6.15 can only be done if the caller of the function is the owner of the contract, otherwise, the function will throw an exception which we will further discuss in subsection 6.3.3. It will add electrical data to the mapping of the electrical data by taking the date and time (following a certain format) - line 6. The data entries will be incremented by one - line 7.

```
1 //node requests access to network
2 //must be accepted by utility
3 //send the Data
4 //timestamp should follow certain convention yyyymdhm
5 function sendData(string timestamp, uint value) public
     restrictedOwner {
6   electricity[timestamp] = value;
7   entries++;
8 }
```

Listing 6.15: Send data

The utility can check the electrical data of the consumer using the "lookupData" function - line 2 - in listing 6.16. The utility will query the mapping of the electrical data using the date and time and return the resulting value - line 3. This will provide the electrical data for the given date and time.

The utility may also send load balancing data using the "fixData" function - line 4 - in listing 6.17 setting the load balancing variable to the load balancing value decided by the utility - line 5. The consumer can then query the smart contract to access this variable at specific intervals.

```
1 //check this data
2 function lookupData(string timestamp) public restrictedUtility view
     returns (uint) {
3   return electricity[timestamp];
4 }
```

Listing 6.16: Look-up data

```
//utility
//send load balancing data
//will be enumerator 0->decrease 1->increase ....
function fixData(uint value) public restrictedUtility {
    loadBalancing = value;
}
```

Listing 6.17: Fix data

6.2.4 Summary

A smart grid communication network will start out empty with only the utility present. The utility initially deploys two smart contracts. The first smart contract (simulation contract) will allow consumers to enter the smart grid communication network, and the second smart contract (factory contract) will allow the consumers to deploy their own data smart contract. Each consumer must deploy its own data smart contract.

The consumers that wish to join the network and become part of the smart grid can do so by requesting to join. The utility can then accept or reject this consumer. Once accepted, the consumer may now proceed to deploy his/her data smart contract. If rejected, the consumer cannot proceed with deploying his/her data smart contract.

The consumers can now deploy their third smart contract (data contract) through the factory contract. The consumer now has access to his/her newly deployed data contract.

The consumer can now use the third smart contract (data contract) to send data periodically to the utility. The utility can also send balancing data to the consumer through the respective data contract.

The utility may use the factory contract to view all the smart contracts deployed by the consumers, query the consumers' data contracts for electric data, and send load balancing data to the consumers' data contracts.

6.3 User Interface

To complete the preventative proposed solution, we set up a user interface for user data interaction. The user may interact with Ethereum blockchain using the Rinkeby Test Network discussed above. Sections 6.1 and 6.2 discuss the general architecture execution and details and this section will describe the user interface put in place to interact with the Ethereum network smart contracts. Some front-end components provide the consumer access to the different smart contracts deployed by the utility or consumers. Other front-end components provide the utility access to the different smart contracts deployed by the utility or consumers. The user interface does not show the deployment of the simulation and factory smart contracts because it should be completed by the utility before any interaction with the smart contracts can occur.

6.3.1 First Smart Contract - Joining the Network

Figure 6.7 shows the user interface where a consumer can request entry into the smart grid communication network. There are certain fields that need to be filled in which the utility needs (to confirm the user is authentic).

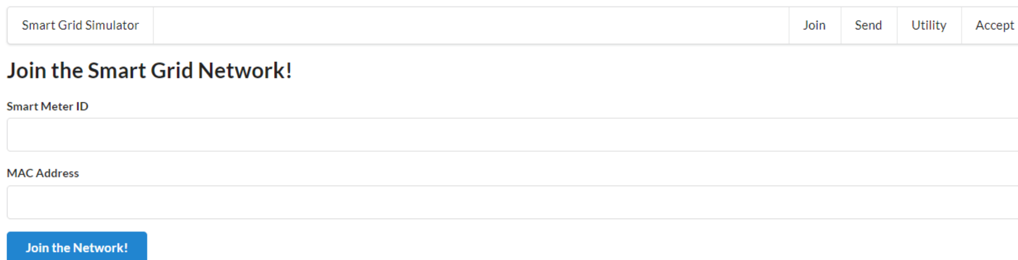

Figure 6.7: Architecture 1: Join

Listing 6.18 shows the code used to interact with the simulation smart contract (first smart contract) using the interface in figure 6.7. The asynchronous[2] transaction - line 14 - calls the smart contract function “requestEntry” (discussed in subsection 6.2.1 listing 6.3) that takes two parameters evident in figure 6.7. The function uses the “send” keyword which leads to a transaction that changes the Ethereum state. As discussed in subsection 2.4.5 the “from” keyword shows who is creating the transaction.

[2]The async keyword is used in the function definition to declare it is asynchronous. The await keyword used to show the action of waiting for a promise. The function is paused in a non-blocking way until the promise settles. The value is returned if the promise fulfills.

```
onSubmit = async (event) =>{
    event.preventDefault();

    try{
        //enable use of web3 instance
        await ethereum.enable();

        //get the accounts provided by metamask
        const accounts = await web3.eth.getAccounts();
        console.log(accounts[0]);

        //call the smart contract function to request entry
        //2 params: smart meter id, and mac address
        await simulation.methods.requestEntry(this.state.smId,
            this.state.mac).send({from: accounts[0]});

    } catch(err) {
        this.setState({errorMessage: err.message});
    }
};
```

Listing 6.18: Join Code

The interaction between the consumer and the first smart contract's "requestEntry" function could either be successful or unsuccessful due to a certain property not being satisfied in the smart contract. This could happen for many reasons:

- The consumer requesting entry is the utility
- The consumer has previously requested entry into the network
- The consumer has previously deployed the data contract
- The consumer has not provided enough gas for the transaction to succeed

Figures 6.8 and 6.9 show the user interface where the utility can accept consumers based on the properties provided. The utility may also reject the consumers by not accepting them. Beneath the list of consumers to accept/remove, there are two values: the total number of requesters and the number of nodes accepted into the network both queried from the smart contract.

Pending Requests

Smart Meter ID	MAC Address	Node Network Address	Approve	Remove

Found 0 request(s)
Found 0 current node(s)

Figure 6.8: Architecture 1: Accept Initial

Smart Grid Simulator | Join | Send | Utility | Accept

Pending Requests

Smart Meter ID	MAC Address	Node Network Address	Approve	Remove
1234	1	0x88B1A85759a33971f6647d77C7c691BB91c125bc	Approve	Remove
5678	1	0x3c3C6893c843Dc25B932F00cB4467e4dF2C1c77F	Approve	Remove
91011	2	0x8D23396c2Fc2ABD2F28a983dFECf289D24A69A3F	Approve	Remove
121314	2	0xC696E696BE257c80F26804c6b231a23cbB67E4Ab	Approve	Remove

Found 4 request(s)
Found 0 current node(s)

Figure 6.9: Architecture 1: Accept

Listing 6.19 shows the code used to interact with the simulation smart contract (first smart contract) using the interface in figure 6.9.

The asynchronous transaction - line 4 - calls the smart contract function "countNodes" (discussed in subsection 6.2.1 listing 6.1) that doesn't take any parameters. This variable is placed below the table of pending requests indicating the number of nodes accepted into the network. The function uses the "call" keyword which does not lead to a change the Ethereum state.

The asynchronous transaction - line 8 - calls the smart contract function "getRequestsCount" (discussed in subsection 6.2.1 listing 6.1) that doesn't take any parameters. This variable is placed below the table of pending requests, indicating the number of nodes requesting entry into the network. The function uses the "call" keyword, which does not lead to a change the Ethereum state.

The asynchronous transaction - line 14 - calls the smart contract getter function for the array of requesters "requestedNodes" (discussed in subsection 6.2.1 listing 6.1) that takes one parameter. The array elements are the requesters displayed in the table of pending requests where each requester is placed in a row of the table. The function uses the "call" keyword, which does not lead to a change the Ethereum state.

```
static async getInitialProps(props) {
    //call the smart contract function to get number of nodes in
       network
    //0 params
    const currentNodes = await
       simulation.methods.countNodes().call();

    //call the smart contract function to get the number of
       requesters
    /0 params
    const requestCount = await
       simulation.methods.getRequestsCount().call();

    const requests = await Promise.all(
        Array(parseInt(requestCount)).fill().map((element,index) =>
           {
            //call the smart contract function to mark node as
               having deployed data contract
            //0 params
            return simulation.methods.requestedNodes(index).call()
        })
    );

    return {requests, requestCount,currentNodes};
}
```

Listing 6.19: Get Counter Code

Listing 6.20 shows the code used to interact with the simulation smart contract (first smart contract) using the interface in figure 6.9. The asynchronous transaction - line 6 - calls the smart contract function "acceptNode" (discussed in subsection 6.2.1 listing 6.4) that takes one parameter. The function uses the "send" keyword which leads to a transaction that changes the Ethereum state.

```
onApprove = async () => {
    const accounts = await web3.eth.getAccounts();

    //call the smart contract function to accept node
    //1 param: id in list
    await simulation.methods.acceptNode(this.props.id).send({from:
       accounts[0]});
};
```

Listing 6.20: Accept Code

The interaction between the utility and the first smart contract's "acceptNode" function could either be successful or unsuccessful due to a certain property not being satisfied in the smart contract. This could happen for many reasons as indicated:

- The caller of the function is not the utility

- The utility has previously accepted the consumer into the network
- The utility has not provided enough gas for the transaction to succeed

Figure 6.10 shows the user interface where the utility may remove consumers. The figure shows that the consumers have already been accepted onto the network (no accept button). Only these consumers may be removed.

Smart Grid Simulator | Join | Send | Utility | Accept

Pending Requests

Smart Meter ID	MAC Address	Node Network Address	Approve	Remove
1234	1	0x88B1A85759a33971f6647d77C7c691BB91c125bc		Remove
5678	1	0x3c3C6893c843Dc25B932F00cB4467e4dF2C1c77F		Remove
91011	2	0x8D23396c2Fc2ABD2F28a983dFECf289D24A69A3F		Remove
121314	2	0xC696E696BE257c80F26804c6b231a23cbB67E4Ab		Remove

Found 4 request(s)
Found 4 current node(s)

Figure 6.10: Architecture 1: Remove

Listing 6.21 shows the code used to interact with the simulation smart contract (first smart contract) using the interface in figure 6.10. The asynchronous transaction - line 6 - calls the smart contract function "removeNode" (discussed in subsection 6.2.1 listing 6.5) that takes one parameter. The function uses the "send" keyword which leads to a transaction that changes the Ethereum state.

```
1 onRemove = async () => {
2     const accounts = await web3.eth.getAccounts();
3 
4     //call the smart contract function to remove node
5     //1 param: id in list
6     await simulation.methods.removeNode(this.props.id).send({from:
          accounts[0]});
7 };
```

Listing 6.21: Remove Code

The interaction between the utility and the first smart contract's "removeNode" function could either be successful or unsuccessful due to a certain property not being satisfied in the smart contract. This could happen for many reasons as indicated:

- The caller of the function is not the utility
- The utility has not previously accepted the consumer into the network
- The utility has not provided enough gas for the transaction to succeed

Before being able to send data back and forth between the consumer and the utility, the consumer must have deployed its data contract. Figure 6.11 shows the screen the consumer will see till the data contract has been deployed.

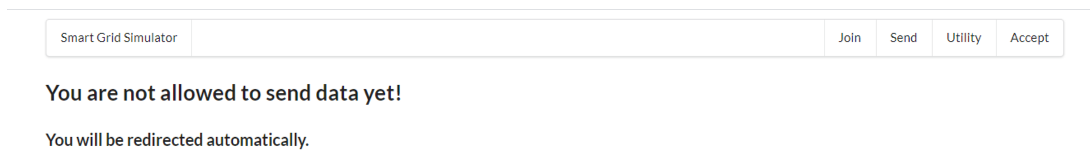

Figure 6.11: Architecture 1: Check

Listing 6.22 shows the code used to interact with the simulation smart contract (first smart contract) using the interface in figure 6.11. The asynchronous transaction - line 4 - calls the smart contract function "canDeploy" (discussed in subsection 6.2.1 listing 6.7) that takes one parameter. The function uses the "call" keyword which does not lead to a change in the Ethereum state.

```
1 checkStatus = async () => {
2     await ethereum.enable();
3     const accounts = await web3.eth.getAccounts();
4 
5     //call the smart contract function to check if consumer can
        deploy
6     //1 param: address
7     const status = await
        simulation.methods.canDeploy(accounts[0]).call();
8 }
```

Listing 6.22: Check Code

We do not show the Metamask interaction between the utility and the first smart contract's "canDeploy" function since this function is not a transaction that changes the Ethereum state. Therefore, it does not need to be verified. It is considered to be querying the distributed ledger.

6.3.2 Second Smart Contract - Setting Up the Communication

Listing 6.23 shows the code used to interact with the simulation smart contract (first smart contract) and the factory smart contract (second smart contract). As noticed, this listing includes listing 6.22 since the three functions, "canDeploy", "createDataContract", and "markDone", directly depend on each other as explained in subsection 6.1 figure 6.3.

The asynchronous transaction - line 14 - calls the smart contract function "createDataContract" (discussed in subsection 6.2.2 listing 6.11) that doesn't take any parameters. The function uses the "send" keyword which leads to a transaction that changes the Ethereum state.

Line 18 shows the extraction of the deployed data contract's Ethereum address (explained in 2.4.2) from the event sent (shown in listing 6.11).

The asynchronous transaction - line 23 - calls the smart contract function "markDone" (discussed in subsection 6.2.1 listing 6.8) that takes one parameter. The function uses the "send" keyword which leads to a transaction that changes the Ethereum state.

The consumer is rerouted to the page where he/she can send and receive data through the deployed data smart contract (whose address is specified in the URL) - line 26.

```
checkStatus = async () => {
    await ethereum.enable();
    const accounts = await web3.eth.getAccounts();

    //call the smart contract function to check if consumer can
        deploy
    //1 param: address
    const status = await
        simulation.methods.canDeploy(accounts[0]).call();

    let dataAddress="";
    if(status)
    {
        //call the smart contract function to deploy data contract
        //0 param
        dataAddress = await
            factory.methods.createDataContract().send({from:
            accounts[0]});

        //get address from event
        //use contract address to access smart contract
        dataAddress = dataAddress.events.Deployed.returnValues._add;
        console.log(dataAddress);

        //call the smart contract function to mark node as having
            deployed data contract
        //1 param: address
        await simulation.methods.markDone(accounts[0]).send({from:
            accounts[0]});

        //reroute to send
        Router.pushRoute(`/data/send/${dataAddress}`);
    }
}
```

Listing 6.23: Check Code

The interaction between the consumer and the second smart contract's "createDataContract" function could be either successful or unsuccessful due to a certain property not being satisfied in the smart contract. The interaction between the consumer and the first smart contract's "markDone" function could be either successful or unsuccessful due to a certain property not being satisfied in the smart contract. This could happen for many reasons as indicated:

- The consumer attempting to call this function is not trying to call it for themselves
- The consumer has not provided enough gas for the transaction to succeed

6.3.3 Third Smart Contract - Communicating

Now that the smart contract is deployed by the consumer, who is called the owner of the contract, he/she can send data to the utility. Figure 6.12 shows the user interface where a consumer can input the electrical data needed by the utility and send it.

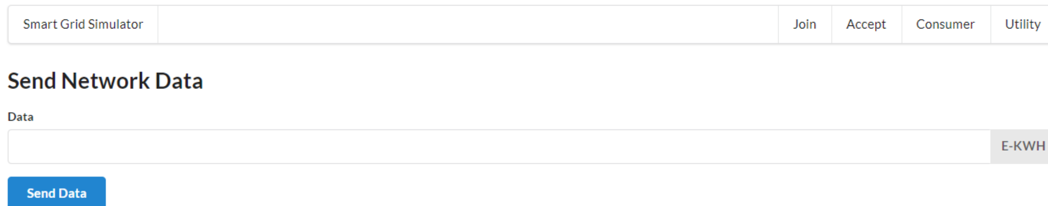

Figure 6.12: Architecture 1: Send Electrical Data

Listing 6.24 shows the code used to interact with the simulation smart contract (first smart contract) using the interface in figure 6.12.

Line 7 shows the smart contract's Ethereum address that is used to access the consumer's data smart contract.

Line 15 shows the combination of the current year, month, date, hour, and minute needed to form a timestamp the utility will use to find the electrical data.

The asynchronous transaction - line 19 - calls the smart contract function "sendData" (discussed in subsection 6.2.3 listing 6.15) that takes two parameters, one of which is the timestamp explained in line 15 and the other is the electrical data and is evident in figure 6.12. The function uses the "send" keyword which leads to a transaction that changes the Ethereum state.

```
onSubmit = async (event) => {
    const {address} = this.props;
    event.preventDefault();
    await ethereum.enable();

    //getting the instance of the contract
    const contract = data(address);

    try {
        const accounts = await web3.eth.getAccounts();
        const today = new Date();

        //used for the timestamp
        //gets current date in specific format
        const time = "" + today.getFullYear() +
            (today.getMonth()+1) + today.getDate() +
            today.getHours() + today.getMinutes();

        //call the smart contract function to send electrical data
        //2 params: timestamp, electrical data
        await contract.methods.sendData(time,
            this.state.data).send({from: accounts[0]});

    } catch(err) {
        this.setState({errorMessage: err.message});
    }
}
```

Listing 6.24: Send Code

The interaction between the consumer and the third smart contract's "sendData" function could be either successful or unsuccessful due to a certain property not being satisfied in the smart contract. This could happen for many reasons as indicated:

- The caller of the function is not the owner of the contract
- The consumer has not provided enough gas for the transaction to succeed

Finally, figure 6.13 shows the user interface where the utility can get the electrical data needed at a certain time by interacting with the third smart contract's "lookupData" function.

Smart Grid Simulator Join Send Utility Accept

Check Network Data

Time and Date Needed

YYYYMDHM

Check

Value Returned:

Figure 6.13: Architecture 1: Get Electrical Data at Time

Listing 6.25 shows the code used to interact with the data smart contract (third smart contract) using the interface in figure 6.13.

Line 7 shows the smart contract's Ethereum address that is used to access the consumer's data smart contract.

The asynchronous transaction - line 13 - calls the smart contract function "lookupData" (discussed in subsection 6.2.3 listing 6.16) that takes one parameter evident in figure 6.13.

Line 15 shows the load balancing data retrieved in the asynchronous function call. The function uses the "call" keyword which does not lead to a change in the Ethereum state.

```
1 onSubmit = async (event) => {
2     const {address} = this.props;
3     event.preventDefault();
4     await ethereum.enable();
5 
6     //getting the instance of the contract
7     const contract = data(address);
8 
9     const accounts = await web3.eth.getAccounts();
10 
11     //call the smart contract function to send electrical data
12     //1 param: timestamp, electrical data
13     const valueReturned = await
           contract.methods.lookupData(this.state.timeDate).call({from:
           accounts[0]});
14     console.log(valueReturned);
15     this.setState({value: valueReturned});
16 }
```

Listing 6.25: Look-up Code

We do not show the Metamask interaction between the utility and the third smart contract's "lookupData" function since this function is not a transaction that changes the Ethereum state. Therefore, it does not need to be verified. It is considered to be querying the distributed ledger.

6.4 Results

The results show the various smart contract components in the different architectures. These components include variables and functions used in the deployed smart contracts. The functions consume certain amounts of gas; thus, the cost of using them varies. These concepts have been discussed in subsection 2.4.7. The test network used in this thesis, as mentioned in section 5.2, is the Rinkeby Test Network where the price of gas is constant and is set at 1 GWei. However, in the real network, the cost of gas varies at an average price of 20 GWei. We used 1 Ether equivalent to 186 USD. The cost is not influenced by the size of the network but by the complexity of the transaction. Thus, simulating a network with few consumers versus many consumers will not affect the results in any way.

The results are those of a smart grid communication network composed of various smart meters and one utility connected via the Ethereum infrastructure. This architecture describes the entry of a consumer into the smart grid communication network created. Entering the smart grid communication network will allow the consumer to send electrical data to the utility and the utility to send load balancing data to the consumer. The details concerning these smart contracts are found in sections 6.1 and 6.2.

For each of the contracts, there will be a table discussing the transaction title, who the transaction is sent from, who the transaction is sent to, the amount of gas used, and the fee (in ether). Moreover, there will be tables discussing: the transaction title, the fee (in ether), the cost (in $), the frequency the transaction is sent at, and the total cost (in $) which is relative to the frequency the transaction is sent at and the cost per transaction.

6.4.1 First Smart Contract - Joining the Network

All the transactions in Table 6.1 and Table 6.2 found in the first column (deploy simulation contract action, join action, accept action, and mark data contract deployed action), are made through smart contract functions. Both the consumer and the utility will be interacting through the use of the simulation smart contract (refer to section 6.2.1 for more details about the smart contract).

Table 6.1 describes the steps a consumer takes to enter into the network. Since the utility deploys the simulation contract, it carries the burden of using up a big amount of gas. The join actions, carried out by the consumers, and the accept actions, carried out by the utility, are transactions sent to the smart contract. They use up varying amounts of gas depending on the complexity of the functions in the smart contract. Clearly the join action is more complex than the accept action, as shown by the difference in the gas consumption. The mark data contract deployed action is a transaction sent to the smart contract and is carried out by the consumer. It will cost considerably less than the deploy action, the join action, and the accept action. The following table describes the logistics of the transactions.

Table 6.1: Architecture 1: Joining the Smart Grid Communication Network - Smart Contract #1

Transaction	From	To	Gas Used	Fee (ether)
Deploy Simulation Contract	Utility	-	616950	0.00061695
Join	Home	Contract	125301	0.000125301
Accept	Utility	Contract	60764	0.000060764
Mark Data Contract Deployed	Home	Contract	42910	0.00004291

Table 6.2 complements Table 6.1 and includes the dollar amount of the cost of entering into the network for both the utility and the consumers. The deployment action will occur once at inception and is quite inexpensive. The join action will be completed once by every consumer that wishes to enter the network. The accept action will be completed once for every consumer that is allowed to enter the network. Finally, the mark data contract deployed action will be completed once by every consumer that has successfully deployed their data contract.

Table 6.2: Architecture 1: Cost of Transactions in Joining - Smart Contract #1

Transaction	Fee (ether)	Cost ($)	Frequency	Total Cost($)
Deploy Simulation Contract	0.00061695	0.099	1	0.099
Join	0.000125301	0.023	1 * # HH	0.023 * # HH
Accept	0.000060764	0.011	1 * # HH	0.011 * # HH
Mark Data Contract Deployed	0.00004291	0.0079	1 * # HH	0.0079 * # HH

Assuming the population in Qatar is currently 2,869,458[3] and the number of buildings built are 216,740[4]. It is safe to estimate that there are around 2,167,400 households (HHs)/establishments that need electricity. Equation 6.1 shows the payment needed to be made by the utility once in the smart grid communication network, and equation 6.2 shows the payment needed to be made by the consumer once in the smart grid communication network.

$$\begin{aligned}\underline{\textbf{Utility Payment \#1}} &= \text{deploy simulation contract action} + \text{accept action} * \#\ \text{HH} \\ &= 0.099\$ + 0.011\$ * 2,167,400 \\ &= 26,008.884\$\end{aligned} \tag{6.1}$$

$$\begin{aligned}\underline{\textbf{Consumer Payment \#1}} &= \text{join action} + \text{mark data contract deployed action} \\ &= 0.023\$ + 0.0079\$ \\ &= 0.0309\$\end{aligned} \tag{6.2}$$

[3] https://www.worldometers.info/world-population/qatar-population/

[4] https://www.gulf-times.com/story/635497/45-6-percent-jump-in-number-of-buildings-in-10-years

6.4.2 Second Smart Contract - Setting Up the Communication & Third Smart Contract - Communicating

All the transactions in Table 6.3, Table 6.4, or Table 6.5 found in the first column (deploy factory contract action, deploy data contract action, and send data action), are made through smart contract functions. Both the consumer and the utility will be interacting through the use of the factory smart contract or the data smart contract (refer to sections 6.2.2 and 6.2.3 respectively for more details about the smart contracts).

Table 6.3 describes the steps a consumer takes to communicate in the network. Since the utility deploys the factory contract, it carries the burden of using up a big amount of gas. The consumer deploys the data contract and carries the burden of using up a big amount of gas. The amounts of gas consumption vary from factory contract to data contract depending on the number of variables, functions, and other factors. The send data actions, done by the consumers are transactions sent to the smart contract. Clearly the deployment actions are more complex than the send data action. The following table describes the logistics of the transactions.

Table 6.3: Architecture 1: Communication in the Smart Grid Communication Network - Smart Contract #2 & #3

Transaction	From	To	Gas Used	Fee (ether)
Deploy Factory Contract	Utility	-	533826	0.000533826
Deploy Data Contract	Home	Contract	345470	0.00033047
Send Data	Home	Contract	63916	0.000063916

Table 6.4 and Table 6.5 complement Table 6.3 and include the dollar amount of the cost of communicating in the network for both the utility and the consumers. The factory contract will be deployed by the utility once at inception and is quite inexpensive. The data contract will be deployed by each consumer once at inception and is quite inexpensive (Table 6.4). The send data action (Table 6.5) will be completed by every consumer that wishes to communicate electrical data to the utility. This action could occur at 15 minute intervals, 30 minute intervals, or 60 minute intervals a day. The total cost per day is calculated for these different intervals and a significant difference can be seen.

Table 6.4: Architecture 1: Cost of Transactions in Initializing the Communication Environment - Smart Contract #2 & #3

Transaction	Fee (ether)	Cost ($)	Frequency	Total Cost ($)
Deploy Factory Contract	0.000533826	0.099	1	0.099
Deploy Data Contract	0.00033047	0.061	1 * # HH	0.061 * # HH

Table 6.5: Architecture 1: Cost of Transactions in Sending Electrical Data - Smart Contract #3

Transaction	Fee (ether)	Cost ($)	Frequency/Day (min)	Total Cost/Day ($)
Send Data	0.000063916	0.011	15 * # HH	1.056 * # HH
Send Data	0.000063916	0.011	30 * # HH	0.528 * # HH
Send Data	0.000063916	0.011	60 * # HH	0.264 * # HH

We assume the same statistics mentioned above about the population, the number of buildings built, and the number of households (HHs)/establishments that need electricity are applicable. Equation 6.3 shows the payment needed to be made by the utility once in the smart grid communication network. Equation 6.4 shows the payment needed to be made by the consumer once in the smart grid communication network. Equation 6.5 shows the maximum payment needed to be made by the consumer once every year.

$$\begin{aligned} \underline{\textbf{Utility Payment \#2}} &= \text{deploy factory contract action} \\ &= 0.099\$ \end{aligned} \tag{6.3}$$

$$\begin{aligned} \underline{\textbf{Consumer Payment \#2}} &= \text{deploy data contract action} \\ &= 0.061\$ \end{aligned} \tag{6.4}$$

$$\begin{aligned} \underline{\textbf{Consumer Payment \#3}} &= \text{send data action} * 365 \\ &= 1.056\$ * 365 \\ &= 385.44\$ \end{aligned} \tag{6.5}$$

6.4.3 Summary

As noticed, the prices are high for an operational smart grid especially when you take the price of gas into consideration in the Main Ethereum Network. The high cost that comes with this approach is tackled in architecture 2. Less smart contracts can be deployed leading to less fees by the consumer and less storage consumption on the distributed ledger. Other steps can be taken to decrease the cost including simplifying the functions in the smart contract while maintaining the efficiency of the security measures. The architecture's limitations will be discussed in section 6.6.

The three smart contracts used in this architecture pose different sums to be paid by both the utility and the consumer at different points in time of the smart grid communication network. Equation 6.1, equation 6.2, equation 6.3, equation 6.4, and equation 6.5 show the payments that need to be made by the utility and the consumer at different points in the smart grid communication process. We wrap up these equations by summing up all the expenses that are paid by the utility and the consumer. Equation 6.6 sums up the expenses paid by the utility to set up the smart grid communication network for themselves and the consumers. Equation 6.7 sums up the expenses paid by each consumer to set up their interaction with the smart grid communication network. Finally, equation 6.8 sums up the expenses that have to be paid by each consumer yearly to interact with the utility in the smart grid communication network.

$$\begin{aligned}\underline{\textbf{Total Initial Utility Payment}} &= \text{Utility Payment \#1} \\ &+ \text{Utility Payment \#2} \\ &= 26,008.884\$ + 0.099\$ \\ &= 26,008.983\$\end{aligned} \tag{6.6}$$

$$\begin{aligned}\underline{\textbf{Total Initial Consumer Payment}} &= \text{Consumer Payment \#1} \\ &+ \text{Consumer Payment \#2} \\ &= 0.0309\$ + 0.061\$ \\ &= 0.0919\$\end{aligned} \tag{6.7}$$

$$\begin{aligned}\underline{\textbf{Total Consumer Payment Per Year}} &= \text{Consumer Payment \#3} \\ &= 385.44\$\end{aligned} \tag{6.8}$$

6.5 Security Properties

The security properties achieved in this architecture can be credited to the use of both the blockchain network and the smart contracts. The Ethereum blockchain ensures various security properties discussed in subsection 2.4.10 whereas the smart contracts and their details are discussed in subsection 2.4.2. Both the Ethereum blockchain and its smart contracts are discussed in terms of the security properties achieved.

6.5.1 Smart Contract Properties

The architecture's three smart contracts discussed in section 6.2 subsections 6.2.1, 6.2.2, and 6.2.3 were structured to provide certain security properties alongside the innate blockchain security properties. The smart contracts contain certain restrictions and requirements that force functions to be constrictive and serve the purpose of securing the use of the smart grid communication network. It is important to note that the use of the term "network" is synonymous with the "smart grid communication network" and not the whole blockchain network.

First Smart Contract - Joining the Network

This contract is deployed only once by the utility. Once deployed, at the inception of the smart grid communication network, the simulation smart contract becomes accessible by all other nodes on the blockchain since it is now part of the blockchain's ledger. This smart contract is deployed to allow the consumers to join the network composed of the different consumers containing legal smart meters registered at the utility. The details of this smart contract can be found in subsection 6.2.1.

The "constructor" function is used by the utility to deploy the contract. The utility places its Ethereum address on the smart contract to restrict the use of some smart contract functions. This provides **authenticity** and limits who can call the functions to **secure** the network.

The "requestEntry" function contains three requirements. The first is that the caller of the function is not the utility in order to avoid the confusion of adding the utility as a consumer. This requirement can be removed based on the requirements of the utility, but it is put in place on the off-chance that the utility has been compromised and attempts to masquerade as a legitimate consumer and send false data. This provides **integrity** of data and **security** of the network. The second is that the caller of the function must not have previously requested entry into the network. This requirement stops the consumer from bombarding the network with useless requests. If the consumer is a malicious user, they could use any of the smart contract functions to overload the network. This function ensures that the consumer has not previously requested entry. This requirement provides **availability** of the blockchain services. The third is that the caller of the function must not already be part of the network. This requirement stops the consumer from bombarding the network with useless requests. If the consumer is a malicious user, they could use any of the smart contract functions to overload the network. This function ensures that the consumer has not already been added to the network. This also provides **availability** of the blockchain services.

The "acceptNode" function contains two requirements. The first is that the caller of the function is the utility. This stops any user in the blockchain network from accepting themselves or any other malicious users onto the network. This provides **legitimacy** of the consumers on the network and **integrity** of the electrical data to be sent by this consumer. The second is that the consumer the utility is trying to accept into the network must not already be part of the network. This requirement stops the utility from accepting a consumer multiple times and wasting resources. This provides **availability** of the blockchain services. Once the consumer is added to the network, we can know that the consumer is a legitimate user. This provides **reliability** of the network.

The “removeNode” function contains two requirements. The first is that the caller of the function is the utility. This stops any user in the blockchain network from accepting themselves or any other malicious users onto the network. This provides **legitimacy** of the consumers on the network and **integrity** of the electrical data to be sent by this consumer. The second is that the consumer the utility is trying to remove from the network must already be part of the network. This requirement stops the utility from removing a consumer multiple times and wasting resources or even attempting to remove a consumer that does not exist on the network to begin with. This provides **availability** of the blockchain services.

The “markDone” function contains one requirement: the caller of the function is the consumer. He/she needs to mark himself/herself as having deployed their data smart contract. This action should not be done by any other user since it could stop a legitimate consumer from deploying their data smart contract or accessing their data smart contract. This requirement provides **reliability** of the network by ensuring the legitimate consumers can access their smart contract to send and receive data on.

All the measures above provide **accountability** of the users and **transparency** and **auditability** of the data.

Second Smart Contract - Setting Up the Communication

This contract is deployed by the utility only once. Once deployed at the inception of the smart grid communication network, the factory smart contract becomes accessible by all other nodes on the blockchain since it is now part of the blockchain's ledger. This smart contract is deployed to allow the consumers to create their data smart contracts, which are needed for the data exchange between the consumer with the utility. The details of this smart contract can be found in subsection 6.2.2.

The "constructor" function is used by the utility to deploy the contract. The utility places its Ethereum address on the smart contract to be used in the creation of the data smart contracts. The utility's Ethereum address will be automatically included in the data smart contracts deployed by all the consumers. This provides **authenticity** of the utility in the data smart contracts. As noticed, this step provides **security** for the to-be-deployed data smart contracts by consumers.

The "createDataContract" function contains no requirements. It is important to note that it does not lead to any security breaches. Any user on the blockchain network will be able to create this data smart contract. If a legitimate consumer creates the data smart contract through the appropriate steps in the first smart contract, then the utility will access this smart contract, gather the electrical data sent from the consumer, and send load balancing data when needed. If an illegitimate user creates the data smart contract, then the utility will not access this smart contract since the user is not one of the accepted users on the network. This provides **reliability** and **security** of the network. It also provides **integrity** of the data supplied.

All the measures above provide **accountability** of the users and **transparency** and **auditability** of the data.

Third Smart Contract - Communicating

This contract is deployed by the consumer only once through the factory contract described in subsection 6.2.2. Once deployed by each consumer, the data smart contract becomes accessible by all other nodes on the blockchain since it is now part of the blockchain's ledger. This smart contract is deployed to allow the consumers to communicate their energy data securely and the utility to send load balancing data back to the consumer (demand-response). This preserves the two-way data flow for which the smart grid is known. The details of this smart contract can be found in subsection 6.2.3.

The "constructor" function is used by the consumer to deploy the contract through the factory smart contract. The consumer places his/her Ethereum address on the smart contract as the owner of the smart contract and the utility's Ethereum address to use for load balancing data to be sent and verified. This provides **authenticity** of the consumer and the utility and limits who can call the functions to **secure** the network.

The "sendData" function contains one requirement: the caller of the function is the consumer that owns the data smart contract which stops any malicious user in the blockchain network from sending incorrect electrical data on behalf of the consumer. This provides **integrity** of the electrical data sent by this consumer.

The "fixData" function contains one requirement: the caller of the function is the utility which stops any malicious user in the blockchain network from sending incorrect load balancing data on behalf of the utility. This requirement provides **integrity** of the load balancing data sent by the utility for this specific consumer.

All the aforementioned measures provide **accountability** of the users and **transparency** and **auditability** of the data.

6.5.2 Blockchain Properties

Various security properties characterize the blockchain platform. These described security properties, including integrity, availability, authenticity, transparency, auditability, accountability, anonymity, privacy, reliability, and termination provided by blockchain, are integral to our thesis work. All the security properties are ensured through the structure of the blockchain and the usage of smart contracts in Ethereum to provide secure communication between the smart meters and the utility.

Integrity

Blockchain's resistance to the modification of data can be accredited to their distributed ledgers. These ledgers are immutable meaning they cannot be changed (modified or removed). Blockchain is thus tamper-proof since any modification to the ledger will be detected.

Integrity in this architecture is ensured through the immutable ledger. Any transactions made are placed on this ledger and no modifications can be made to these transactions. The cryptographic hash function and signatures for transactions and blocks guarantee integrity.

The electrical data sent from the consumer through a transaction will be placed on this immutable ledger. The load balancing data sent from the utility through a transaction will also be placed on this immutable ledger.

Availability

Blockchain's decentralization (explained in subsection 2.3.1) provides availability since each node has a full copy of the ledger. Therefore, attacks on the ledger at certain nodes cannot cause damage to the entire network. There is no single server that has total control. In addition, transactions that already exist cannot be deleted which maintains data persistence.

Availability in this architecture is ensured through the distributed ledger. The blockchain network is a network of nodes connected together. Each node contains the ledger, making the transactions available at any time. The blockchain verification, authentication, and mining services are always available since no single point of failure exists which guarantees availability.

The electrical data sent from the consumer through a transaction and the load balancing data sent from the utility through a transaction are placed on the distributed ledger to be accessed at any time.

Authenticity

Blockchain's use of asymmetric cryptography (discussed in subsection 2.3.4) provides the tools needed to generate a digital signature (discussed in subsection 2.3.4) for a transaction. This authenticates the transaction that is sent. This transaction is further verified for authenticity by nodes in the network.

Authenticity in this architecture is ensured through the cryptographic digital signatures. The transactions made by the utility or the consumers will be signed using their private key. This can later be verified using the corresponding public key.

Smart contract functions were designed in a way to only allow specific users to call them as is shown in the description of the smart contracts of this architecture in section 6.2. Malicious users cannot access certain functions to change data stored in the smart contract.

The electrical data sent from the consumer through a transaction is signed by the consumer. The load balancing data sent from the utility through a transaction is signed by the utility.

Transparency

Blockchain's transparency of information is provided through a verified distributed ledger of transactions. It allows any of the nodes in the network (depending on the blockchain platform used) to view the transactions made.

Transparency in this architecture is ensured through the immutable distributed ledger. The electrical data sent from the consumer through a transaction and the load balancing data sent from the utility through a transaction are placed on the distributed ledger to be accessed at any time by anyone in the blockchain network.

Auditabiliy

Blockchain's transactions are traceable since every transaction made, added to a block, and broadcast to the network is documented in the distributed ledger. Tracking transactions is possible through the sue of the blockchain platform.

Data provenance can be seen through the distributed ledger. All transactions are placed on the ledger, so any address's transactions can be traced to the very first transaction.

Auditability in this architecture is ensured through the immutable ledger. The

transactions are stored on every node's ledger creating a log of all transactions occurring in the blockchain network.

The electrical data sent from the consumer through a transaction will be placed on this immutable ledger. The load balancing data sent from the utility through a transaction will also be placed on this immutable ledger.

Accountability

Blockchain's transparency and auditability provide the necessary tools to accommodate accountability. The distributed ledger can be referenced to check for any needed data related to a certain blockchain address.

Accountability in this architecture is ensured through the immutable ledger. The transactions contain the Ethereum address of the transaction creator. This ensures that the transaction is always linked back to the account making the transactions auditable.

The electrical data sent from the consumer through a transaction contains the consumer's Ethereum address. The load balancing data sent from the utility through a transaction contains the utility's Ethereum address.

Anonymity

Blockchain's blockchain-specific account per user provides anonymity. This account is used for all the user's transactions in the network. As long as there is no link between the user's blockchain address and the user's identity, the transaction remains anonymous.

Anonymity in this architecture is ensured through the Ethereum address. Each account has its own Ethereum address that is used in the blockchain network. The Ethereum address is unique to each account. The account does not need to provide any personal information; thus, the account is not linked to any personal information.

The electrical data is sent from the consumer through a transaction, using the consumer's Ethereum address. The load balancing data is sent from the utility through a transaction, using the utility's Ethereum address. Both of these addresses do not link back to any personal data of the user of the account thereby guaranteeing pseudo-anonymity.

Privacy

Privacy is ensured through the pseudo-anonymity of the blockchain addresses. The blockchain ledger is readable by any node in the network and transactions can be traced for every address but these addresses cannot be linked to real identities.

Privacy in this architecture is ensured through the Ethereum address. This address allows the consumer to use an account address instead of personal information to send the electrical data through transactions.

By remaining pseudo-anonymous, the consumer can send electrical data openly without worrying about the data being traced back to them personally. The data can only be traced back to the Ethereum address which, as mentioned, is not linked to the consumer's personal information.

The utility can send load balancing data openly without worrying about the data being traced back to the consumer personally since this consumer is pseudo-anonymous. The data can only be traced back to their Ethereum address which, as mentioned, is not linked to the consumer's personal information.

The electrical data is sent from the consumer through a transaction, using the consumer's Ethereum address. The load balancing data is sent from the utility through a transaction, using the utility's Ethereum address. Both of these addresses do not link back to any personal data of the user of the account and thus do not disclose who is sending the data.

Termination

In our smart contracts, there are no for loops written out to avoid issues leading to gas depletion at both the utility and consumer sides.

Termination in this architecture is ensured through the use of gas/gas limit (explained in subsection 2.4.7). Utilizing gas and gas limits forces the function to terminate whether complete or incomplete.

Reliability

Reliability in this architecture is ensured through the blockchain structure. The distributed ledger provides all the security properties mentioned in this section. The smart contract provides restrictions for the functions. Both the smart contract and ledger provide the reliability needed. The ledger makes sure the data has integrity and is available, authentic, auditable, transparent, and private. The ledger makes sure the user is anonymous and accountable. The smart contract

makes sure the transactions called are terminable. The smart contract is also the entity that contains all the rules that the users in the network must follow to enter the network and communicate data.

6.6 Limitations

Various security properties are achieved in this architecture, which are mentioned in section 6.5. These properties make the Ethereum blockchain a good solution for securing the two-way communication between the consumers and the utility. However, a few limitations arise such as issues with privacy using pseudo-anonymity, the cost of smart contracts, and scalability.

6.6.1 Privacy using Pseudo-anonymity

In the blockchain network, a user that has a blockchain account cannot be considered fully anonymous. Pseudo-anonymity is when a user is linked back to their blockchain address but not to any of their personal information. All transactions are stored publicly on the blockchain ledger and are visible to anyone to analyze and interpret. Anonymity of the sender depends on the pseudonym not being linked to his/her true identity. Thus, a user can preserve his/her privacy as long as pseudonym is not linked back to the individual. If a link is made, the identity of the user is revealed and the pseudo-anonymity is broken. All transactions made by the user and previously identified through the blockchain address can now be traced back to the user's identity.

Tools are used by attackers to break the link between the user's account and the user's identity called deanonymization. Users can attempt to avoid linking their blockchain address with any personal information, but it is not very feasible. Users can turn to virtual private networks (VPNs) and other services (Tor [144], Tumbler/Mixer[5]). Users can even use a new blockchain address for each transaction. Each of these solutions comes with its own set of problems.

6.6.2 Cost of Smart Contracts

The deployment and interaction with the three smart contracts discussed in section 6.2 subsections 6.2.1, 6.2.2, and 6.2.3 is costly as demonstrated in the results section 6.4 of this chapter. The cost incurred - the use of gas in the smart contracts - could be reduced using certain optimizations.

Data Types

The Ethereum Virtual Machine, when dealing with storage, manages storage slots of size 256 bits. The use of variables with smaller data type sizes will force the EVM to fill in the rest of the bits with zeros which costs gas. Another thing that costs gas when storing small variable types are the calculations that are performed. The calculations are performed on integer variables of size 256 thus

[5]https://cryptomixer.io/

any variable with any other type and size will need to be converted before the calculations takes place.

Data in Contract Bytecode

The data that needs to be stored in the blockchain from the beginning can be placed there through the bytecode of the smart contract. There is no need to waste transactions and pay fees when the variables can be placed there before deployment. This allows the contract to save a lot on gas consumption. However, these values cannot be altered afterwards.

Packing Variables

When data is stored on the blockchain smart contract, it must be done in a way that maintains the order of the different variables types. Variables of the same type should be placed after one another to allow them to be tightly packed and reduce padded zeros.

6.6.3 Scalability

Both the utility and the consumers in the smart grid communication network will have to send an abundance of messages. The consumers will each have to send data at intervals of 15 minutes, 30 minutes, or an hour. The utility will have to send load balancing data to consumers when required. All these transactions are not scalable on the Ethereum blockchain network. As described in table 2.1 from section 2.3, the number of transactions per second in the Ethereum network is 15. This scalability issue will have to be amended in order to allow our solution to work properly and scalably.

Chapter 7

Architecture 2 (Part A)

The first part of Architecture 2, discussed in this chapter, explores a method for sending encrypted data using a combination of the Ethereum platform and cryptographic tools. We propose to create two separate smart contracts each for a different functionality. The first smart contract is the contract that allows consumers to join the smart grid communication network. It greatly resembles the first smart contract in the first architecture (subsection 6.2.1). It is called the simulation smart contract. The second smart contract is the contract that defines the protocols used in the transmission of data from the consumer to the utility and vice versa. It is called the communication smart contract. These smart contracts have been carefully designed to reduce the cost of using them (to understand the concept of smart contract fees and transaction costs refer to subsections 2.4.2 and 2.4.9). This chapter shows the scenarios the architecture will follow in section 7.1, the smart contract details in section 7.2, the user interface used to interact with the smart contracts in section 7.3, the results in section 7.4, the security properties achieved in section 7.5, and the limitations of this architecture in section 7.6.

7.1 Scenario

Using a control flow graph (CFG), we display the order of execution of the calls and updates to the smart contract. It shows all possible steps that can be executed in our program.

7.1.1 Initial Contract Deployment

Figure 7.1 shows the steps taken by the utility to deploy the two smart contracts needed to join and communicate in the smart grid communication network. The utility initially deploys the simulation contract which would lead to a transaction that would update the blockchain state (step 1). This contract deals with consumers requesting entry and utility accepting consumers into the network.

If the simulation contract is deployed successfully (step 2), the utility can deploy the communication contract (step 3) which would lead to a transaction that would update the blockchain state (step 4). This contract deals with the consumer and utility data interaction. If the simulation contract is not deployed successfully, the utility will try to deploy the simulation contract again.

Deploying the simulation contract and the communication contract lead to changes in the Ethereum state.

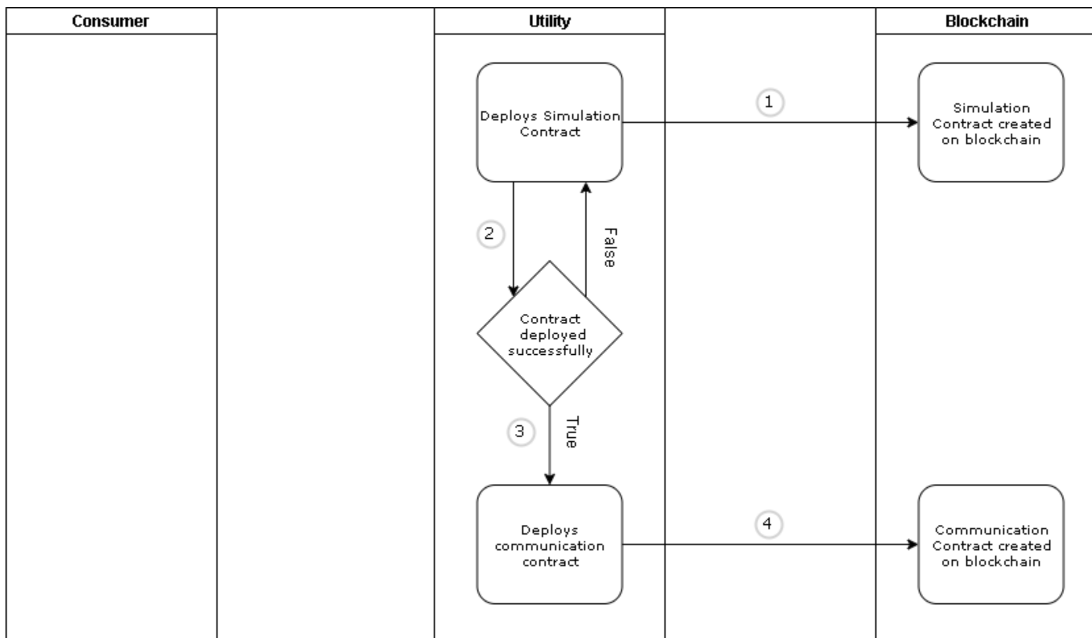

Figure 7.1: Architecture 2A: Initial Contract Deployment

7.1.2 Contract Interaction

Figure 6.2 from chapter 6 shows the steps taken by the consumer to try and enter the network. The consumer can request entry that would lead to a transaction that would update the blockchain state (step 1).

Figure 6.2 from chapter 6 also shows the steps taken by the utility to accept or reject the consumer. The utility queries for requests from consumers. The utility finds a request (step 2). If there is a request and the utility wishes to reject the consumer, then it would lead back to a state of waiting for a request. For instance, the utility can choose to reject a consumer if they did not register with the utility beforehand. If there is a request and the utility wishes to accept the consumer, then it would lead to a transaction that would update the blockchain state (step 3). If the utility does not find a request, it waits 15 minutes before rechecking (step 4). The 15 minute wait can be updated to 30 minutes or an hour depending what the utility wants. The 15 minute intervals will recur throughout the architecture and can be updated to 30 minutes or an hour also depending on what the utility wants. The utility can also remove a consumer from the network that would lead to a transaction that would update the blockchain state (step 5).

Requesting, accepting, and removing consumers leads to changes in the Ethereum state. Querying for requests does not lead to changes in the Ethereum state.

Figure 7.2 shows the steps taken by the consumer to send data. Electrical data can be sent from the consumer to the utility through an event and can be received by the utility. The consumer first encrypts the electrical data (step 1) and attempts to send an event containing the encrypted electrical data. The consumer checks if it is allowed to send data (step 2). If the consumer is not allowed to send data, then it would lead back to a state of waiting to send electrical data. If the consumer is allowed to send data, then it would lead to a transaction that would update the blockchain state (step 3). The consumer attempts to send data every hour (step 4). The 15 minute intervals or 1 hours intervals will recur throughout the architecture and can be updated depending on what the utility wants.

Figure 7.2 also shows the steps taken by the utility to send data. Load balancing data can be sent from the utility to the consumer through an event and can be received by the consumer. The utility first encrypts the load balancing data (step 5) and sends an event containing the encrypted load balancing data that would lead to a transaction that would update the blockchain state (step 6).

Sending data, electrical data by the consumer and load balancing data by the utility, leads to changes in the Ethereum state.

Every 15 minutes the consumer can get an event containing the encrypted load balancing data (step 8) if the utility has sent load balancing data. The data is decrypted for use (step 7). Every hour the utility gets an event containing the encrypted electrical data (step 10). The data is decrypted for use (step 9). The use of cryptographic techniques here is to ensure the data remains confidential, unchanged, and authentic. These properties will be explained further in section 7.5.

The consumer receiving load balancing data and the utility receiving electrical data does not lead to changes in the Ethereum state.

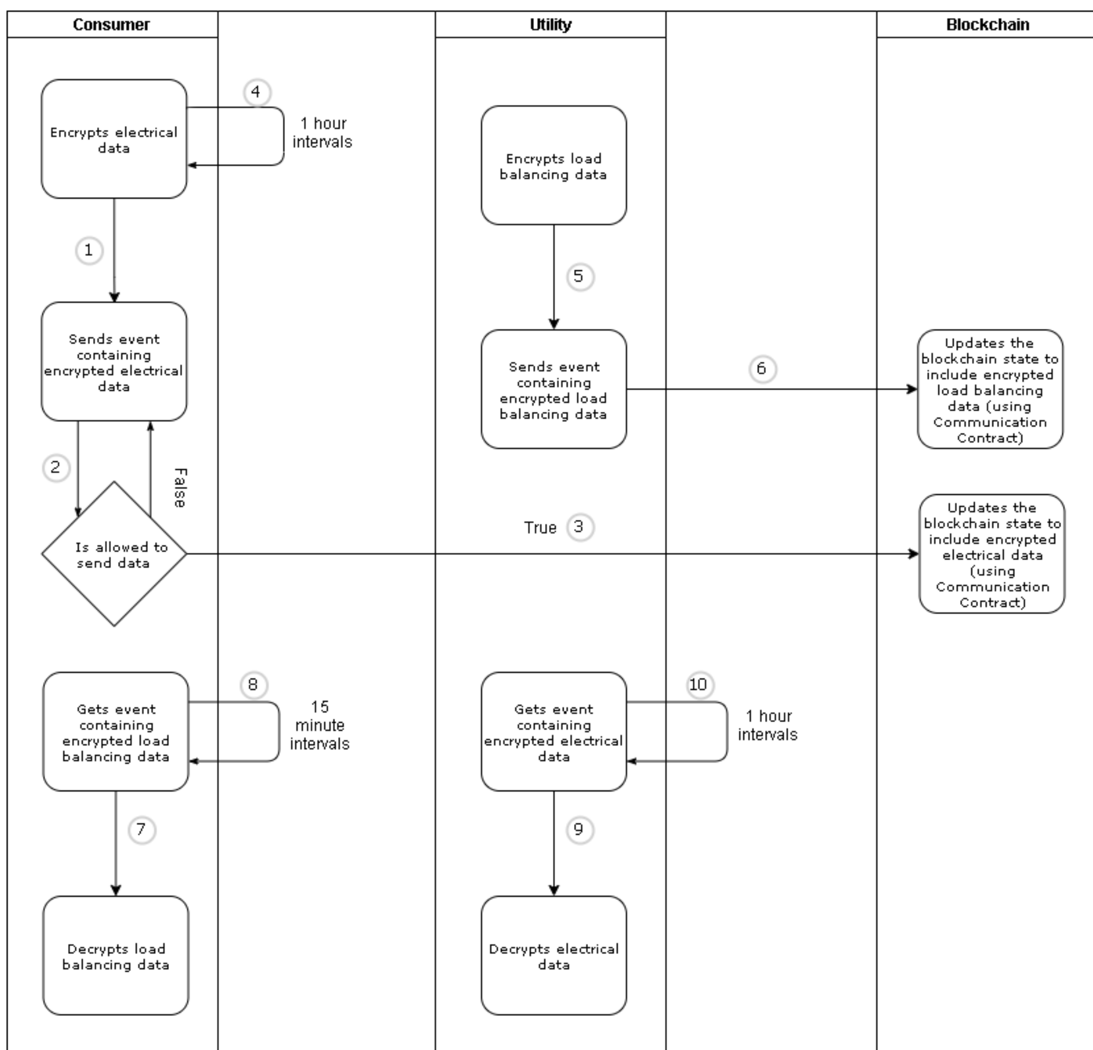

Figure 7.2: Architecture 2A: Contract Interaction (Part 2)

7.2 Smart Contract Details

Now that we have gone through the control flow of the smart contracts' executions, we answer the following questions, for each smart contract: By who was the contract deployed? When was it deployed? For what purpose? What are the contracts' contents?

7.2.1 First Smart Contract - Joining the Network

This contract is deployed by the utility only once at inception. Figure 7.3 shows a high level view of the smart grid communication network in relation to the first smart contract. This figure illustrates a city in Qatar that contains 10 households/establishments (HHs) numbered 1 through 10 and the utility numbered 11. All of the establishments numbered 1 through 10 contain smart meters that measure their electrical consumption.

There are two distinguishable components in figure 7.3. The first component is the blockchain network. This network contains all the households and the utility. The HHs and the utility are part of the peer-to-peer Ethereum blockchain network as illustrated by the road which connects all the HHs and establishments. The HHs are nodes in the blockchain network and have access to the distributed blockchain ledger. Node 11, the utility, contains the first smart contract known as the simulation smart contract. Once deployed at the inception of the smart grid communication network, the smart contract becomes accessible by all other nodes on the blockchain since it is now part of the blockchain's ledger. This smart contract is deployed to allow the consumers to join the network, which is composed of the different consumers containing legal smart meters registered at the utility.

The second component is the distributed ledger. This is the Ethereum ledger that every node will have access to. This ledger will contain all the transactions that have been mined and added to the blockchain. The transactions include all the updates to the Ethereum state (explained in subsection 2.4.5).

The symbol table explains that the different nodes 1 through 11 can send a transaction "T" at a certain time "t" that is added to the distributed ledger. The legend displays the objects that exist in the smart grid communication network.

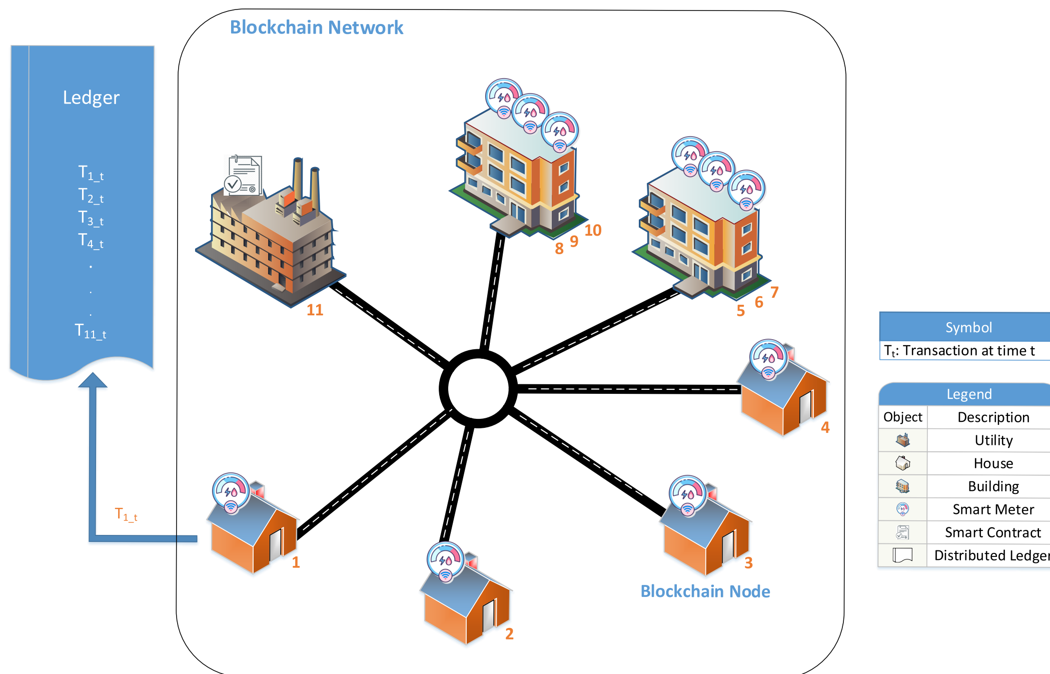

Figure 7.3: Architecture 2A: Smart Contract #1 Distribution

As we've explained in subsection 2.4.2, variables are set up to be interacted with, and functions exist to allow actions to occur on the blockchain. The detailed code of the first smart contract variables and functions is available in Appendix A listing A.4. Here, we will briefly explain the variables, functions, and their purpose in the smart contract.

Listing 7.1 displays the variable names and descriptions in this contract. These variables are used in functions we discuss below.

- **Line 1 - countNodes**: Number of consumers accepted onto the network
- **Line 2 - publicKey**: Public key of the utility
- **Line 3 - utility**: Utility's Ethereum address
- **Line 5 - requestedNodes**: Array of requesters (represented by the node structure)
- **Line 7 - hasRequested**: Map of requested consumers which are consumers that want to enter the network
- **Line 8 - isAccepted**: Map of accepted consumers which are consumers that have been accepted onto the network

- **Lines 11 till 16 - Node**: Node structure that contains properties such as the consumer's public key, smart meter id, Ethereum address, and contract completion attribute - lines 12, 13, 14, 15 respectively. It will be used to create requests by consumers

The array of requesters will be used by the utility to display the requesters in the user interface whereas the mapping of requesters will be used by the smart contract to insure that the requester cannot request entry again (security measure).

```
uint32 public countNodes;
string constant public publicKey = "public key";
address public utility;

Node[] public requestedNodes; //to be added to network by utility

mapping(address => bool) private hasRequested;
mapping(address => bool) public isAccepted;

//request for node
struct Node {
    string key;
    uint32 smId;
    address add;
    bool complete;
}
```

Listing 7.1: Variables

The functions included in this smart contract are listed below and will be explained in listings 7.2, 7.3, 7.4, and 7.5 respectively whereas the "getRequestersCount" function is provided in listing 6.6 in subsection 6.2.1.

- Constructor
- Request entry
- Accept node
- Remove node
- Get requesters count

We assume that the smart grid communication network is starting with no consumers (smart meters). The "constructor" function - line 3 - initializes the variables in listing 7.1 as seen in listing 7.2. The Ethereum utility address is set to the caller of the constructor function "msg.sender" which is the utility - line 4. The number of nodes is set to 0 since we assume the network is starting with no consumers - line 5.

```
1 //only called once by utility
2 //since deployed by utility
3 function Simulation() public {
4     utility = msg.sender;
5     countNodes = 0;
6 }
```

Listing 7.2: Constructor

Via this smart contract, the consumer (smart meter) must request entry into the network to become part of it. Using the "requestEntry" function - line 3 - shown in listing 7.3, the consumers can request to enter the network providing certain properties to be approved by the utility. Before the function can be completed, a rule must be checked. The function must not be called by a consumer that has previously requested entry (called this function) - line 6. This restriction forces the function to be called only when necessary thereby alleviating any problems with overusing this function and stressing the network. The consumer is added to the mapping of requesters and array of requesters and waits for the utility to accept it. If these conditions are not satisfied, the function will throw an exception which we have discussed in subsection 7.3.1. If it is satisfied, a node structure is created with the passed properties - line 8. The consumer's node is now added to the mapping of requesters - line 9 - and array of requesters - line 10 - and waits for the utility to accept it.

```
1 //node requests access to network
2 //must be accepted by utility
3 function requestEntry(uint32 smId, string key) public {
4     //shouldn't be a requestor
5     //we don't have to check if it isn't already be a Node becuase
          once accepted we do not sent hasRequested to false...
6     require(!hasRequested[msg.sender]);
7 
8     Node memory r = Node(key, smId, msg.sender, false);
9     requestedNodes.push(r);
10    hasRequested[msg.sender] = true;
11 }
```

Listing 7.3: Request entry

The utility can accept the entry request using the "acceptNode" function - line 2 - shown in listing 7.4 or reject it (by not accepting it). If the caller of the function is not the utility, the function will throw an exception which we have discussed in subsection 7.3.1. The node is retrieved from the array of requesters - line 4. If the consumer is not already part of the network, the function continues execution - line 11, otherwise, the function will throw an exception which we have discussed in subsection 7.3.1. Once the utility accepts the consumer, the consumer is added to the mapping of accepted consumers - line 14 - and the number of nodes is increased by one - line 15. The node is now marked as complete meaning it can be

considered part of the network - line 20.

The utility also has the ability to accept an entry and then remove it later for any reason using the "removeNode" function - line 2 - in listing 7.5. If the caller of the function is not the utility, the function will throw an exception which we have discussed in subsection 7.3.1. Removal reasons can be caused by an entity requesting to stop its power service due to relocation, or disciplinary actions taken by the utility against malicious consumers. The node is retrieved from the array of requesters - line 4. If the consumer is part of the network (can be found in mapping of accepted consumers), the function continues execution - line 7, otherwise, the function will throw an exception which we have discussed in subsection 7.3.1. Once the utility removes the consumer, the number of nodes is decreased by one - line 9 - and the consumer is removed from both mappings (requesters and accepted consumers - lines 10, 11 respectively) thus clearing it from the network.

```
1 //can only add node if the utility accepts
2 function acceptNode(uint32 index) public restrictedUtility {
3     //the node requested
4     Node storage r = requestedNodes[index];
5 
6     //check if this node is valid
7     //based on smId db at utility
8     //done manually by calling verify function -> comparing values
          to values they have -> making the call
9 
10    //shouldn't already be a Node
11    require(!isAccepted[r.add]);
12 
13    //add node to list that needs to create a contract
14    isAccepted[r.add] = true;
15    countNodes ++;
16 
17    //mark as complete in request
18    //for ui:
19    //when complete true the row is disabled
20    r.complete = true;
21 }
```

Listing 7.4: Accept node

```
1 //utility can remove a node from network
2 function x_removeNode(uint32 index) public restrictedUtility {
3     //the node requested
4     Node storage r = requestedNodes[index];
5 
6     //should already be a Node
7     require(isAccepted[r.add]);
8 
9     countNodes --;
10    hasRequested[r.add] = false;
11    isAccepted[r.add] = false;
12 }
```

Listing 7.5: Remove node

In addition, the contract includes a function that allows the utility to view how many requesters there are in the array of requesters using "getRequestsCount" function which provided in chapter 6's listing 6.6 subsection 6.2.1.

7.2.2 Second Smart Contract - Communicating

This contract is deployed by the utility only once at inception. Figure 7.4 shows a high level view of the smart grid communication network in relation to the second smart contract. The specifics of the city, households, and utility can be found in subsection 7.2.1.
The symbol table explains that the different nodes 1 through 11 can send a message with the data "M" at a certain time "t". It shows the encrypt "E" and decrypt "D" symbols that are used to encrypt and decrypt the data sent to and received from the distributed ledger respectively. The legend displays the objects that exist in the smart grid communication network.

Node 11, the utility, contains the second smart contract known as the communication smart contract. Once deployed at the inception of the smart grid communication network, the smart contract becomes accessible by all other nodes on the blockchain since it is now part of the blockchain's ledger. This smart contract is deployed to allow the consumers to communicate their energy data securely and the utility to send load balancing data back to the consumer (demand-response). This preserves the two-way data flow for which the smart grid is known.

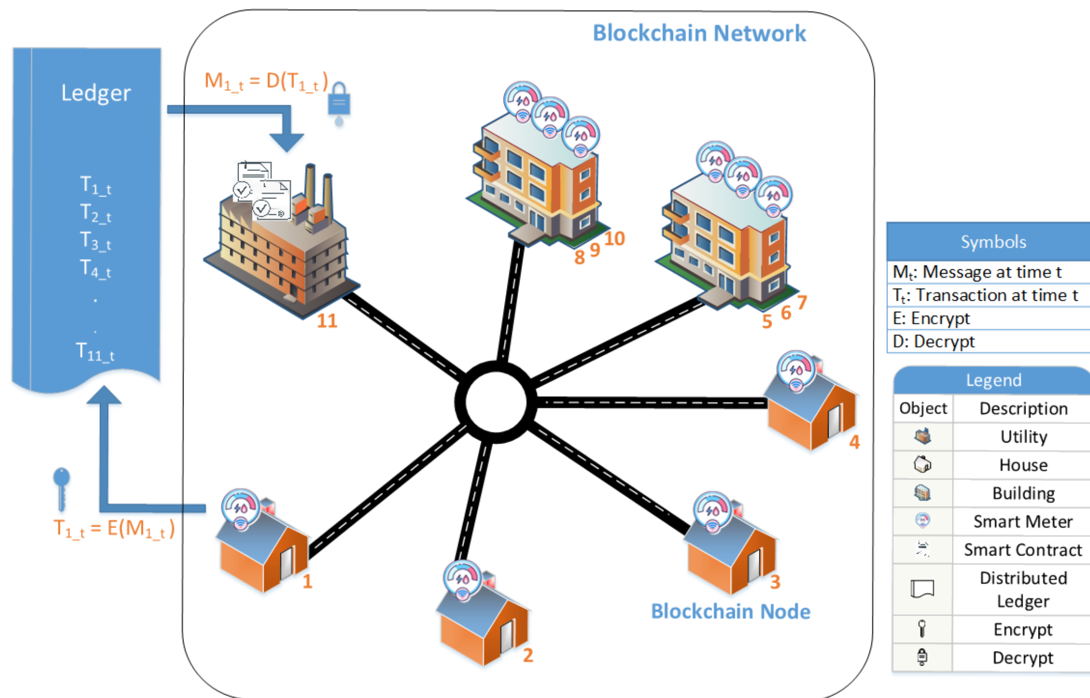

Figure 7.4: Architecture 2A: Smart Contract #2 Distribution

As we have explained in subsection 2.4.2, variables are set up to be interacted with, and functions exist to allow actions to occur on the blockchain. The detailed code of the second smart contract variables and functions is available in Appendix

A listing A.5. Here, we will briefly explain the variables, functions, and their purpose in the smart contract.
Listing 7.6 displays the variable names and descriptions in this contract. These variables are used in functions we discuss below.

- **Line 1 - S**: Simulation contract instance declaration
- **Line 4 - DataSent**: Event declaration for electrical data to be sent by the consumer that returns the consumer's Ethereum address, the timestamp of the data, and the value of the data (the concept of smart contract events is discussed extensively in subsection 2.4.8)
- **Line 5 - LoadBalancingData**: Event declaration for load balancing data to be sent by the utility that returns the consumer's Ethereum address (the consumer to which the data should be sent) and the value of the data

Both of these events will be discussed further below.

```
1 Simulation private S;
2
3 //events
4 event DataSent(address _from, bytes32 indexed _timestamp, string
    _value);
5 event LoadBalancingSent(address indexed _to, string _value);
```

Listing 7.6: Variables

The functions included in this smart contract are listed below and will be explained in listings 7.7, 7.8, and 7.9 respectively.

- Constructor
- Send data
- Fix data

We assume that the consumers using this smart contract have been accepted into the network and are accessing this contract to send electrical data to the utility and to get load balancing data from the utility. The "constructor" function - line 3 - initializes the variables above as seen in listing 7.7. The address that is passed as a parameter is the simulation smart contract address - line 3. This address is produced when the utility deploys its simulation smart contract. It is passed to the constructor so that the communication smart contract can make use of the components in the simulation smart contract. Using this smart contract address, the simulation contract may be referenced - line 4.

```
//constructor
//takes the simulation contract to refer to it later
function Communication(address a) public {
    S = Simulation(a);
}
```

Listing 7.7: Constructor

Via this smart contract, the consumer can send electrical data to the utility. Sending data using the "sendData" function - line 3 - in listing 7.8 will emit an event containing the electrical data in addition to the date and time (following a certain format which is year-moth-day-hour-minute). Before doing so, the caller of this function must be proven to be a member of the smart grid communication network - line 4. If this condition is not satisfied, the function will throw an exception which we will further discuss in subsection 7.3.2. If it is satisfied, the consumer's Ethereum address, the timestamp of the data, and the value of the data are sent as an event - line 5. The concept of smart contract events is discussed extensively in subsection 2.4.8. By checking the simulation contract's mapping of accepted consumers, the smart contract can ensure that the caller of this function is part of the communication network before allowing the transaction to be processed.

The utility can then watch for the event at specific intervals. It will watch for events related to electrical data sent by the consumers. These events are filtered based on what timestamp is needed (to be discussed in subsection 7.3.2).

```
//send the Data
//timestamp should follow certain convention yyyymdhm
function sendData(bytes32 timestamp, string value) public {
    require(S.isAccepted(msg.sender));
    DataSent(msg.sender,timestamp,value);
}
```

Listing 7.8: Send data

The utility can also send load balancing data to the consumer. Sending data using the "fixData" function - line 3 - in listing 7.9 will emit an event containing the load balancing data. Before doing so, the caller of this function must be proven to be the utility since load balancing data can only be sent by the utility (whose address is derived from the simulation contract) - line 4. If this condition is not satisfied, the function will throw an exception which we will further discuss in subsection 7.3.2. If it is satisfied, the consumer's Ethereum address and the value of the data are sent as an event - line 5. The concept of smart contract events is discussed extensively in subsection 2.4.8.

The consumer can then watch for the event at specific intervals. They will watch

for events related to load balancing data sent by the utility. These events are filtered based on who their Ethereum address (to be discussed in subsection 7.3.2).

```
//utility
//send load balancing data
function fixData(address to, string value) public {
    require(msg.sender == S.utility());
    LoadBalancingSent(to, value);
}
```

Listing 7.9: Fix data

7.2.3 Summary

A smart grid communication network will start out empty with only the utility present. The utility initially deploys two smart contracts. The first smart contract (simulation contract) will allow consumers to enter the smart grid communication network and the second smart contract (communication contract) will allow the consumers and the utility to communicate data.

The consumers that wish to join the network and become part of the smart grid can do so by requesting to join. The utility can then accept or reject this consumer. Once accepted, the consumer may now proceed to send and receive data. If rejected, the consumer cannot send and receive data.

The consumer can now use the second smart contract (communication contract) to send data periodically to the utility. The utility can also send load balancing data to the consumer through the respective communication contract.

7.3 User Interface

To complete the preventative proposed solution, we set up a user interface for user data interaction. The user may interact with Ethereum blockchain using the Rinkeby Test Network discussed above. Sections 7.1 and 7.2 discussed the general architecture execution and details and this section will describe the user interface put in place to interact with the Ethereum network smart contracts. Some front-end components provide the consumer access to the different smart contracts deployed by the utility or consumers. Other front-end components provide the utility access to the different smart contracts deployed by the utility or consumers. The user interface does not show the deployment of the simulation and communication smart contracts (found in subsection 5.5) since it should be completed by the utility before any interaction with the smart contracts can occur.

7.3.1 First Smart Contract - Joining the Network

There are minor differences between the first smart contract from Architecture 1 in chapter 6 and the first smart contract from Architecture 2 Part A.

Figure 7.5 shows the user interface where a consumer can request entry into the smart grid communication network. The smart meter id field needs to be filled in which the utility needs (to confirm the user is authentic).

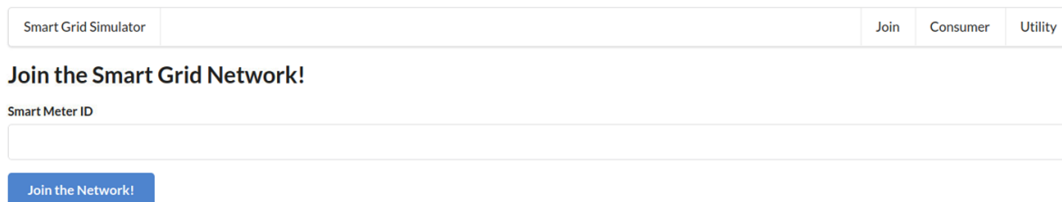

Figure 7.5: Architecture 2A: Join

Listing 7.10 shows the code used to interact with the simulation smart contract (first smart contract) using the interface in figure 7.5.

The public-private key pair explained in section 5.8 listing 5.6 is created by the consumer - line 13 - and the private key is stored locally - line 14 - as explained in subsection 5.8.2.

The asynchronous[1] transaction - line 18 - calls the smart contract function "re-

[1]The async keyword is used in the function definition to declare it is asynchronous. The await keyword used to show the action of waiting for a promise. The function is paused in a non-blocking way until the promise settles. The value is returned if the promise fulfills.

questEntry" (discussed in subsection 7.2.1 listing 7.3) that takes two parameters one of which is evident in figure 7.5 and the the public key created for the consumer. The function uses the "send" keyword which leads to a transaction that changes the Ethereum state. As discussed in subsection 2.4.5 the "from" keyword shows who is creating the transaction.

```
1 onSubmit = async (event) =>{
2     event.preventDefault();
3 
4     try{
5         //enable use of web3 instance
6         await ethereum.enable();
7 
8         //get the accounts provided by metamask
9         const accounts = await web3.eth.getAccounts();
10 
11         //store private key
12         // console.log(k.publicK);
13         var k = new (require('../encryption/getKeys'))();
14         localStorage.setItem(accounts[0], k.privateK);
15 
16         //call the smart contract function to request entry
17         //2 params: smart meter id, and public key
18         await simulation.methods.requestEntry(web3.utils.utf8ToHex(
               this.state.smId) , k.publicK).send({from: accounts[0]});
19     } catch(err) {
20         this.setState({errorMessage: err.message});
21     }
22 };
```

Listing 7.10: Join Code

The interaction between the consumer and the first smart contract's "requestEntry" function could either be successful or unsuccessful due to a certain property not being satisfied in the smart contract. This could happen for many reasons:

- The consumer has previously requested entry into the network
- The consumer has not provided enough gas for the transaction to succeed

Figures 7.6 and 7.7 show the user interface where the utility can accept consumers based on the properties provided. The utility may also reject the consumers by not accepting them. We can notice the new field labeled "Public Key" that contains the consumer's shared public key. Beneath the list of consumers to accept/remove, there are two values: the total number of requesters and the number of nodes accepted into the network both queried from the smart contract.

Pending Requests

Smart Meter ID	MAC Address	Node Network Address	Approve	Remove

Found 0 request(s)
Found 0 current node(s)

Figure 7.6: Architecture 2A: Accept Initial

Smart Grid Simulator | Join | Consumer | Utility

Pending Requests

Smart Meter ID	Node Network Address	Public Key	Approve	Remove
1997	0x88B1A85759a33971f6647d77C7c691BB91c125bc	-----BEGIN PUBLIC KEY----- MFwwDQYJKoZIhvcNAQEBBQADSwAwSAJBAMIh2IZ3F97UrEefMTmMk3gawrnqLCR7 +hvniNBIvfq4VcOJzEOvLtR1C0YSNpRhE2wMNo5cpUxoIT+DG8d0WrcCAwEAAQ== -----END PUBLIC KEY-----	Approve	Remove
01	0x3c3C6893c843Dc25B932F00cB4467e4dF2C1c77F	-----BEGIN PUBLIC KEY----- MFwwDQYJKoZIhvcNAQEBBQADSwAwSAJBAJBrZ3UuZRbI915Kg4JC8bcabQjIDmv7 m/rOCroUJS4MhmzwynKSwoTm9HF1wRW06dnSQWMqZF+EfngolugCLGUCAwEAAQ== -----END PUBLIC KEY-----	Approve	Remove
17	0x8D23396c2Fc2ABD2F28a983dFECf289D24A69A3F	-----BEGIN PUBLIC KEY----- MFwwDQYJKoZIhvcNAQEBBQADSwAwSAJBAI8u5G5PgVOc+kNXkMbcpfDIg9ibUJx1 wuWrzxfWssWwg3I0nLoq0J1fZOqp7IKWOJE+3Pt6J7Cbi7/loBwSm/MCAwEAAQ== -----END PUBLIC KEY-----	Approve	Remove

Found 3 request(s)
Found 0 current node(s)

Figure 7.7: Architecture 2A: Accept

The utility can navigate to the page used to send load balancing data to a specific consumer through the hyperlink seen on the smart meter id. Figures 7.8 and 7.9 show the URL created when the utility clicks on the hyperlink marked for the smart meter id. It includes the smart meter's Ethereum address: "0x88B1A85759a33971f6647d77C7c691BB91c125bc" as shown in figure 7.8 with the smart contract Ethereum address (converted to HEX for simplicity of the URL) as shown in figure 7.9. The page the utility will navigate to will also contain the electrical data received from that consumer in real-time or by querying using a timestamp. This page will be further discussed in subsection 7.3.2.

ⓘ localhost:3000/data/read/0x88B1A85759a33971f6647d77C7c691BB91c125bc/

Figure 7.8: Architecture 2A: URL - Account Address

/0x2d2d2d2d2d424547494e205055424c4943204b45592d2d2d2d2d0a4d4677774451...

Figure 7.9: Architecture 2A: URL - Contract Address

Listing 7.11 shows the code used to navigate to the selected consumer's page which is used by the utility to send load balancing data and receive electrical

data through the URL seen in figures 7.8 and 7.9. Line 2 shows the route taken that navigates to the next page including the smart meter's Ethereum address: "0x88B1A85759a33971f6647d77C7c691BB91c125bc" as shown in figure 7.8 with the smart contract Ethereum address (converted to HEX for simplicity of the URL) as shown in figure 7.9. Line 3 shows the hyperlink marked for the smart meter id.

```
1 <Table.Cell>
2     <Link route={`/data/read/${request.add}/${web3.utils.utf8ToHex(
        request.key)}`}>
3         <a>{web3.utils.hexToUtf8(request.smId)}</a>
4     </Link>
5 </Table.Cell>
```

Listing 7.11: URL Code

Listing 7.12 shows the code used to interact with the simulation smart contract (first smart contract) using the interface in figure 7.7.

The asynchronous transaction - line 4 - calls the smart contract function "countNodes" (discussed in subsection 7.2.1 listing 7.1) that doesn't take any parameters. This variable is placed below the table of pending requests indicating the number of nodes accepted into the network. The function uses the "call" keyword which does not lead to a change the Ethereum state.

The asynchronous transaction - line 8 - calls the smart contract function "getRequestsCount" (discussed in subsection 7.2.1 listing 7.1) that doesn't take any parameters. This variable is placed below the table of pending requests indicating the number of nodes requesting entry into the network. The function uses the "call" keyword which does not lead to a change the Ethereum state.

The asynchronous transaction - line 14 - calls the smart contract getter function for the array of requesters "requestedNodes" (discussed in subsection 7.2.1 listing 7.1) that takes one parameter. The array elements are the requesters displayed in the table of pending requests where each requester is placed in a row of the table. The function uses the "call" keyword which does not lead to a change the Ethereum state.

```
1 static async getInitialProps(props) {
2     //call the smart contract function to get number of nodes in
         network
3     //0 params
4     const currentNodes = await
         simulation.methods.countNodes().call();
5 
6     //call the smart contract function to get the number of
         requesters
7     /0 params
8     const requestCount = await
         simulation.methods.getRequestsCount().call();
9 
10    const requests = await Promise.all(
11        Array(parseInt(requestCount)).fill().map((element,index) =>
              {
12            //call the smart contract function to mark node as
                 having deployed data contract
13            //0 params
14            return simulation.methods.requestedNodes(index).call()
15        })
16    );
17 
18    return {requests, requestCount,currentNodes};
19 }
```

Listing 7.12: Get Counter Code

Listing 7.13 shows the code used to interact with the simulation smart contract (first smart contract) using the interface in figure 7.7. The asynchronous transaction - line 6 - calls the smart contract function “acceptNode” (discussed in subsection 7.2.1 listing 7.4) that takes one parameter. The function uses the “send” keyword which leads to a transaction that changes the Ethereum state.

```
1 onApprove = async () => {
2     const accounts = await web3.eth.getAccounts();
3 
4     //call the smart contract function to accept node
5     //1 param: id in list
6     await simulation.methods.acceptNode(this.props.id).send({from:
         accounts[0]});
7 };
```

Listing 7.13: Accept Code

The interaction between the utility and the first smart contract's "acceptNode" function could be either successful or unsuccessful due to a certain property not being satisfied in the smart contract. This could happen for many reasons:

- The caller of the function is not the utility
- The utility has previously accepted the consumer into the network
- The utility has not provided enough gas for the transaction to succeed

Figure 7.10 shows the user interface where the utility may remove consumers. The figure shows that the consumers have already been accepted onto the network (no accept button). Only these consumers may be removed.

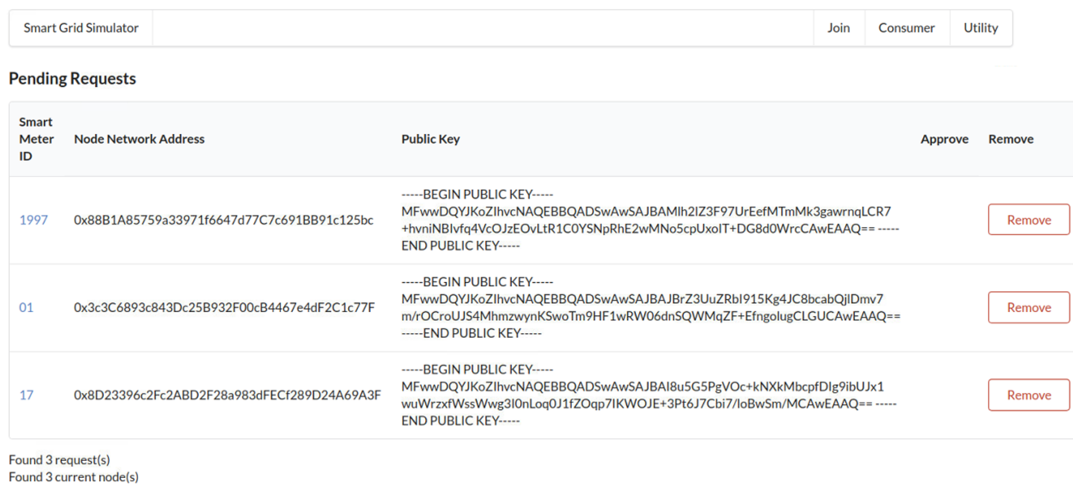

Figure 7.10: Architecture 2A: Remove

Listing 7.14 shows the code used to interact with the simulation smart contract (first smart contract) using the interface in figure 7.10. The asynchronous transaction - line 6 - calls the smart contract function "removeNode" (discussed in subsection 7.2.1 listing 7.5) that takes one parameter. The function uses the "send" keyword which leads to a transaction that changes the Ethereum state.

```
1 onRemove = async () => {
2     const accounts = await web3.eth.getAccounts();
3
4     //call the smart contract function to remove node
5     //1 param: id in list
6     await simulation.methods.removeNode(this.props.id).send({from:
          accounts[0]});
7 };
```

Listing 7.14: Remove Code

The interaction between the utility and the first smart contract's "removeNode" function could be either successful or unsuccessful due to a certain property not being satisfied in the smart contract. This could happen for many reasons:

- The caller of the function is not the utility
- The utility has not previously accepted the consumer into the network
- The utility has not provided enough gas for the transaction to succeed

7.3.2 Second Smart Contract - Communicating

Figure 7.11 shows the user interface with which the consumer can interact. The consumer can send electrical data to the utility in figure 7.12 and can receive load balancing data from the utility in figure 7.13.

Figure 7.11: Architecture 2A: Consumer Interface

Figure 7.12 shows the user interface where a consumer can input the electrical data needed by the utility and send it.

Figure 7.12: Architecture 2A: Send Electrical Data

Listing 7.15 shows the code used to interact with the communication smart contract (second smart contract) using the interface in figure 7.12.

The public-private key pair explained in section 5.8 listing 5.6 and created by the utility (explained in subsection 5.8.1) is used. The consumer extracts the utility's public key and converts it to the appropriate RSA key format (explained in listing 5.7) - line 1.

Line 13 shows the combination of the current year, month, date, hour, and minute needed to form a timestamp the utility will use to find the electrical data.

The utility's public key is used to encrypt the electrical data sent - line 17.

The asynchronous transaction - line 21 - calls the smart contract function "sendData" (discussed in subsection 7.2.2 listing 7.8) that takes two parameters, one of which is the timestamp explained in line 13 and the other is the encrypted electrical data explained in line 17. The function uses the "send" keyword which leads to a transaction that changes the Ethereum state.

```
1 import key from '../../encryption/publicKey';
2
3 onSubmit = async (event) => {
4     event.preventDefault();
5     await ethereum.enable();
6
7     try {
8         const accounts = await web3.eth.getAccounts();
9         const today = new Date();
10
11        //used for the timestamp
12        //gets current date in specific format
13        const time = "" + today.getFullYear() +
              (today.getMonth()+1) + today.getDate() +
              today.getHours() + today.getMinutes();
14        console.log(time);
15
16        //encrypt and send
17        const enc = key.encrypt(this.state.data,'base64');
18
19        //call the smart contract function to send electrical data
20        //2 params: timestamp, encrypted electrical data
21        await
              communication.methods.sendData(web3.utils.utf8ToHex(time)
              , web3.utils.utf8ToHex(enc)).send({from: accounts[0]});
22
23    } catch(err) {
24        this.setState({errorMessage: err.message});
25    }
26 }
```

Listing 7.15: Send Electrical Data Code

The interaction between the consumer and the second smart contract's "sendData" function could be either successful or unsuccessful due to a certain property not being satisfied in the smart contract. This could happen for many reasons:

- The caller of the function has not been previously accepted into the network
- The consumer has not provided enough gas for the transaction to succeed

Figure 7.13 shows the user interface where the consumer can get the load balancing data from the utility instantaneously.

Get Live Updates

Index	Value
0	1996

Figure 7.13: Architecture 2A: Get Load Balancing Data Instantaneously

Listing 7.16 shows the code used to display the events received as a result of the utility sending load balancing data (explained in subsection 7.2.2) using the interface in figure 7.13.

The public-private key pair explained in section 5.8 listing 5.6 and created by the consumer in listing 7.10 (explained in subsection 5.8.2) is used. The consumer extracts his/her private key from local storage - line 7 - and converts it to the appropriate RSA key format (explained in listing 5.10) - line 8.

The subscription (explained in section 5.7) - lines 12 to 29 - subscribes the consumer to the event containing the load balancing data from the utility.

Line 13 describes the topics of this subscription which are the signature of the event (event name with the parameters) and the Ethereum address of the consumer. Only the event containing the load balancing data will be read. In addition, by adding the Ethereum address of the consumer to the topics, we can now filter the events by this address. This means that only the events that contain this consumer's Ethereum address will be read. For example, the consumer with Ethereum account "0x88B1A85759a33971f6647d77C7c691BB91c125bc" will only receive load balancing data from the utility where the event contains the consumer's Ethereum account: "0x88B1A85759a33971f6647d77C7c691BB91c125bc".

Line 19 gets the load balancing data from the event. This data is not one of the topics mentioned since it is not an indexed variable in the event (explained in subsection 2.4.8).

The consumer's private key is used to decrypt the load balancing data received from the utility through the event - line 23.

The consumer can now display any load balancing data received in real-time and place it in the table seen in figure 7.13 - lines 26 to 28.

```
async componentDidMount() {
    let accounts = await web3.eth.getAccounts();
    const t = this;
    let b = web3.utils.padLeft(accounts[0], 64);

    //get private key locally
    var privateKey = localStorage.getItem(accounts[0]);
    var k = new (require('../encryption/privateKey2'))(privateKey);

    //subscription
    //2 params: event signature, consumer address (in topics)
    web3.eth.subscribe('logs', {
        topics:
            [web3.utils.sha3("LoadBalancingSent(address,string)"),b]
    }, function(error, result){
        if (!error)
            console.log(result);

            //get the data value (not indexed)
            let d = web3.eth.abi.decodeParameters(['string'],
                result.data);
            console.log(d);

            //decrypt the data
            const dec =
                k.key.decrypt(web3.utils.hexToUtf8(d[0]),'utf8');

            //place all the values in the table
            t.setState({
                lst: (t.state.lst).concat([{to:result.topics[1],
                    value:dec}])
            });
    });
}
```

Listing 7.16: Instantaneously Getting Load Balancing Data Code

We do not show the Metamask interaction between the consumer and the event containing the load balancing data since the receipt of an event is not a transaction that changes the Ethereum state and consequently does not need to be verified. It is considered to be querying the distributed ledger.

Figure 7.14 shows the user interface that the utility can interact with. The utility can send load balancing data to a consumer in figure 7.15 and can receive electrical data from a consumer in figures 7.16 and 7.17.

Figure 7.14: Architecture 2A: Utility Interface

Figure 7.15 shows the user interface where the utility can input the load balancing data needed by a consumer and send it.

Figure 7.15: Architecture 2A: Send Load Balancing Data

Listing 7.17 shows the code used to interact with the communication smart contract (second smart contract) using the interface in figure 7.15.

The public-private key pair explained in section 5.8 listing 5.6 and created by the consumer in listing 7.10 (explained in subsection 5.8.2) is used. The utility extracts the consumer's public key and converts it to the appropriate RSA key

format (explained in listing 5.9) - line 6.

The consumer's public key is used to encrypt the load balancing data - line 12.

The asynchronous transaction - line 16 - calls the smart contract function "fixData" (discussed in subsection 7.2.2 listing 7.9) that takes two parameters, one of which is the consumer's Ethereum address and is evident in figure 7.15 and the other is the encrypted load balancing data explained in line 12. The function uses the "send" keyword which leads to a transaction that changes the Ethereum state.

```
1 onSubmit = async (event) =>{
2     event.preventDefault();
3     this.setState({loading:true, errorMessage:''});
4     await ethereum.enable();
5 
6     var k = new
        (require('../encryption/publicKey2'))(web3.utils.hexToUtf8(
        this.props.pk));
7 
8     try {
9         const accounts = await web3.eth.getAccounts();
10 
11         //encrypt and send
12         const enc = k.key.encrypt(this.state.data,'base64');
13 
14         //call the smart contract function to send load balancing
              data
15         //2 params: address, encrypted load balancing data
16         await communication.methods.fixData(this.props.address,
              web3.utils.utf8ToHex(enc)).send({from: accounts[0]});
17     } catch(err) {
18         this.setState({errorMessage: err.message});
19     }
20 };
```

Listing 7.17: Send Load Balancing Data Code

The interaction between the consumer and the second smart contract's "fixData" function could be either successful or unsuccessful due to a certain property not being satisfied in the smart contract. This could happen for many reasons:

- The caller of the function is not the utility
- The consumer has not provided enough gas for the transaction to succeed

Figure 7.16 shows the user interface where the utility can get the electrical data from the utility instantaneously.

Get Live Updates

Node Network Address	Date and Time	Value
0x88B1A85759a33971f6647d77C7c691BB91c125bc	2020517235	22

Figure 7.16: Architecture 2A: Get Electrical Data Instantaneously

Listing 7.18 shows the code used to display the events received as a result of the consumer sending electrical data (explained in subsection 7.2.2) using the interface in figure 7.16.

The public-private key pair explained in section 5.8 listing 5.6 and created by the utility (explained in subsection 5.8.1) is used. The utility extracts their private key from local storage and converts it to the appropriate RSA key format (explained in listing 5.8) - line 1.

The subscription (explained in section 5.7) - lines 8 to 25 - subscribes the utility to the event containing the electrical data from the consumers.

Line 9 describes the topic of this subscription which is the signature of the event (event name with the parameters). Only the event containing the electrical data will be read.

Line 15 gets the electrical data from the event. This data is not one of the topics mentioned since it is not an indexed variable in the event (explained in subsection 2.4.8).

The utility's private key is used to decrypt the electrical data received from the consumers through the event - line 19.

The utility can now display any electrical data received in real-time and place it in the table seen in figure 7.16 - lines 22 to 24.

```
import key from '../encryption/privateKey';

componentDidMount() {
    const t = this;

    //subscription
    //1 param: event signature (in topics)
    web3.eth.subscribe('logs', {
        topics:
            [web3.utils.sha3("DataSent(address,bytes32,string)")]
    }, function(error, result){
        if (!error)
            console.log(result);

            //get the data value (not indexed)
            let d = web3.eth.abi.decodeParameters(['address',
                'string'], result.data)
            console.log(d);

            //decrypt the data
            const dec =
                key.decrypt(web3.utils.hexToUtf8(d[1]),'utf8')

            //place all the values in the table
            t.setState({
                lst: (t.state.lst).concat([{from:d[0],
                    time:web3.utils.hexToUtf8(result.topics[1]),
                    value:dec}])
            });
    });
}
```

Listing 7.18: Instantaneously Getting Electrical Data Code

We do not show the Metamask interaction between the utility and the event containing the electrical data since the receipt of an event is not a transaction that changes the Ethereum state and consequently does not need to be verified. It is considered to be querying the distributed ledger.

Figures 7.17 and 7.18 show the user interface where the utility can get the electrical data from a consumer using a specific timestamp.

Get Data By Time

Time

YYYYMDHM

GET!

Node Network Address	Date and Time	Value

Figure 7.17: Architecture 2A: Get Electrical Data at Time Initial

Get Data By Time

Time

2020517235 YYYYMDHM

GET!

Node Network Address	Date and Time	Value
0x88B1A85759a33971f6647d77C7c691BB91c125bc	2020517235	22

Figure 7.18: Architecture 2A: Get Electrical Data at Time

Listing 7.19 shows the code used to display the events received as a result of the consumer sending electrical data (explained in subsection 7.2.2) at a certain time of the day (timestamp) using the interface in figure 7.18.

The public-private key pair explained in section 5.8 listing 5.6 and created by the utility (explained in listing 5.8.1) is used. The utility extracts their private key from local storage and converts it to the appropriate RSA key format (explained in listing 5.8) - line 1.

Lines 9 to 25 look for an event containing the electrical data from the consumers at the given timestamp.

Line 10 describes the filter used on the event which is the timestamp at which the utility needs the consumer electrical data. Only the event containing the electrical data will be read. Moreover, by adding the timestamp to the filter, we can now filter the events by this timestamp which means that only the events that contain this timestamp will be read. Line 11 describes which block the lookup should start from for this event with the specific timestamp.

The utility's private key is used to decrypt the electrical data received from the consumers through the event retrieved - line 17.

The utility can now display the electrical data received at a certain time of the day and place it in the table seen in figure 7.18 - lines 21 to 23.

```
1 import key from '../../encryption/privateKey';
2
3 onSubmit = async (event) =>{
4     event.preventDefault();
5     const t = this;
6
7     //look for event
8     //2 params: timestamp (in filter) , which block to start from
9     communication.events.DataSent({
10        filter: {
              _timestamp:[web3.utils.utf8ToHex(this.state.time)]}
11        , fromBlock: 0
12    })
13    .on('data', function(event){
14        console.log(event);
15
16        //decrypt
17        const dec =
              key.decrypt(web3.utils.hexToUtf8(event.returnValues[2]),
              'utf8')
18
19
20        //place all the values in the table
21        t.setState({
22            lst: t.state.lst.concat([{from:event.returnValues[0],
                  time:web3.utils.hexToUtf8(event.returnValues[1]),
                  value:dec}])
23        });
24    })
25    .on('error', console.error);
26 };
```

Listing 7.19: Getting Electrical Data by Timestamp Code

We do not show the Metamask interaction between the utility and the event containing the electrical data since the receipt of an event is not a transaction that changes the Ethereum state and consequently does not need to be verified. It is considered to be querying the distributed ledger.

7.4 Results

The results show the various smart contract components in the different architectures. These components include variables and functions used in the deployed smart contracts. The functions consume certain amounts of gas; therefore, the cost of using them varies. These concepts have been discussed in subsection 2.4.7. The test network used in this thesis, as mentioned in section 5.2, is the Rinkeby Test Network where the price of gas is constant and is set at 1 GWei. However, in the real network, the cost of gas varies at an average price of 20 GWei. We used 1 Ether equivalent to 186 USD. The cost is not influenced by the size of the network but by the complexity of the transaction. Thus, simulating a network with few consumers versus many consumers will not affect the results in any way.

The results are those of a smart grid communication network composed of various smart meters and one utility connected via the Ethereum infrastructure. This architecture describes the entry of a consumer into the smart grid communication network created. Entering the smart grid communication network will allow the consumer to send electrical data to the utility and the utility to send load balancing data to the consumer. The details concerning these smart contracts are found in sections 7.1 and 7.2.

For each of the contracts, there will be a table discussing: the transaction title, who the transaction is sent from, who the transaction is sent to, the amount of gas used and the fee (in ether). Furthermore, there will be tables discussing: the transaction title, the fee (in ether), the cost (in $), the frequency the transaction is sent at, and the total cost (in $) which is relative to the frequency the transaction is sent at, and the cost per transaction.

7.4.1 First Smart Contract - Joining the Network

All the transactions in Table 7.1 and Table 7.2, found in the first column (deploy simulation contract action, join action, and accept action), are made through smart contract functions. Both the consumer and the utility will be interacting through the use of the simulation smart contract (refer to section 7.2.1 for more details about the smart contract).

Table 7.1 describes the steps a consumer takes to enter into the network. Since the utility deploys the simulation contract, it carries the burden of using up a big amount of gas. The join actions, carried out by the consumers, and the accept actions, carried out by the utility, are transactions sent to the smart contract and use up varying amounts of gas depending on the complexity of the functions in the smart contract. Clearly the join action is more complex than the accept action as demonstrated by the difference in the gas consumption. This table describes the logistics of the transactions.

Table 7.1: Architecture 2A: Joining the Smart Grid Communication Network - Smart Contract #1

Transaction	From	To	Gas Used	Fee (ether)
Deploy Simulation Contract	Utility	-	512023	0.000512023
Join	Home	Contract	215753	0.000215753
Accept	Utility	Contract	46863	0.000046863

Table 7.2 complements Table 7.1 and includes the dollar amount of the cost of entering into the network for both the utility and the consumers. The deployment action will occur once at inception and is quite inexpensive. The join action will be completed once by every consumer that wishes to enter the network. The accept action will be completed once for every consumer that is allowed to enter the network.

Table 7.2: Architecture 2A: Cost of Transactions in Joining - Smart Contract #1

Transaction	Fee (ether)	Cost ($)	Frequency	Total Cost($)
Deploy Simulation Contract	0.000512023	0.09	1	0.09
Join	0.000215753	0.04	1 * # HH	0.04 * # HH
Accept	0.000046863	0.0087	1 * # HH	0.0087 * # HH

Assuming the population in Qatar is currently 2,869,458[2] and the number of buildings built are 216,740[3]. It is safe to estimate that there are around 2,167,400 households (HHs)/establishments that need electricity. Equation 7.1 shows the payment needed to be made by the utility once in the smart grid communication network. Equation 7.2 shows the payment needed to be made by the consumer once in the smart grid communication network.

$$\begin{aligned}\underline{\textbf{Utility Payment \#1}} &= \text{deploy simulation contract action} + \text{accept action} * \#\ \text{HH} \\ &= 0.09\$ + 0.0087\$ * 2,167,400 \\ &= 18,856.47\$\end{aligned} \tag{7.1}$$

$$\begin{aligned}\underline{\textbf{Consumer Payment \#1}} &= \text{join action} \\ &= 0.04\$\end{aligned} \tag{7.2}$$

[2] https://www.worldometers.info/world-population/qatar-population/

[3] https://www.gulf-times.com/story/635497/45-6-percent-jump-in-number-of-buildings-in-10-years

7.4.2 Second Smart Contract - Communicating

All the transactions in Table 7.3, Table 7.4, Table 7.5, or Table 7.6 found in the first column (deploy communication contract action, send data action, and fix data action), are made through smart contract functions. Both the consumer and the utility will be interacting through the use of the communication smart contract (refer to section 7.2.2 for more details about the smart contract).

Table 7.3 describes the steps a consumer takes to communicate in the network. Since the utility deploys the communication contract, it carries the burden of using up a big amount of gas. The send data actions, carried out by the consumers are transactions sent to the smart contract and the fix data actions, done by the utility are transactions sent to the smart contract. Clearly the deployment actions are more complex than the send data action and fix data action. This table describes the logistics of the transactions.

Table 7.3: Architecture 2A: Communication in the Smart Grid Communication Network - Smart Contract #2

Transaction	From	To	Gas Used	Fee (ether)
Deploy Communication Contract	Home	Contract	264086	0.000264086
Send Data	Home	Contract	33069	0.000033069
Send Fix	Home	Contract	32915	0.000032915

Table 7.4 complements Table 7.3 and includes the dollar amount of the cost of communicating in the network for both the utility and the consumers. The communication contract will be deployed by the utility once at inception and is quite inexpensive.

Table 7.4: Architecture 2A: Cost of Transactions in Initializing the Communication Environment - Smart Contract #2

Transaction	Fee (ether)	Cost ($)	Frequency	Total Cost ($)
Deploy Communication Contract	0.000264086	0.054	1	0.054

Table 7.5 and Table 7.6 complement Table 7.3 and include the dollar amount of the cost of communicating in the network for both the utility and the consumers. The send data action (Table 7.5) will be completed by every consumer that wishes to communicate electrical data to the utility. This action could occur at 15 minute intervals, 30 minute intervals, or 60 minute intervals a day. The total

cost per day is calculated for these different intervals and a significant difference can be seen. The fix data action (Table 7.6) will be completed by the utility that wishes to communicate load balancing data to a consumer. This action could occur anytime the utility believes there is a need to send load balancing data. It could be as frequent as 1 hour or could go up to every 6 hours.

Table 7.5: Architecture 2A: Cost of Transactions in Sending Electrical Data - Smart Contract #2

Transaction	Fee (ether)	Cost ($)	Frequency (min)	Total Cost / Day ($)
Send Data	0.000033069	0.0061	15 * # HH	0.58 * # HH
Send Data	0.000033069	0.0061	30 * # HH	0.29 * # HH
Send Data	0.000033069	0.0061	60 * # HH	0.14 * # HH

Table 7.6: Architecture 2A: Cost of Transactions in Sending Load Balancing Data - Smart Contract #2

Transaction	Fee (ether)	Cost ($)	Frequency / Day	Total Cost / Day ($)
Fix Data	0.000032915	0.0061	1 * # HH	0.0061 * # HH

We assume the same statistics mentioned above about the population, the number of buildings built, and the number of households (HHs)/establishments that need electricity are applicable. Equation 7.3 shows the payment needed to be made by the utility once in the smart grid communication network. Equation 7.4 shows the maximum payment needed to be made by the consumer once every year. Finally, equation 7.5 shows the payment needed to be made by the utility every time they send a load balancing transaction to a consumer.

$$\begin{aligned}\underline{\textbf{Utility Payment \#2}} &= \text{deploy communication contract action} \\ &= 0.054\$\end{aligned} \tag{7.3}$$

$$\begin{aligned}\underline{\textbf{Consumer Payment \#2}} &= \text{send data action} * 365 \\ &= 0.14\$ * 365 \\ &= 51.1\$\end{aligned} \tag{7.4}$$

$$\begin{aligned}\underline{\textbf{Utility Payment \#3}} &= \text{fix data action} \\ &= 0.0061\$\end{aligned} \tag{7.5}$$

7.4.3 Summary

As noticed, the prices are lower than in chapter 6's architecture. This is due to optimizations made. Some payments may have increased because of the inclusion of the utility's public key in the contract and the change in certain variable types. The architecture's limitations will be discussed in section 7.6.

The two smart contracts used in this architecture pose different sums to be paid by both the utility and the consumer at different points in time of the smart grid communication network. Equation 7.1, equation 7.2, equation 7.3, equation 7.4, and equation 7.5 show the payments that need to be made by the utility and the consumer at different points in the smart grid communication process. We wrap up these equations by summing up all the expenses that are paid by the utility and the consumer. Equation 7.6 sums up the expenses paid by the utility to set up the smart grid communication network for themselves and the consumers. Equation 7.7 sums up the expenses paid by the utility to interact with the utility in the smart grid communication network. Equation 7.8 sums up the expenses paid by each consumer to set up their interaction with the smart grid communication network. Finally, equation 7.9 sums up the expenses that have to be paid by each consumer yearly to interact with the utility in the smart grid communication network.

$$\begin{aligned}\underline{\textbf{Total Initial Utility Payment}} &= \text{Utility Payment \#1} \\ &+ \text{Utility Payment \#2} \\ &= 18,856.47\$ + 0.054\$ \\ &= 18,856.524\$\end{aligned} \tag{7.6}$$

$$\begin{aligned}\underline{\textbf{Utility Payment Per Load Balancing Data}} &= \text{Utility Payment \#3} \\ &= 0.0061\$\end{aligned} \tag{7.7}$$

$$\begin{aligned}\underline{\textbf{Total Initial Consumer Payment}} &= \text{Consumer Payment \#1} \\ &= 0.04\$\end{aligned} \tag{7.8}$$

$$\begin{aligned}\underline{\textbf{Total Consumer Payment Per Year}} &= \text{Consumer Payment \#2} \\ &= 51.1\$\end{aligned} \tag{7.9}$$

7.5 Security Properties

The security properties achieved in this architecture can be credited to the use of both the blockchain network and the smart contracts. The Ethereum blockchain ensures various security properties discussed in subsection 2.4.10 whereas the smart contracts and their details are discussed in subsection 2.4.2. Both the Ethereum blockchain and its smart contracts are discussed in terms of the security properties achieved.

7.5.1 Smart Contract Properties

The architecture's two smart contracts discussed in section 7.2 subsections 7.2.1 and 7.2.2 were structured to provide certain security properties alongside the innate blockchain security properties. The smart contracts contain certain restrictions and requirements that force functions to be constrictive and serve the purpose of securing the use of the smart grid communication network. It is important to note that the use of the term "network" is synonymous with the "smart grid communication network" and not the whole blockchain network.

First Smart Contract - Joining the Network

This contract is deployed by the utility only once. Once deployed at the inception of the smart grid communication network, the simulation smart contract becomes accessible by all other nodes on the blockchain since it is now part of the blockchain's ledger. This smart contract is deployed to allow the consumers to join the network composed of the different consumers containing legal smart meters registered at the utility. The details of this smart contract can be found in subsection 7.2.1.

The "constructor" function is used by the utility to deploy the contract. The utility places its Ethereum address on the smart contract to restrict the use of some smart contract functions. This provides **authenticity** and limits who can call the functions to **secure** the network.

The "requestEntry" function contains one requirement: the caller of the function must not have previously requested entry into the network. This requirement stops the consumer from bombarding the network with useless requests. If the consumer is a malicious user, they could use any of the smart contract functions to overload the network. This function ensures that the consumer has not previously requested entry. This provides **availability** of the blockchain services.

The "acceptNode" function contains two requirements. The first is that the caller of the function is the utility. This stops any user in the blockchain network from accepting themselves or any other malicious users onto the network. This provides **legitimacy** of the consumers on the network and **integrity** of the electrical data to be sent by this consumer. The second is that the consumer the utility is trying to accept into the network must not already be part of the network. This requirement stops the utility from accepting a consumer multiple times and wasting resources. This provides **availability** of the blockchain services. Once the consumer is added to the network, we can know that the consumer is a legitimate user. This provides **reliability** of the network.

The "removeNode" function contains two requirements. The first is that the caller of the function is the utility. This stops any user in the blockchain network from accepting themselves or any other malicious users onto the network. This provides **legitimacy** of the consumers on the network and **integrity** of the electrical data to be sent by this consumer. The second is that the consumer the utility is trying to remove from the network must already be part of the network. This requirement stops the utility from removing a consumer multiple times and wasting resources or even attempting to remove a consumer that does not exist on the network to begin with. This provides **availability** of the blockchain services.

All the measures above provide **accountability** of the users and **transparency** and **auditability** of the data.

Second Smart Contract - Communicating

This contract is deployed by the utility only once. Once deployed at the inception of the smart grid communication network, the communication smart contract becomes accessible by all other nodes on the blockchain since it is now part of the blockchain's ledger. This smart contract is deployed to allow the consumers to communicate their energy data securely and the utility to send load balancing data back to the consumer (demand-response). This preserves the two-way data flow for which the smart grid is known. The details of this smart contract can be found in subsection 7.2.2.

The "constructor" function is used by the utility to deploy the contract. The first smart contract's Ethereum address is passed to this function to be able to access it in order to check for the existence or non-existence of a consumer in the network. The reference is used to restrict the use of the smart contract function used to send electrical data. This provides **authenticity** and limits who can call specific functions to **secure** the network. It also provides **integrity** of the electrical data to be sent by this consumer.

The "sendData" function contains one requirement: the caller of the function is already be part of the network. This stops any malicious user in the blockchain network from sending incorrect electrical data on behalf of the consumer. This requirement provides **integrity** of the electrical data sent by this consumer.

The "fixData" function contains one requirement: the caller of the function is the utility. This stops any malicious user in the blockchain network from sending incorrect load balancing data on behalf of the utility. This requirement provides **integrity** of the load balancing data sent by the utility for this specific consumer.

All the measures above provide **accountability** of the users and **transparency** and **auditability** of the data.

7.5.2 Blockchain Properties

Various security properties characterize the blockchain platform. These security properties described including confidentiality and privacy, integrity, availability, authenticity, transparency, auditability, accountability, anonymity, reliability, and termination provided by blockchain are integral to our thesis work. All the security properties are ensured through the structure of the blockchain and the use of smart contracts in Ethereum to provide secure communication between the smart meters and the utility.

Confidentiality and Privacy

By remaining pseudo-anonymous, the consumer can send electrical data openly without worrying about the data being traced back to them personally. The data can only be traced back to the Ethereum address which, as mentioned, is not linked to the consumer's personal information. Even if the link between the consumer's personal information and the consumer's account, the consumer's data will still remain confidential due to the encryption of both the electrical data and the load balancing data.

The electrical data sent from the consumer through a transaction is encrypted using the utility's public key and sent using the consumer's Ethereum address. The load balancing data sent from the utility through a transaction is encrypted using the consumer's public key and sent using the utility's Ethereum address.

Integrity

The electrical data sent from the consumer through a transaction will be placed on the immutable ledger. The load balancing data sent from the utility through a transaction will also be placed on the immutable ledger.

Availability

The electrical data sent from the consumer through a transaction and the load balancing data sent from the utility through a transaction are placed on the distributed ledger to be accessed at any time.

Authenticity

Smart contract functions were designed in a way to only allow specific users to call them as can be seen in the description of the smart contracts of this architecture in section 7.2. Malicious users cannot access certain functions to change data stored in the smart contract.

The electrical data sent from the consumer through a transaction is signed by the consumer. The load balancing data sent from the utility through a transaction is signed by the utility.

Transparency

The electrical data sent from the consumer through a transaction and the load balancing data sent from the utility through a transaction are placed on the distributed ledger to be accessed at any time by anyone in the blockchain network.

Auditabiliy

The electrical data sent from the consumer through a transaction will be placed on this immutable ledger. The load balancing data sent from the utility through a transaction will also be placed on this immutable ledger.

Accountability

The electrical data sent from the consumer through a transaction contains the consumer's Ethereum address. The load balancing data sent from the utility through a transaction contains the utility's Ethereum address.

Anonymity

The electrical data sent from the consumer through a transaction is sent using the consumer's Ethereum address. The load balancing data sent from the utility through a transaction sent using the utility's Ethereum address. Both of these addresses do not link back to any personal data of the user of the account and thus guarantee pseudo-anonymity.

Termination

Utilizing gas and gas limits (explained in subsection 2.4.7) forces the function to terminate whether complete or incomplete.

Reliability

The distributed ledger provides all the security properties mentioned in this section. The smart contract provides restrictions for the functions. Both the smart contract and ledger provide the reliability needed. The ledger makes sure the data has integrity and is available, authentic, auditable, transparent, and private. The ledger makes sure the user is anonymous and accountable. The smart contract makes sure the transactions called are terminable. The smart contract is also the entity that contains all the rules that the users in the network must follow to enter the network and communicate data.

7.6 Limitations

Various security properties are achieved in this architecture, which are mentioned in section 7.5 and make the Ethereum blockchain a good solution for securing the two-way communication between the consumers and the utility. However, a limitation arises which is scalability. Before discussing the issue of scalability, we will go over the limitations mentioned in chapter 6 section 6.6 to discuss how this architecture overcame them.

7.6.1 Privacy using Pseudo-anonymity

In the blockchain network, a user that has a blockchain account cannot be considered fully anonymous. Pseudo-anonymity is when a user is linked back to their blockchain address but not to any of their personal information. All transactions are stored publicly on the blockchain ledger and are visible to anyone to analyze and interpret. Anonymity of the sender depends on the pseudonym not being linked to his/her true identity. Thus, a user can preserve his/her privacy as long as pseudonym is not linked back to the individual. If a link is made, the identity of the user is revealed and the pseudo-anonymity is broken. Even if this link is made, the consumer's electrical data will not be compromised. The data has been encrypted. This data is still transparent but it will no longer be understandable unless the private key of the consumer is available to decrypt the load balancing data or the private key of the utility is available to decrypt the electrical data.

7.6.2 Cost of Smart Contracts

The deployment and interaction with the three smart contracts discussed in chapter 6 section 6.2 is costly, and the two smart contracts discussed in this chapter have reduced this cost as seen in section 7.4. This reduction in cost is due to the optimizations made in the smart contract structure such as carefully picking data types, introducing data in the contract bytecode, packing variables, and other optimizations.

7.6.3 Scalability

Both the utility and the consumers in the smart grid communication network will have to send an abundance of messages. The consumers will each have to send data at intervals of 15 minutes, 30 minutes, or an hour. The utility will have to send load balancing data to consumers when required. All these transactions are not scalable on the Ethereum blockchain network. As described in table 2.1 section 2.3, the number of transactions per second in the Ethereum network is 15. This scalability issue will have to be amended in order to allow our solution to work properly and scalably.

Chapter 8

Architecture 2 (Part B)

The second part of Architecture 2, discussed in this chapter, explores an alternative method for sending encrypted data using a combination of the Ethereum platform, cryptographic tools, and the cloud. We propose to create two separate smart contracts each for a different functionality. These smart contracts resemble the ones found in chapter 8 with minor differences to accommodate a change in handling electrical and load balancing data. This chapter shows the scenarios the architecture will follow in section 8.1, the smart contract details in section 8.2 and the cloud database details in section 8.3, the user interface used to interact with the smart contracts in section 8.4, the results in section 8.5, the security properties achieved in section 8.6, and the limitations of this architecture in section 8.7.

8.1 Scenario

Using a control flow graph (CFG), we display the order of execution of the calls and updates to the smart contract. It shows all possible steps that can be executed in our program.

8.1.1 Initial Contract Deployment

Figure 7.1 from chapter 7 shows the steps taken by the utility to deploy the two smart contracts needed to join the smart grid communication network and communicate in the smart grid communication network. The utility initially deploys the simulation contract which would lead to a transaction that would update the blockchain state (step 1). This contract deals with consumers requesting entry and utility accepting consumers into the network.

If the simulation contract is deployed successfully (step 2), the utility can deploy the communication contract (step 3) which would lead to a transaction that would update the blockchain state (step 4). This contract deals with the consumer and utility data interaction. If the simulation contract is not deployed successfully, the utility will try to deploy the simulation contract again.

Deploying the simulation contract and the communication contract leads to changes in the Ethereum state.

8.1.2 Contract Interaction

Figure 6.2 from chapter 6 shows the steps taken by the consumer to try and enter the network. The consumer can request entry which would lead to a transaction that would update the blockchain state (step 1).

Figure 6.2 from chapter 6 also shows the steps taken by the utility to accept or reject the consumer. The utility queries for requests from consumers. The utility finds a request (step 2). If there is a request and the utility wishes to reject the consumer, then it would lead back to a state of waiting for a request. For instance, the utility can choose to reject a consumer if they did not register with the utility beforehand. If there is a request and the utility wishes to accept the consumer, then it would lead to a transaction that would update the blockchain state (step 3). If the utility does not find a request, it waits 15 minutes before rechecking (step 4). The 15 minute wait can be updated to 30 minutes or an hour depending what the utility wants. The 15 minute intervals will recur throughout the architecture and can be updated to 30 minutes or an hour also depending on what the utility wants. The utility can also remove a consumer from the network that would lead to a transaction that would update the blockchain state (step 5).

Requesting, accepting and removing consumers leads to changes in the Ethereum state. Querying for requests does not lead to changes in the Ethereum state.

Figure 8.1 shows the steps taken by the consumer to send data. Electrical data can be sent from the consumer to the utility through an event and can be received by the utility. The consumer first encrypts the electrical data and sends it to the cloud (step 1), it then hashes the electrical data (step 2) and attempts to send an event containing the hashed electrical data. The consumer checks if it is allowed to send data (step 3). If the consumer is not allowed to send data, then it would lead back to a state of waiting to send electrical data. If the consumer is allowed to send data, then it would lead to a transaction that would update the blockchain state (step 4). The consumer attempts to send data every hour (step 5). The 15 minute intervals or 1 hours intervals will recur throughout the architecture and can be updated depending on what the utility wants.

Figure 8.1 also shows the steps taken by the utility to send data. Load balancing data can be sent from the utility to the consumer through an event and can be received by the consumer. The utility first encrypts the load balancing data and sends it to the cloud (step 6). It then hashes the load balancing data (step 7) and sends an event containing the hashed load balancing data that would lead to a transaction that would update the blockchain state (step 8).

Sending data, electrical data by the consumer and load balancing data by the

utility, leads to changes in the Ethereum state.

Every 15 minutes the consumer can get an event containing the hashed load balancing data (step 12) if the utility has sent load balancing data. The consumer queries the cloud for the load balancing data and decrypts it (step 9). The consumer then hashes the decrypted load balancing data (step 10) and compares the hashed load balancing data queried from the cloud with the hashed load balancing data received from the event (step 11). If the hashes match, then the data has not been tampered with, otherwise it has been. The use of cryptographic techniques here is to ensure the data remains confidential, unchanged, and authentic. These properties will be explained further in section 8.6.

Every hour the utility can get an event containing the hashed electrical data (step 16) if the utility has sent electrical data. The utility queries the cloud for the electrical data and decrypts it (step 13). The utility then hashes the decrypted electrical data (step 14) and compares the hashed electrical data queried from the cloud with the hashed electrical data received from the event (step 15). If the hashes match, then the data has not been tampered with, otherwise it has been. The use of cryptographic techniques here is to ensure the data remains confidential, unchanged, and authentic. These properties will be explained further in section 8.6.

The consumer receiving load balancing data and the utility receiving electrical data does not lead to changes in the Ethereum state.

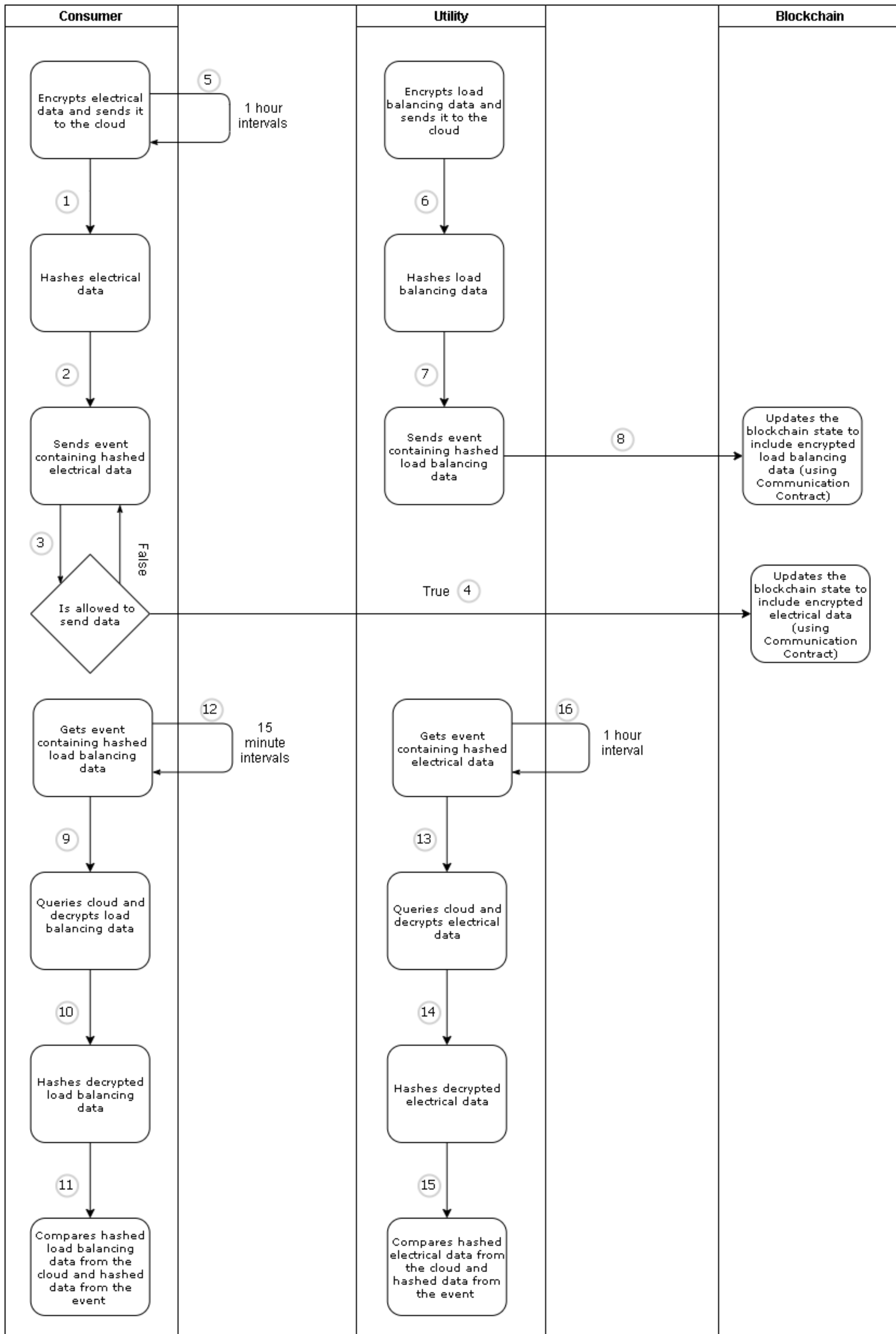

Figure 8.1: Architecture 2B: Contract Interaction (Part 2)

8.2 Smart Contract Details

Now that we have gone through the control flow of the smart contracts' executions, we answer the following questions for each smart contract: By who was the contract deployed? When was it deployed? For what purpose? What are the contracts' contents?

This chapter explores an alternative method for sending encrypted data using a combination of the Ethereum platform, cryptographic tools, and the cloud. The two smart contracts described in section 7.2 are the smart contracts used in this chapter. As we've explained in subsection 2.4.2, variables are set up to be interacted with and functions exist to allow actions to occur on the blockchain.

8.2.1 First Smart Contract - Joining the Network

The specifics of the first smart contract can be found in subsection 7.2.1 from chapter 7. The detailed code of the first smart contract variables and functions is available in Appendix A listing A.6.

8.2.2 Second Smart Contract - Communicating

This contract is deployed by the utility only once at inception. Figure 8.2 shows a high level view of the smart grid communication network in relation to the first smart contract. This figure illustrates a city in Qatar that contains 10 households/establishments (HHs) numbered 1 through 10 and the utility numbered 11. All of the establishments numbered 1 through 10 contain smart meters that measure their electrical consumption.

There are three distinguishable components in figure 8.2. The first component is the blockchain network. This network contains all the households and the utility. The HHs and the utility are part of the peer-to-peer Ethereum blockchain network as illustrated by the road which connects all the HHs and establishments. The HHs are nodes in the blockchain network and have access to the distributed blockchain ledger. Node 11, the utility, contains the first smart contract known as the simulation smart contract. Once deployed at the inception of the smart grid communication network, the smart contract becomes accessible by all other nodes on the blockchain since it is now part of the blockchain's ledger. This smart contract is deployed to allow the consumers to communicate their energy data securely and the utility to send load balancing data back to the consumer (demand-response). This preserves the two-way data flow for which the smart grid is known.

The second component is the distributed ledger which is the Ethereum ledger that every node will have access to. This ledger will contain all the transactions that have been mined and added to the blockchain. The transactions include all the updates to the Ethereum state (explained in subsection 2.4.5).

The third component is the cloud database which is the database that every node will access to store their electrical data and to access their load balancing data. The data will be encrypted and stored on the cloud database. The consumers and utility can now access the data and decrypt it for use.

The symbol table explains that the different nodes 1 through 11 can send a message with the data "M" at a certain time "t". It shows the encrypt "E" and decrypt "D" symbols that are used to encrypt and decrypt the data sent to and received from the cloud respectively. It also shows the hash function "H" which is used on the message to produce a digest "D" that is added to the distributed ledger. The legend displays the objects that exist in the smart grid communication network.

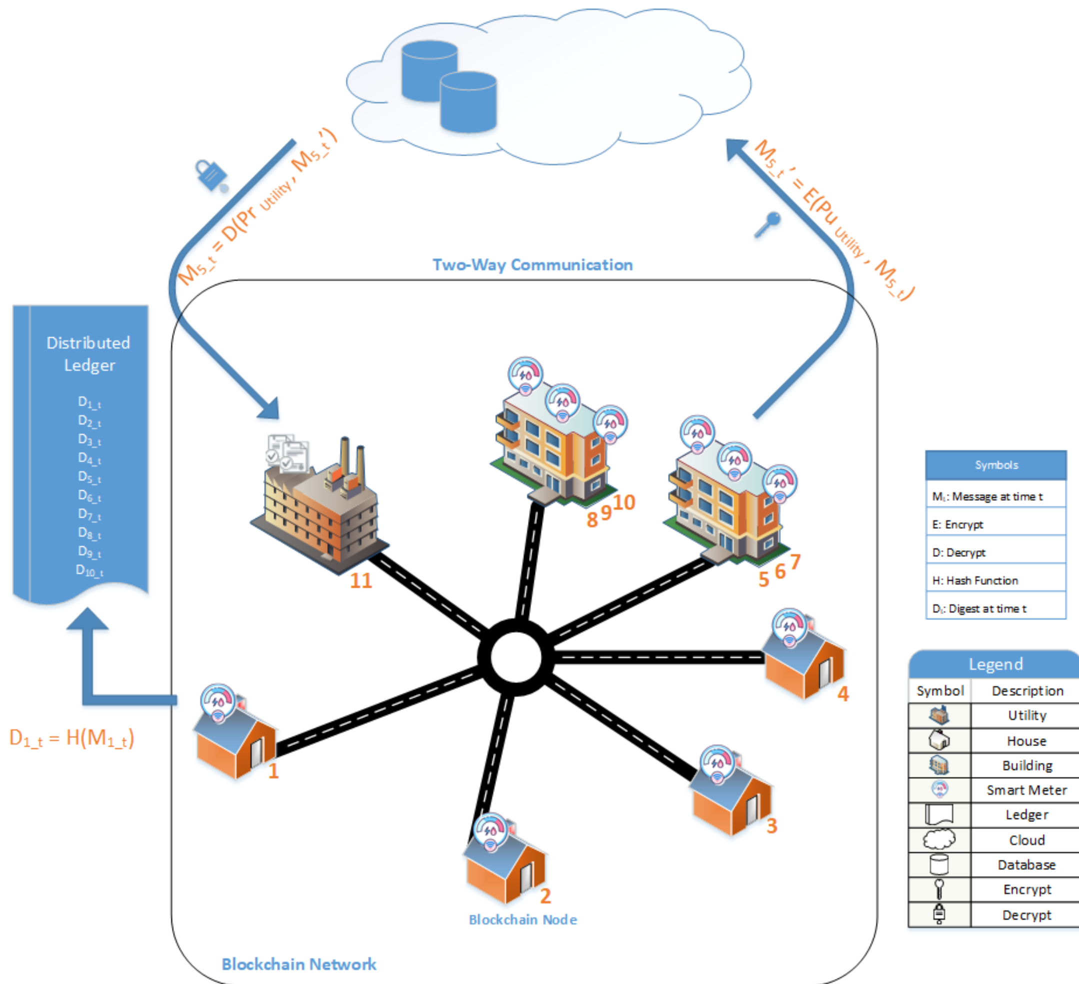

Figure 8.2: Architecture 2B: Smart Contract #2 Distribution

As we've explained in subsection 2.4.2, variables are set up to be interacted with and functions exist to allow actions to occur on the blockchain. The detailed code of the second smart contract variables and functions is available in Appendix A listing A.7. Here, we will briefly explain the variables, functions, and their purpose in the smart contract.

Listing 8.1 displays the variable names and descriptions in this contract. These variables are used in functions we discuss below.

- **Line 1 - S**: Simulation contract instance declaration
- **Line 4 - DataSent**: Event declaration for electrical data to be sent by the consumer that returns the consumer's Ethereum address, the timestamp of the data, and the value of the data (the concept of smart contract events is discussed extensively in subsection 2.4.8)

- **Line 5 - LoadBalancingData**: Event declaration for load balancing data to be sent by the utility that returns the consumer's Ethereum address (the consumer to which the data should be sent) and the value of the data

Both of these events will be discussed further below. It can be noticed that the change from architecture 2 (Part A) in subsection 7.3.2 listing 7.6 to listing 8.1 is in the event's parameters. The type of the parameters changed from the "string" type to the "bytes32" type - lines 4 and 5. This change will lead to a difference in smart contract cost as will be seen in section 8.5.

```
1 Simulation private S;
2
3 //events
4 event DataSent(address _from, bytes32 indexed _timestamp, bytes32
    _value);
5 event LoadBalancingSent(address indexed _to, bytes32 _value);
```

Listing 8.1: Variables

The functions included in this smart contract are listed below and will be explained in listings 8.2, 8.3, and 8.4 respectively.

- Constructor
- Send data
- Fix data

We assume that the consumers using this smart contract have been accepted onto the network and are accessing this contract to send electrical data to the utility and to get load balancing data from the utility. The "constructor" function - line 3 - initializes the variables above as seen in listing 8.2. The address that is passed as a parameter is the simulation smart contract address - line 3. This address is produced when the utility deploys its simulation smart contract. It is passed to the constructor so that the communication smart contract can make use of the components in the simulation smart contract. Using this smart contract address, the simulation contract may be referenced - line 4.

```
1 //constructor
2 //takes the simulation contract to refer to it later
3 function Communication(address a) public {
4     S = Simulation(a);
5 }
```

Listing 8.2: Constructor

Via this smart contract, the consumer can send electrical data to the utility. Sending data using the "sendData" function - line 3 - in listing 8.3 will emit an

event containing the electrical data in addition to the date and time (following a certain format which is year-moth-day-hour-minute). Before doing so, the caller of this function must be proven to be a member of the smart grid communication network - line 4. If this condition is not satisfied, the function will throw an exception which we will further discuss in subsection 8.4.2. If it is satisfied, the consumer's Ethereum address, the timestamp of the data, and the value of the data are sent as an event - line 5. The concept of smart contract events is discussed extensively in subsection 2.4.8. By checking the simulation contract's mapping of accepted consumers, the smart contract can ensure that the caller of this function is part of the communication network before allowing the transaction to be processed.

The utility can then watch for the event at specific intervals. It will watch for events related to electrical data sent by the consumers. These events are filtered based on what timestamp is needed (to be discussed in subsection 8.4.2).

It can be noticed that the change from architecture 2 (Part A) in subsection 7.3.2 listing 7.8 to listing 8.3 is in the event's parameters. The type of the parameters changed from the "string" type to the "bytes32" type - line 3. This change will lead to a difference in smart contract cost as will be seen in section 8.5.

```
1 //send the Data
2 //timestamp should follow certain convention yyyymdhm
3 function sendData(bytes32 timestamp, bytes32 value) public {
4     require(S.isAccepted(msg.sender));
5     DataSent(msg.sender,timestamp,value);
6 }
```

Listing 8.3: Send data

The utility can also send load balancing data to the consumer. Sending data using the "fixData" function - line 4 - in listing 8.4 will emit an event containing the load balancing data. Before doing so, the caller of this function must be proven to be the utility since load balancing data can only be sent by the utility (whose address is derived from the simulation contract) - line 5. If this condition is not satisfied, the function will throw an exception which we will further discuss in subsection 8.4.2. If it is satisfied, the consumer's Ethereum address and the value of the data are sent as an event - line 6. The concept of smart contract events is discussed extensively in subsection 2.4.8.

The consumer can then watch for the event at specific intervals. They will watch for events related to load balancing data sent by the utility. These events are filtered based on who their Ethereum address (to be discussed in subsection 8.4.2).

It can be noticed that the change from architecture 2 (Part A) in subsection 7.3.2 listing 7.9 to listing 8.4 is in the event's parameters. The type of the parameters changed from the "string" type to the "bytes32" type - line 4. This change will lead to a difference in smart contract cost as will be seen in section 8.5.

```
1 //utility
2 //send load balancing data
3 //will be enumerator 0->decrease 1->increase ....
4 function fixData(address to, bytes32 value) public {
5     require(msg.sender == S.utility());
6     LoadBalancingSent(to, value);
7 }
```

Listing 8.4: Fix data

8.2.3 Summary

A smart grid communication network will start out empty with only the utility present. The utility initially deploys two smart contracts. The first smart contract (simulation contract) will allow consumers to enter the smart grid communication network, and the second smart contract (communication contract) will allow the consumers and the utility to communicate data.

The consumers that wish to join the network and become part of the smart grid can do so by requesting to join. The utility can then accept or reject this consumer. Once accepted, the consumer may now proceed to send and receive data. If rejected, the consumer cannot send and receive data.

The consumer can now use the second smart contract (communication contract) to send data periodically to the utility. The utility can also send load balancing data to the consumer through the respective communication contract.

8.3 Cloud Details

This chapter explores an alternative method for sending encrypted data using a combination of the Ethereum platform, cryptographic tools, and the cloud. Now that we have gone through the smart contract details, we discuss next, the following questions: Which database will be used? Why is the database used? Who will have access to the database? What will be sent to the database and by what entities? What is the procedure followed in the smart grid communication network?

The cloud platform we use for communicating grid data, in this work, is Microsoft Azure. A database is created to store two tables each for a different purpose. A table is created for the electrical data sent by the consumers that the utility will have access to, and another is created for the load balancing data sent by the utility that the consumers will have access to. The table that stores the load balancing data is called "LoadBalancingData", matching the Ethereum event used for sending the hash of the load balancing data. The table that stores the electrical data is called "DataSent" matching the Ethereum event used for sending the hash of the electrical data.

The utility will need to connect to the database once a day, week, or month. It will then have to check for any electrical data that should be sent at a specific interval of 15 minutes, 30 minutes, or an hour to the database. Finally, the utility could update the database by sending load balancing data. Thus, the three operations are connecting, querying, and updating the database.

The same logic applies on the consumer side. The consumer will need to connect to the database once a day, week, or month. It will then have to update the database by sending electrical data. Finally, the consumer could check for any load balancing data at 15 minute intervals. Thus, the three operations are connecting, updating, and querying.

The smart grid communication network will need the cloud database to store electrical and load balancing data to improve scalability. To access the cloud database, certain steps must be taken by the entities which we refer to as operations.

8.3.1 Connecting to the Database

The first operation - the update operation - is done by the consumer and the utility to connect to the database. Listing 8.5 shows the code for connecting to the cloud database designated to store both electrical data and load balancing data. To test how long it would take to update the database table we record the time before and after the connection - lines 10 and 16. The connection is made to the database - line 13.

```
1  try {
2    //connect
3    Class.forName("com.mysql.cj.jdbc.Driver");
4  
5    String connectionUrl =
        "jdbc:sqlserver://s-1.database.windows.net:1433;" +
        "database=Data;"
6         + "user=rxa05@s-1;" + "password=Raphaelle2000;" +
            "encrypt=true;" + "trustServerCertificate=false;"
7         + "hostNameInCertificate=*.database.windows.net;" +
            "loginTimeout=30;";
8  
9      //record
10   long startTime = System.nanoTime();
11 
12   //connect
13   Connection con = DriverManager.getConnection(connectionUrl);
14 
15     //stop recording
16   long endTime = System.nanoTime();
17   long timeElapsed = endTime - startTime;
18 
19   values = timeElapsed / 1000000;
20 
21   con.close();
22 } catch (Exception e) {
23   System.out.println(e);
24 }
```

Listing 8.5: Connect to Database in Cloud Code

8.3.2 Updating the Tables

The second operation, the update operation, is done by the consumer to send electrical data and by the utility to send load balancing data. Listing 8.6 shows the consumer's code for updating the data onto the cloud database's table "DataSent" designated for the electrical data. To test how long it would take to update the database table, we record the time before and after the update of the data - lines 16 and 22. A string is created that contains the smart meter id, electrical data, and timestamp needed - line 11. The data is updated in the table - line 19.

```
try {
  //send
  Statement stmt = con.createStatement();

  //variables for insert statement
  int data = (int)(Math.random()*1000);
  int hour = (int)(Math.random()*22+1);
  int minute = (int)(Math.random()*59);
  int second = (int)(Math.random()*59);

  String sql = "INSERT INTO Data.dbo.[DataSent] VALUES
      ("+counter+","+data+",'2020-02-27
      "+hour+":"+minute+":"+second+"');";
  System.out.println(sql);

    try {
        //record
        long startTime = System.nanoTime();

        //update
        stmt.executeUpdate(sql);

        //stop recording
        long endTime = System.nanoTime();
        long timeElapsed = endTime - startTime;

        values[1] = timeElapsed / 1000000;
    } catch (Exception e) {
    System.out.println(e);
  }
} catch (Exception e) {
  System.out.println(e);
}
```

Listing 8.6: Send Electrical Data in Cloud Code

8.3.3 Querying from the Tables

The third operation, the query operation, is carried out by the consumer to retrieve load balancing data and by the utility to retrieve electrical data. Listing 8.7 shows the utility's code for querying the data from the cloud database's table "DataSent" designated for the electrical data. To test how long it would take to query the database table, we record the time before and after the receipt of the data - lines 6 and 12. The data is queried from the table - line 9.

```
1 try {
2     //receive
3     Statement stmt = con.createStatement();
4 
5     //record
6     long startTime = System.nanoTime();
7 
8     //query at timestamp
9     ResultSet rs = stmt.executeQuery("select * from
          Data.dbo.DataSent where timestamp='"+t+"'");
10 
11     //stop recording
12     long endTime = System.nanoTime();
13     long timeElapsed = endTime - startTime;
14 
15     values[2] = timeElapsed / 1000000;
16 
17 } catch (Exception e) {
18     System.out.println(e);
19 }
```

Listing 8.7: Query Electrical Data from Cloud Code

8.4 User Interface

To complete the preventative proposed solution, we set up a user interface for user data interaction. The user may interact with Ethereum blockchain using the Rinkeby Test Network discussed above. Sections 8.1 and 8.2 discussed the general architecture execution and details and this section will describe the user interface put in place to interact with the Ethereum network smart contracts. Some front-end components provide the consumer access to the different smart contracts deployed by the utility or consumers. Other front-end components provide the utility access to the different smart contracts deployed by the utility or consumers. The user interface does not show the deployment of the simulation and communication smart contracts (found in subsection 5.5) since it should be completed by the utility before any interaction with the smart contracts can occur.

8.4.1 First Smart Contract - Joining the Network

The specifics of the first smart contract can be found in subsection 7.2.1. The detailed code of the first smart contract variables and functions is available in Appendix A listing A.6. The specifics of the user interface for the first smart contract can be found in subsection 7.3.1.

8.4.2 Second Smart Contract - Communicating

Figure 8.3 shows the user interface that the consumer can interact with. The consumer can send electrical data to the utility in figure 8.4 and can receive load balancing data from the utility.

Figure 8.3: Architecture 2B: Consumer Interface

Figure 8.4 shows the user interface where a consumer can input the electrical data needed by the utility and send it.

Figure 8.4: Architecture 2B: Send Electrical Data

Listing 8.8 shows the code used to interact with the communication smart contract (second smart contract) using the interface in figure 8.4.

Line 8 shows the hash of the provided parameters which are four sets of the electrical data and their timestamps. This hashes all the data together. The utility will have to hash the electrical data and the timestamps in the same order to get the same hash value.

The first timestamp is selected out of the four timestamps since it will be the beginning sequence from which the utility will start collecting the electrical data and timestamps to hash. This will provide the correct ordering needed by the utility to get the correct hash.

The asynchronous transaction - line 13 - calls the smart contract function "sendData" (discussed in subsection 8.2.2 listing 8.3) that takes two parameters one of which is the first timestamp as explained and the other is the the hash explained in line 8 that is evident in figure 8.4. The function uses the "send" keyword which leads to a transaction that changes the Ethereum state.

```
1 onSubmit = async (event) => {
2     event.preventDefault();
3     this.setState({loading:true, errorMessage:''});
4     await ethereum.enable();
5 
6     try {
7         //calculate the hash using multiple values
8         var hash =
             web3.utils.soliditySha3(this.state.data,this.state.time,
             this.state.data1,this.state.time1,this.state.data2,
             this.state.time2,this.state.data3,this.state.time3);
9 
10        //call the smart contract function to send electrical data
11        //2 params: timestamp, hashed electrical data
12        //time1 is placed as the timestamp, because it was the
             first time recorded for this sequence
13        await
             communication.methods.sendData(web3.utils.utf8ToHex(time)
             , hash).send({from: accounts[0]});
14    } catch(err) {
15        this.setState({errorMessage: err.message});
16    }
17 }
```

Listing 8.8: Send Electrical Data Code

The specifics of the user interface for receiving load balancing data from the utility can be found in subsection 7.3.2.

Figure 8.5 shows the user interface that the utility can interact with. The utility can send load balancing data to a consumer and can receive electrical data from a consumer in seen figure 8.6.

Smart Grid Simulator | Join | Consumer | Utility

Send Fix Data

Data

E-KWH

SEND!

Get Data By Time

Time

YYYYMDHM

GET!

Node Network Address	Date and Time	Value

Get Live Updates

Node Network Address	Date and Time	Value

Figure 8.5: Architecture 2B: Utility Interface

The specifics of the user interface for sending load balancing data to a consumer can be found in subsection 7.3.2.

Figure 8.6 shows the user interface where the utility can get the electrical data from the consumer instantaneously.

Get Live Updates

Node Network Address	Date and Time	Value
0x88B1A85759a33971f6647d77C7c691BB91c125bc	2020631536	0x1e3b4dcd3def5962f34204b851793d7492373fc00cbb74784932852196d02753

Figure 8.6: Architecture 2B: Get Electrical Data Instantaneously

Listing 8.9 shows the code used to display the events received as a result of the consumer sending electrical data (explained in subsection 8.2.2) using the interface in figure 8.6.

The subscription (explained in section 5.7) - lines 6 to 20 - subscribes the utility to the event containing the electrical data from the consumers.

Line 7 describes the topic of this subscription which is the signature of the event (event name with the parameters). Only the event containing the electrical data will be read.

Line 13 gets the electrical data from the event. This data is not one of the topics mentioned since it is not an indexed variable in the event (explained in subsection 2.4.8).

The utility can now display any electrical data received in real-time and place it in the table seen in figure 8.6 - lines 17 to 19.

```
1 componentDidMount() {
2     const t = this;
3 
4     //subscription
5     //1 param: event signature (in topics)
6     web3.eth.subscribe('logs', {
7         topics:
              [web3.utils.sha3("DataSent(address,bytes32,bytes32)")]
8     }, function(error, result){
9         t.setState({lst:[]});
10        if (!error)
11            console.log(result);
12            //get the data value (not indexed)
13            let d = web3.eth.abi.decodeParameters(['address',
                  'bytes32'], result.data)
14            console.log(d);
15 
16            //place all the values in the table
17            t.setState({
18                lst: (t.state.lst).concat([{from:d[0],
                      time:web3.utils.hexToUtf8(result.topics[1]),
                      value:d[1]}])
19            });
20    })
21 }
```

Listing 8.9: Instantaneously Getting Electrical Data Code

We do not show the Metamask interaction between the utility and the event containing the electrical data since the receipt of an event is not a transaction that changes the Ethereum state and consequently does not need to be verified. It is considered to be querying the distributed ledger.

8.5 Results

The results show the various smart contract components in the different architectures. These components include variables and functions used in the deployed smart contracts. The functions consume certain amounts of gas and thus the cost of using them varies. These concepts have been discussed in subsection 2.4.7. The test network used in this thesis, as mentioned in section 5.2, is the Rinkeby Test Network where the price of gas is constant and is set at 1 GWei. However, in the real network, the cost of gas varies at an average price of 20 GWei. We used 1 Ether equivalent to 186 USD. The cost is not influenced by the size of the network but by the complexity of the transaction. Thus, simulating a network with few consumers versus many consumers will not affect the results in any way.

The results are those of a smart grid communication network composed of various smart meters and one utility connected via the Ethereum infrastructure. This architecture describes the entry of a consumer into the smart grid communication network created. Entering the smart grid communication network will allow the consumer to send electrical data to the utility and will allow the utility to send load balancing data to the consumer. The details concerning these smart contracts are found in sections 8.1 and 8.2.

For each of the contracts, there will be a table discussing: the transaction title, who the transaction is sent from, who the transaction is sent to, the amount of gas used and the fee (in ether). In addition, there will be tables discussing: the transaction title, the fee (in ether), the cost (in $), the frequency the transaction is sent at, and the total cost (in $) which is relative to the frequency the transaction is sent at and the cost per transaction.

The results also show the tests performed on the Azure Cloud platform to test the feasibility of the communication at the different stages mentioned in section 8.3. The smart grid communication network will need the cloud database to store electrical and load balancing data. To access the cloud database, certain operations must be handled by the entities specifically connecting to the database in the cloud, updating the tables in the database, and querying the tables in the database.

Testing the cloud database performance in relation to these three different operations is done at 5 minute intervals over the span of 10 runs. It was done a minimum of 4 times for each of the operations. Each of these operations is done using a certain number of synchronous users between 1 and 300. Graphs will display averaged data at the different intervals for each number of users and other graphs will display averaged data compared for the different user quantities.

8.5.1 First Smart Contract - Joining the Network

All the transactions in Table 8.1 and Table 8.2, found in the first column (deploy simulation contract action, join action, and accept action), are made through smart contract functions. Both the consumer and the utility will be interacting through the use of the simulation smart contract (refer to section 8.2.1 for more details about the smart contract).

Table 8.1 describes the steps a consumer takes to enter into the network. Since the utility deploys the simulation contract, it carries the burden of using up a big amount of gas. The join actions, carried out by the consumers, and the accept actions, carried out by the utility, are transactions sent to the smart contract and use up varying amounts of gas depending on the complexity of the functions in the smart contract. Clearly the join action is more complex than the accept action as seen by the difference in the gas consumption. The following table describes the logistics of the transactions.

Table 8.1: Architecture 2B: Joining the Smart Grid Communication Network - Smart Contract #1

Transaction	From	To	Gas Used	Fee (ether)
Deploy Simulation Contract	Utility	-	512023	0.000512023
Join	Home	Contract	195537	0.000195537
Accept	Utility	Contract	46863	0.000046863

Table 8.2 complements Table 8.1 and includes the dollar amount of the cost of entering into the network for both the utility and the consumers. The deployment action will occur once at inception and is quite inexpensive. The join action will be completed once by every consumer that wishes to enter the network. Finally, the accept action will be completed once for every consumer that is allowed to enter the network.

Table 8.2: Architecture 2B: Cost of Transactions in Joining - Smart Contract #1

Transaction	Fee (ether)	Cost ($)	Frequency	Total Cost($)
Deploy Simulation Contract	0.000512023	0.09	1	0.09
Join	0.000195537	0.036	1 * # HH	0.036 * # HH
Accept	0.000046863	0.0087	1 * # HH	0.0087 * # HH

Assuming the population in Qatar is currently 2,869,458[1] and the number of buildings built are 216,740[2]. It is safe to estimate that there are around 2,167,400 households (HHs)/establishments that need electricity. Equation 8.1 shows the payment needed to be made by the utility once in the smart grid communication network and equation 8.2 shows the payment needed to be made by the consumer once in the smart grid communication network.

$$\begin{aligned}\underline{\textbf{Utility Payment \#1}} &= \text{deploy simulation contract action} + \text{accept action} * \#\ \text{HH} \\ &= 0.09\$ + 0.0087\$ * 2,167,400 \\ &= 18,856.47\$ \end{aligned} \tag{8.1}$$

$$\begin{aligned}\underline{\textbf{Consumer Payment \#1}} &= \text{join action} \\ &= 0.036\$ \end{aligned} \tag{8.2}$$

[1] https://www.worldometers.info/world-population/qatar-population/

[2] https://www.gulf-times.com/story/635497/45-6-percent-jump-in-number-of-buildings-in-10-years

8.5.2 Second Smart Contract - Communicating

All the transactions in Table 8.3, Table 8.4, Table 8.5, or Table 8.6 found in the first column (deploy communication contract action, send data action, and fix data action), are made through smart contract functions. Both the consumer and the utility will be interacting through the use of the communication smart contract (refer to section 8.2.2 for more details about the smart contract).

Table 8.3 describes the steps a consumer takes to communicate in the network. Since the utility deploys the communication contract, it carries the burden of using up a big amount of gas. The send data actions, carried out by the consumers are transactions sent to the smart contract and the fix data actions, carried out by the utility are transactions sent to the smart contract. This table describes the logistics of the transactions.

Table 8.3: Architecture 2B: Communication in the Smart Grid Communication Network - Smart Contract #2

Transaction	From	To	Gas Used	Fee (ether)
Deploy Communication Contract	Home	Contract	194077	0.000194077
Send Data	Home	Contract	33069	0.000027365
Send Fix	Home	Contract	27186	0.000027186

Table 8.4 complements Table 8.3 and includes the dollar amount of the cost of communicating in the network for both the utility and the consumers. The communication contract will be deployed by the utility once at inception and is quite inexpensive.

Table 8.4: Architecture 2B: Cost of Transactions in Initializing the Communication Environment - Smart Contract #2

Transaction	Fee (ether)	Cost ($)	Frequency	Total Cost ($)
Deploy Communication Contract	0.000194077	0.036	1	0.036

Table 8.5 and Table 8.6 complement Table 8.3 and include the dollar amount of the cost of communicating in the network for both the utility and the consumers. The send data action (Table 8.5) will be completed by every consumer that wishes to communicate electrical data to the utility. This action could occur at 15 minute intervals, 30 minute intervals, or 60 minute intervals a day. The total cost per day is calculated for these different intervals and a significant difference

can be seen. The fix data action (Table 8.6) will be completed by the utility that wishes to communicate load balancing data to a consumer. This action could occur anytime the utility believes there is a need to send load balancing data. It could be as frequent as 1 hour or could go up to every 6 hours.

Table 8.5: Architecture 2B: Cost of Transactions in Sending Electrical Data - Smart Contract #2

Transaction	Fee (ether)	Cost ($)	Frequency (min)	Total Cost / Day ($)
Send Data	0.000027365	0.0050	15 * # HH	0.48 * # HH
Send Data	0.000027365	0.0050	30 * # HH	0.24 * # HH
Send Data	0.000027365	0.0050	60 * # HH	0.12 * # HH

Table 8.6: Architecture 2B: Cost of Transactions in Sending Load Balancing Data - Smart Contract #2

Transaction	Fee (ether)	Cost ($)	Frequency / Day	Total Cost / Day ($)
Fix Data	0.000027186	0.0050	1 * # HH	0.0050 * # HH

We assume the same statistics mentioned above about the population, the number of buildings built, and the number of households (HHs)/establishments that need electricity are applicable. Equation 8.3 shows the payment needed to be made by the utility once in the smart grid communication network. Equation 8.4 shows the maximum payment needed to be made by the consumer once every year. Finally, equation 8.5 shows the payment needed to be made by the utility every time they send a load balancing transaction to a consumer.

$$\begin{aligned} \underline{\textbf{Utility Payment \#2}} &= \text{deploy communication contract action} \\ &= 0.036\$ \end{aligned} \tag{8.3}$$

$$\begin{aligned} \underline{\textbf{Consumer Payment \#2}} &= \text{send data action} * 365 \\ &= 0.12\$ * 365 \\ &= 43.8\$ \end{aligned} \tag{8.4}$$

$$\begin{aligned} \underline{\textbf{Utility Payment \#3}} &= \text{fix data action} \\ &= 0.0050\$ \end{aligned} \tag{8.5}$$

8.5.3 Connecting to the Database

Connecting to the database in Azure can be done by either the consumer or the utility. To test the connection time, we attempt to connect to the database once every 5 minutes for the span of an 50 minutes (10 runs). Multiple different users attempt to access the database. In this test, we connect to the database using 1 user, 100 concurrent users, 200 concurrent users, and 300 concurrent users. Each of these users is a color in the graph. The x-axis shows the different time intervals at which we attempt to connect to the database. The y-axis shows the connection time in milliseconds (ms). We notice that the pattern is almost similar for all different numbers of users connecting. This shows that the connection time for 1 user is almost as much as for 300 concurrent users.

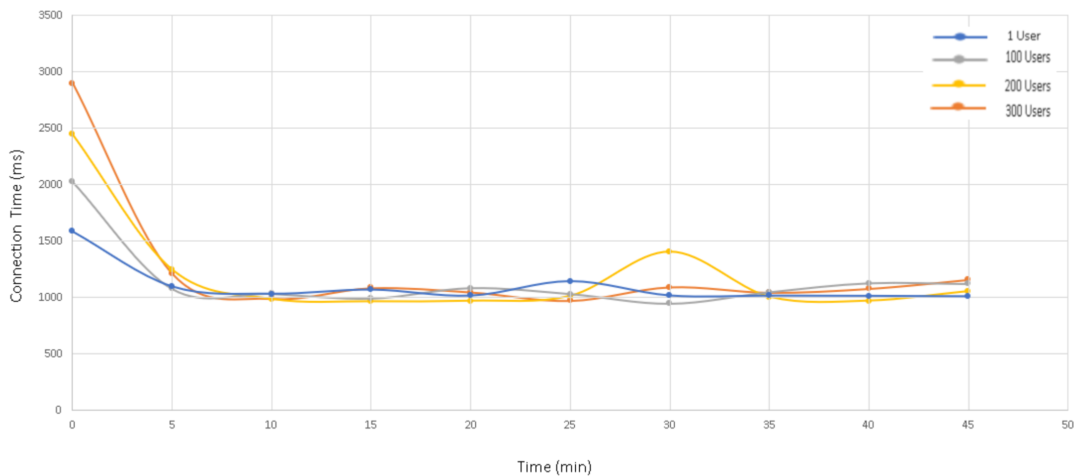

Figure 8.7: Architecture 2B: Connection Time at Intervals

Now that we've seen the connection time for each user at a point in time, we take the average of the connections of each user. For instance, throughout the 50 minute time span where the 1 user is connecting to the database, we collect the time taken to connect every 5 minutes as mentioned. These values are then averaged and form the average connection time for 1 user. The same goes for the other users. We notice that the averages are almost similar for all different numbers of users connecting.

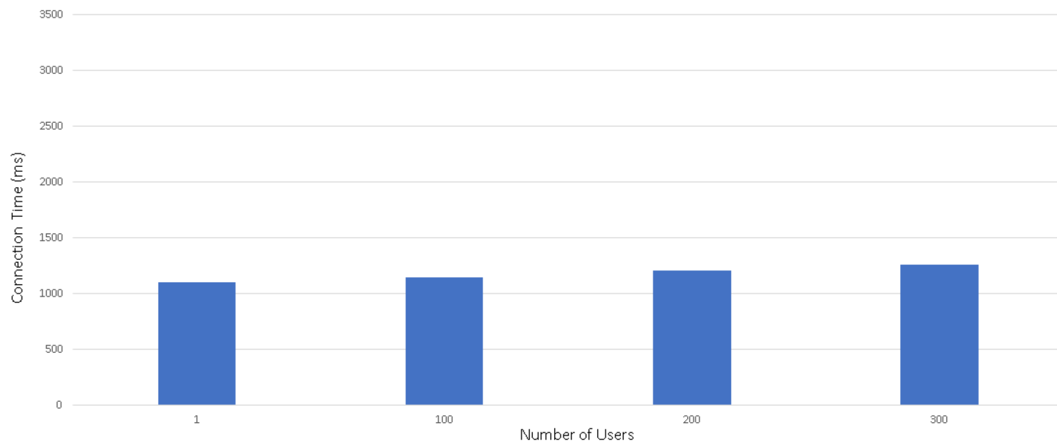

Figure 8.8: Architecture 2B: Connection Time on Average

We can state that no matter the number of users trying to connect to the database, the performance of the cloud is steady and provides good results. The average connection time is around 1.2 seconds.

8.5.4 Updating the Tables

Updating the tables in the database in Azure can be done by either the consumer or the utility. The consumer updates the "DataSent" table to add electrical data for the utility to access and the utility updates the "LoadBalancingData" table to add load balancing data for the consumer to access. To test the update time, we attempt to update the database tables once every 5 minutes for the span of an 50 minutes (10 runs). Multiple different users attempt to update the database tables. In this test, we update the database tables using 1 user, 100 concurrent users, 200 concurrent users, and 300 concurrent users. Each of these users is a color in the graph. The x-axis shows the different time intervals at which we attempt to update the database tables. The y-axis shows the connection time in milliseconds (ms). We notice that all different numbers of users updating follow a similar pattern minus a few outliers. This shows that the update time for 1 user is almost as much as for 300 concurrent users.

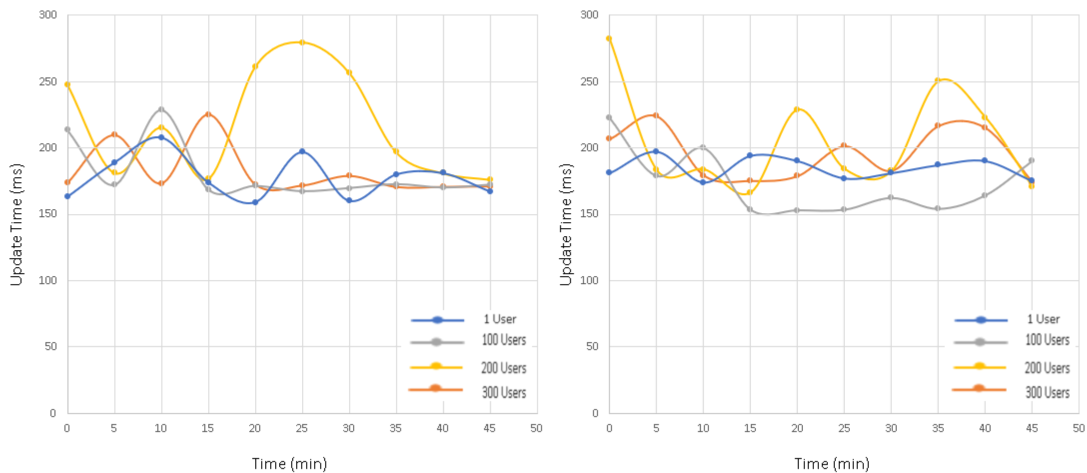

Figure 8.9: Architecture 2B: Consumer Updating at Intervals

Now that we've seen the update time for each user at a point in time, we take the average of the updates of each user. For instance, throughout the 50 minute time span where the 1 user is updating the database tables, we collect the time taken to update every 5 minutes as mentioned. These values are then averaged and form the average update time for 1 user. The same goes for the other users. We notice that the averages are almost similar for all different numbers of users connecting.

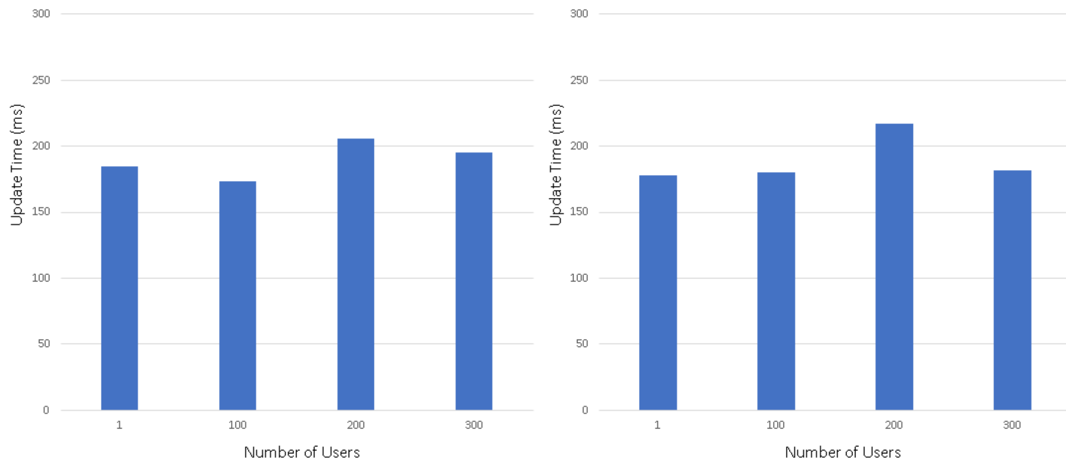

Figure 8.10: Architecture 2B: Consumer Updating on Average

We can state that no matter the number of users trying to update the database tables, the performance of the cloud is steady and provides good results. The average update time is around 0.17 seconds.

8.5.5 Querying from the Tables

Querying the tables in the database in Azure can be done by either the consumer or the utility. The consumer queries the "LoadBalancingData" table to get load balancing data from the utility and the utility updates the "DataSent" table to get electrical data from the consumer. To test the query time, we attempt to query the database tables once every 5 minutes for the span of an 50 minutes (10 runs). Multiple different users attempt to query the database tables. In this test, we query the database tables using 1 user and 10 concurrent users. Each of these users is a color in the graph. The x-axis shows the different time intervals at which we attempt to query the database tables. The y-axis shows the connection time in milliseconds (ms). We notice that the 1 user querying the database table is the utility and the pattern it follows is almost linear.

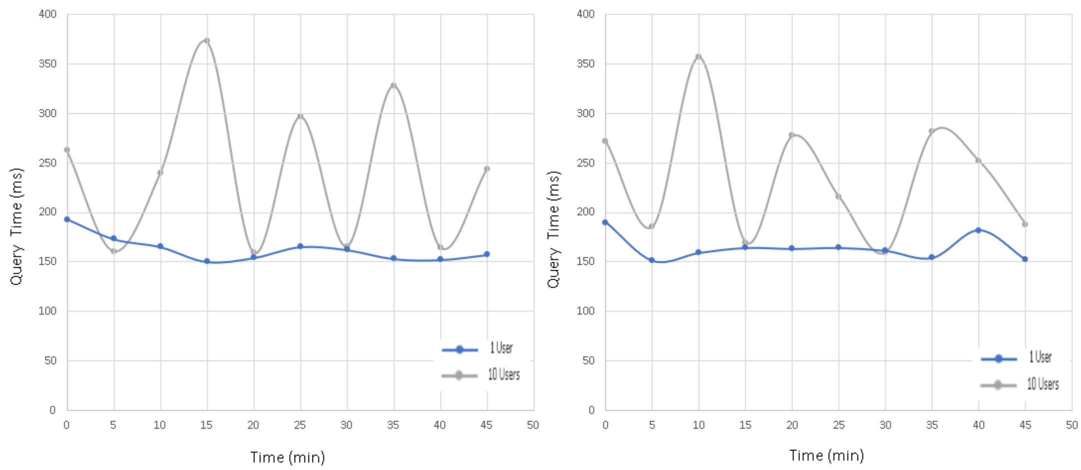

Figure 8.11: Architecture 2B: Utility Querying at Intervals

Now that we've seen the query time for each user at a point in time, we take the average of the queries of each user. For instance, throughout the 50 minute time span where the 1 user is querying the database tables, we collect the time taken to query every 5 minutes as mentioned. These values are then averaged and form the average query time for 1 user. We notice that the average for the 1 user, the utility, querying the table is almost identical for 1,000 rows, 100,000 rows, and 1,000,000 rows of electrical data.

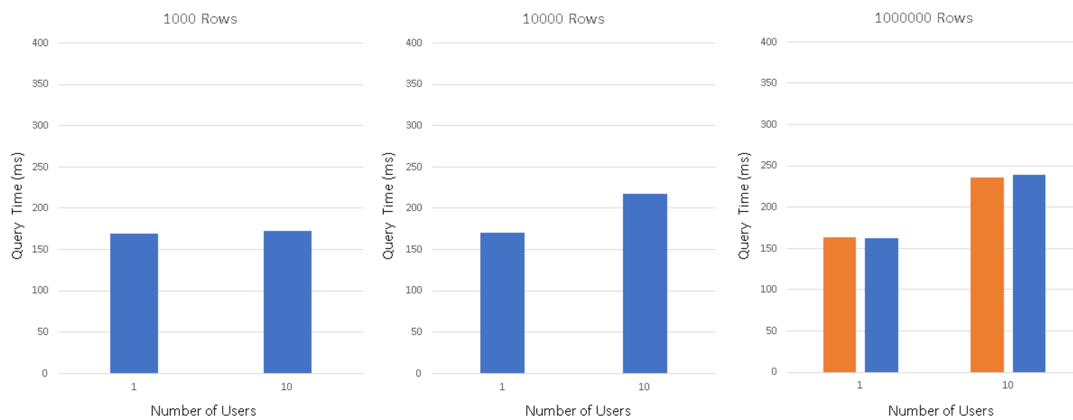

Figure 8.12: Architecture 2B: Utility Querying on Average

We can state that when the utility tries to query the database tables, the performance of the cloud is steady and provides good results. The average query time for 1,000 rows, 100,000 rows, and 1,000,000 rows of electrical data is around 0.16 seconds.

8.5.6 Summary

As observed, the prices are lower than in chapter 8's architecture. This is due to differences in variables types due to the lack of encrypted data sent. The architecture's limitations will be discussed in section 8.7.

The two smart contracts used in this architecture pose different sums to be paid by both the utility and the consumer at different points in time of the smart grid communication network. Equation 8.1, equation 8.2, equation 8.3, equation 8.4, and equation 8.5 show the payments that need to be made by the utility and the consumer at different points in the smart grid communication process. We wrap up these equations by summing up all the expenses that are paid by the utility and the consumer. Equation 8.6 sums up the expenses paid by the utility to set up the smart grid communication network for themselves and the consumers. Equation 8.7 sums up the expenses paid by the utility to interact with the utility in the smart grid communication network. Equation 8.8 sums up the expenses paid by each consumer to set up their interaction with the smart grid communication network. Finally, equation 8.9 sums up the expenses that have to be paid by each consumer yearly to interact with the utility in the smart grid communication network.

$$\begin{aligned} \underline{\textbf{Total Initial Utility Payment}} &= \text{Utility Payment } \#1 \\ &+ \text{Utility Payment } \#2 \\ &= 18,856.47\$ + 0.036\$ \\ &= 18,856.506\$ \end{aligned} \tag{8.6}$$

$$\begin{aligned} \underline{\textbf{Utility Payment Per Load Balancing Data}} &= \text{Utility Payment } \#3 \\ &= 0.0050\$ \end{aligned} \tag{8.7}$$

$$\begin{aligned} \underline{\textbf{Total Initial Consumer Payment}} &= \text{Consumer Payment } \#1 \\ &= 0.036\$ \end{aligned} \tag{8.8}$$

$$\begin{aligned} \underline{\textbf{Total Consumer Payment Per Year}} &= \text{Consumer Payment } \#2 \\ &= 43.8\$ \end{aligned} \tag{8.9}$$

8.6 Security Properties

The security properties achieved in this architecture can be credited to the use of both the blockchain network and the use of the smart contracts. The Ethereum blockchain ensures various security properties discussed in subsection 2.4.10 whereas the smart contracts and their details are discussed in subsection 2.4.2. Both of the Ethereum blockchain and its smart contracts are discussed in terms of the security properties achieved.

8.6.1 Smart Contract Properties

The architecture's two smart contracts discussed in section 8.2 subsections 8.2.1, and 8.2.2 were structured to provide certain security properties alongside the innate blockchain security properties. The smart contracts contain certain restrictions and requirements that force functions to be constrictive and serve the purpose of securing the use of the smart grid communication network. The specifics of the security properties achieved through smart contracts can be found in subsection 7.5.1.

8.6.2 Blockchain Properties

Various security properties characterize the blockchain platform. These described security properties, including confidentiality and privacy, integrity, availability, authenticity, transparency, auditability, accountability, anonymity, reliability, and termination provided by blockchain, are integral to our thesis work. All the security properties are ensured through the structure of the blockchain and the use of smart contracts in Ethereum to provide secure communication between the smart meters and the utility. The specifics of the security properties achieved through the blockchain can be found in subsection 7.5.2. Moreover, the cloud platform provides supplementary security properties.

Confidentiality and Privacy

The electrical data sent from the consumer is encrypted using the utility's public key and sent to the cloud database. Only the utility will have access to the electrical data using their private key. Thus, the data on the cloud is confidential. The hash of the electrical data sent from the consumer through a transaction will be placed on the immutable ledger. Thus, the data on the blockchain is private.

The load balancing data sent from the utility is encrypted using the consumer's public key and sent to the cloud database. Only the consumer will have access to the load balancing data using their private key. Thus, the data on the cloud is confidential. The hash of the load balancing data sent from the utility through

a transaction will be placed on the immutable ledger. Thus, the data on the blockchain is private.

Integrity

The electrical data is sent to a cloud database where it can be stored for future retrieval. The hash of the electrical data sent from the consumer through a transaction will be placed on the immutable ledger. The utility can query for the electrical data from the cloud and hash it. If this hash matches the hash queried from the blockchain, the data has not been tampered with and can be considered by the utility.

The load data is sent to a cloud database where it can be stored for future retrieval. The hash of the load balancing data sent from the utility through a transaction will be placed on the immutable ledger. The consumer can query for the load balancing data from the cloud and hash it. If this hash matches the hash queried from the blockchain, the data has not been tampered with and can be used by the consumer.

Availability

The electrical data sent from the consumer through a transaction will not be placed on the immutable ledger as it was before. The electrical data will now be sent to a cloud database where it can be stored for future retrieval. The cloud platform will always provide access to the electrical data. The load balancing data sent from the utility through a transaction will be not be placed on the immutable ledger as it was before. The load balancing data will now be sent to a cloud database where it can be stored for future retrieval. The cloud platform will always provide access to the load balancing data.

8.7 Limitations

Various security properties are achieved in this architecture, mentioned in section 8.6, that make the Ethereum blockchain a good solution for securing the two-way communication between the consumers and the utility. We will go over the limitation mentioned in chapter 8 section 7.6 to discuss how this architecture overcame it.

8.7.1 Scalability

Both the utility and the consumers in the smart grid communication network will have to send an abundance of messages. The consumers will each have to send data at intervals of 15 minutes, 30 minutes, or an hour. The utility will have to send load balancing data to consumers when required. All these transactions are not scalable on the Ethereum blockchain network. As described in table 2.1 section 2.3, the number of transactions per second in the Ethereum network is 15. This scalability issue will have to be amended in order to allow our solution to work properly and scalably.

Instead of sending electrical data every 30 minutes to the blockchain which has proven to not be scalable, we use the cloud platform to send this electrical data every 30 minutes. The hash of the electrical data will be placed on the blockchain every 4 hours thereby decreasing the transactions on the blockchain network and making this architecture more scalable than the previous architecture. This architecture will need to be made even more scalable and this can be done through improvements that are coming soon to Ethereum. It can also be made more scalable through the use of the EOS blockchain which is the intended goal for our future work.

Sharding

Sharding is the process of breaking up the ledger in the blockchain network that each node has access to. It moves away from the full node concept which is resource intensive. This benefits us in our work where the smart meters can now handle the ledger better. The node will now store a subset of the data that was found in the distributed ledger and only verifies the transactions related to the data stored in this subset. Nodes will have to trust other nodes in the network. Trust can be insured by providing incentives to ensure that nodes act in a non-malicious way. This trust poses a bit of an issue where nodes could get access to data that could have been tampered with.

Off-Chain Transactions

Creating transactions that will not be stored on the chain can also benefit the blockchain network. This solution will not require nodes to trust other nodes like in Sharding. Payment channels will be used to create off-chain channels. These micro-payment channels will be set up between two accounts and will allow transactions to be made without the need to document them all on the chain. When the accounts are done sending transactions, the channel can be closed. Both opening and closing the channel will, however, be documented on the blockchain ledger. The use of payment channels increases in importance as the number of transactions increase. This is very useful in our work where transactions must occur every 15, 30, or 60 minutes.

Proof of Stake

As mentioned in subsection 2.3.3, Proof of Stake (PoS) does not require solving computationally expensive problems. The miner is chosen pseudo-randomly depending on the node's wealth or stake. Attacking it would be expensive since to attack the network you need to own the near majority and the attacker will suffer severely from their own attack.

The process of shifting from Proof of Work (PoW) consensus to Proof of Stake consensus has already begun. Ethereum is exchanging its resource intensive consensus mechanism with PoS, a less resource intensive mechanism, to achieve a more scalable Ethereum blockchain. The shift will also make it more feasible concerning the smart meters in our work as PoS does not require high computational capabilities.

EOS

EOS is a blockchain platform that also supports decentralized applications through smart contract functionality. This blockchain platform outperforms Ethereum in terms of scalability. It can complete millions of transactions per second. The consensus mechanism used is Delegated Proof of Stake (DPoS). DPoS provides block validation of transactions in seconds. This will make our architectures completely scalable. The nodes on the EOS platform also complete transactions without the need for transaction fees. In doing so, the cost is completely eradicated in terms of our architectures.

Chapter 9

Conclusion

The shift from the traditional electric grid to the new smart grid brings with it a new set of possibilities for enhancement and improvement at the level of both efficiency and security. The smart grid is considered a heterogeneous network that inherits the security vulnerabilities of cyber systems; therefore, it puts the electric grid at risk of cyber-attacks. If the data communicated from the meters to the utility is not secured, the energy management programs used by the utility might act on wrong data and issue actions that lead to harmful consequences. The impact of security breaches in the smart grid can be severe and may affect a country's entire infrastructure, its economy, and people' s lives. These risks are further elevated at times when there are growing concerns of malicious intentions for electronic intrusions on consumer devices.

A robust and comprehensive security solution must be devised to ensure smart grid security. Hence, it is vital to ensure confidentiality, integrity, availability, authorization, authenticity, and non-repudiation in different parts of the smart grid. Regular security measures, which are employed in traditional communication and network systems, fail to secure the complex network that composes the smart grid, given the specificity of the smart grid components.

Securing the smart grid communication is imperative for the safety of its users and the integrity of the data used to improve and enhance the grid. Securing the grid using traditional techniques falls short from achieving good security due to the heterogeneity of the smart grid components. Blockchain, specifically Ethereum, presents itself as a good solution. Blockchain is emerging as a prominent solution in power systems [145] where its applications are generally concentrated on financial solutions concerning currency and payment strategies without a trusted third party. By structure, the blockchain ensures transparency and prevents any alteration of the communicated data.

In this thesis, we proposed a prevention technique to secure smart grid communication between the utility and the customer meters using blockchain as a building block. In this technique, communicated data is treated as transactions that are stored in a distributed fashion to ensure security and privacy. In addition to the blockchain and in order to ensure confidentiality and non-repudiation, we employ cryptographic and authentication techniques.

The first architecture discussed, which uses three smart contracts, provides the security properties targeted but falls short in terms of privacy. It also proved to be quite costly and did not scale well in terms of the excess storage needs of the architecture. The first part of the second architecture discussed, which uses two smart contracts, provides the security properties targeted including privacy. It also greatly improved the cost for both the utility and the consumers. However, the issue of scalability remained. Finally, the second part of the second architecture discussed, which also uses two smart contracts, provides all the security properties target and even improves the cost. The scalability issue was decreased through the use of the cloud platform. It is important to note that, in all three of these architectures, we notice an issue in the scalability of the Ethereum blockchain.

Of all the blockchain platforms, even though Ethereum is the most flexible smart contract platform, it still suffers from the issue of scalability. Improvements are being made to allow the Ethereum blockchain to scale better, thereby providing our architectures with more scalability in the process. In the meantime, our future work will focus on implementing our architectures on the EOS blockchain platform to test its performance and compare it to our current work.

Appendix A

Smart Contract Code

A.1 Architecture 1

A.1.1 First Smart Contract - Joining the Network

```
1 pragma solidity ^0.4.17;
2
3 contract Simulation {
4     address public utility;
5     uint public countNodes;
6
7     //request for node
8     struct Node {
9         string smId;
10        string mac;
11        address add;
12        bool complete;
13    }
14    Node[] public requestedNodes; //to be added to network by
       utility
15
16    mapping(address => bool) hasRequested;
17    mapping(address => bool) isDeployed;
18    mapping(address => bool) hasContract;
19
20    //only the utility can see the information
21    modifier restrictedUtility() {
22        require(msg.sender == utility);
23        _;
24    }
25
26    //only called once by utility
27    //since deployed by utility
28    function Simulation() public {
29        utility = msg.sender;
30        countNodes = 0;
```

```
    }

    //node requests access to network
    //must be accepted by utility
    function requestEntry(string smId, string mac) public {
        //shouldn't be the utility AND shouldn't be a requestor AND shouldn't already be a Node
        require(msg.sender != utility && !hasRequested[msg.sender] && !isDeployed[msg.sender]);

        Node memory r = Node(smId, mac, msg.sender, false);
        requestedNodes.push(r);
        hasRequested[msg.sender] = true;
    }

    //can only add node if the utility accepts
    function acceptNode(uint index) public restrictedUtility {
        //the node requested
        Node storage r = requestedNodes[index];

        //check if this node is valid
        //based on smId db at utility
        //done manually by calling verify function -> comparing values to values they have -> making the call

        //shouldn't already be a Node
        require(r.complete != true);

        //add node to list that needs to create a contract
        isDeployed[r.add] = true;
        countNodes++;

        //mark as complete in request
        //for ui:
        //when complete true the row is disabled
        r.complete = true;
    }

    //utility can remove a node from network
    function removeNode(uint index) public restrictedUtility {
        //the node requested
        Node storage r = requestedNodes[index];

        //should already be a Node
        require(isDeployed[r.add]);

        countNodes--;
        hasRequested[r.add] = false;
        isDeployed[r.add] = false;
        hasContract[r.add] = false;
    }
```

```
79
80      //number of requests
81      //for ui
82      function getRequestsCount() public view returns (uint) {
83          return requestedNodes.length;
84      }
85
86      //the next two functions are called one after the other
87      //the first is called...
88      //if it evaluates to true...
89      //the second is called
90      //else the second isnt called
91
92      //is node in network but does not have deployed contract
93      //evaluates to true when the node is in the network (accepted
            by the utility)
94      //and when the node has not deployed a contract yet
95      function canDeploy(address add) public view returns (bool) {
96          return isDeployed[add] && !hasContract[add];
97      }
98
99      //mark as not able to deploy data contract anymore
100     function markDone(address add) public {
101         //marking should be done by node not any other node
102         require(msg.sender == add);
103
104         hasContract[add] = true;
105     }
106 }
```

Listing A.1: Architecture 1: Joining the Network

A.1.2 Second Smart Contract - Setting Up the Communication

```
1 pragma solidity ^0.4.17;
2
3 contract Factory {
4     address[] public deployedDataContracts;
5     address public utility;
6     uint public numberContracts;
7
8     event Deployed(address indexed _add);
9
10     function Factory() public {
11         utility = msg.sender;
12     }
13
14     //create the factory contract
15     function createDataContract() public returns (address){
16         address c = new Data(utility, msg.sender);
17         deployedDataContracts.push(c);
18         numberContracts++;
19
20         Deployed(c);
21         return c;
22     }
23
24     //get addresses for all contracts
25     function getDeployedDataContracts() public view
        returns(address[]) {
26         return deployedDataContracts;
27     }
28 }
```

Listing A.2: Architecture 1: Setting Up the Communication

A.1.3 Third Smart Contract - Communicating

```
1 pragma solidity ^0.4.17;
2
3 contract Data {
4     address public owner;
5     address public utility;
6     uint public entries;
7     uint public loadBalancing;
8     mapping(string => uint) electricity;
9
10
11     //only the utility can see the information
12     modifier restrictedUtility() {
13         require(msg.sender == utility);
14         _;
15     }
16
17     //only the owner can see the information
18     modifier restrictedOwner() {
19         require(msg.sender == owner);
20         _;
21     }
22
23     function Data(address utilityAdd, address creator) public {
24         owner = creator;
25         utility = utilityAdd;
26         loadBalancing = 0;
27     }
28
29     //send the Data
30     //timestamp should follow certain convention yyyymdhm
31     function sendData(string timestamp, uint value) public
           restrictedOwner {
32         electricity[timestamp] = value;
33         entries++;
34     }
35
36     //check this data
37     function lookupData(string timestamp) view returns (uint) {
38         return electricity[timestamp];
39     }
40
41     //utility
42     //send load balancing data
43     //will be enumerator 0->decrease 1->increase ....
44     function fixData(uint value) public restrictedUtility {
45         loadBalancing = value;
46     }
47 }
```

Listing A.3: Architecture 1: Communicating

A.2 Architecture 2 (Part A)

A.2.1 First Smart Contract - Joining the Network

```
pragma solidity ^0.4.17;

contract Simulation {
    uint32 public countNodes;
    string constant public publicKey = "public key";
    address public utility;

    Node[] public requestedNodes; //to be added to network by
        utility

    mapping(address => bool) private hasRequested;
    mapping(address => bool) public isAccepted;

    //event Initialize(string k);

    //request for node
    struct Node {
        string key;
        uint32 smId;
        address add;
        bool complete;
    }

    //node requests access to network
    //must be accepted by utility
    function requestEntry(uint32 smId, string key) public {
        //shouldn't be a requestor
        //we don't have to check if it isn't already be a Node
            becuase once accepted we do not sent hasRequested to
            false...
        require(!hasRequested[msg.sender]);

        Node memory r = Node(key, smId, msg.sender, false);
        requestedNodes.push(r);
        hasRequested[msg.sender] = true;
    }

    //can only add node if the utility accepts
    function acceptNode(uint32 index) public restrictedUtility {
        //the node requested
        Node storage r = requestedNodes[index];

        //check if this node is valid
        //based on smId db at utility
        //done manually by calling verify function -> comparing
            values to values they have -> making the call

```

```
        //shouldn't already be a Node
        require(!isAccepted[r.add]);

        //add node to list that needs to create a contract
        isAccepted[r.add] = true;
        countNodes ++;

        //mark as complete in request
        //for ui:
        //when complete true the row is disabled
        r.complete = true;
    }

    //utility can remove a node from network
    function x_removeNode(uint32 index) public restrictedUtility {
        //the node requested
        Node storage r = requestedNodes[index];

        //should already be a Node
        require(isAccepted[r.add]);

        countNodes --;
        hasRequested[r.add] = false;
        isAccepted[r.add] = false;
    }

    //number of requests
    //for ui
    function getRequestsCount() public view returns (uint) {
        return requestedNodes.length;
    }

        //only the utility can see the information
    modifier restrictedUtility() {
        require(msg.sender == utility);
        _;
    }

    //only called once by utility
    //since deployed by utility
    function Simulation() public {
        utility = msg.sender;
        countNodes = 0;
    }
}
```
Listing A.4: Architecture 2A: Joining the Network

A.2.2 Second Smart Contract - Communicating

```
1 pragma solidity ^0.4.17;
2
3 contract Communication {
4     Simulation private S;
5
6     //events
7     event DataSent(address _from, bytes32 indexed _timestamp,
          string _value);
8     event LoadBalancingSent(address indexed _to, string _value);
9
10    //send the Data
11    //timestamp should follow certain convention yyyymdhm
12    function sendData(bytes32 timestamp, string value) public {
13        require(S.isAccepted(msg.sender));
14        DataSent(msg.sender,timestamp,value);
15    }
16
17    //utility
18    //send load balancing data
19    //will be enumerator 0->decrease 1->increase ....
20    function fixData(address to, string value) public {
21        require(msg.sender == S.utility());
22        LoadBalancingSent(to, value);
23    }
24
25    //constructor
26    //takes the simulation contract to refer to it later
27    function Communication(address a) public {
28        S = Simulation(a);
29    }
30 }
```

Listing A.5: Architecture 2A: Communicating

A.3 Architecture 2 (Part B)

A.3.1 First Smart Contract - Joining the Network

```
1 pragma solidity ^0.4.17;
2
3 contract Simulation {
4     uint32 public countNodes;
5     string constant public publicKey = "public key";
6     address public utility;
7
8     Node[] public requestedNodes; //to be added to network by
         utility
9
10    mapping(address => bool) private hasRequested;
11    mapping(address => bool) public isAccepted;
12
13    //request for node
14    struct Node {
15        string key;
16        uint32 smId;
17        address add;
18        bool complete;
19    }
20
21    //node requests access to network
22    //must be accepted by utility
23    function requestEntry(uint32 smId, string key) public {
24        //shouldn't be a requestor
25        //we don't have to check if it isn't already be a Node
             because once accepted we do not sent hasRequested to
             false...
26        require(!hasRequested[msg.sender]);
27
28        Node memory r = Node(key, smId, msg.sender, false);
29        requestedNodes.push(r);
30        hasRequested[msg.sender] = true;
31    }
32
33    //can only add node if the utility accepts
34    function acceptNode(uint32 index) public restrictedUtility {
35        //the node requested
36        Node storage r = requestedNodes[index];
37
38        //check if this node is valid
39        //based on smId db at utility
40        //done manually by calling verify function -> comparing
             values to values they have -> making the call
41
42        //shouldn't already be a Node
43        require(!isAccepted[r.add]);
```

```
44
45          //add node to list that needs to create a contract
46          isAccepted[r.add] = true;
47          countNodes ++;
48
49          //mark as complete in request
50          //for ui:
51          //when complete true the row is disabled
52          r.complete = true;
53      }
54
55      //utility can remove a node from network
56      function x_removeNode(uint32 index) public restrictedUtility {
57          //the node requested
58          Node storage r = requestedNodes[index];
59
60          //should already be a Node
61          require(isAccepted[r.add]);
62
63          countNodes --;
64          hasRequested[r.add] = false;
65          isAccepted[r.add] = false;
66      }
67
68      //number of requests
69      //for ui
70      function getRequestsCount() public view returns (uint) {
71          return requestedNodes.length;
72      }
73
74          //only the utility can see the information
75      modifier restrictedUtility() {
76          require(msg.sender == utility);
77          _;
78      }
79
80      //only called once by utility
81      //since deployed by utility
82      function Simulation() public {
83          utility = msg.sender;
84          countNodes = 0;
85      }
86  }
```

Listing A.6: Architecture 2B: Joining the Network

A.3.2 Second Smart Contract - Communicating

```
pragma solidity ^0.4.17;

contract Communication {
    Simulation private S;

    //events
    event DataSent(address indexed _from, bytes32 indexed
        _timestamp, bytes32 _value);
    event LoadBalancingSent(address indexed _to, bytes32 _value);

    //send the Data
    //timestamp should follow certain convention yyyymdhm
    function sendData(bytes32 timestamp, bytes32 value) public {
        require(S.isAccepted(msg.sender));
        DataSent(msg.sender,timestamp,value);
    }

    //utility
    //send load balancing data
    //will be enumerator 0->decrease 1->increase ....
    function fixData(address to, bytes32 value) public {
        require(msg.sender == S.utility());
        LoadBalancingSent(to, value);
    }

    function Communication(address a) public {
        S = Simulation(a);
    }
}
```

Listing A.7: Architecture 2B: Communicating

Appendix B

Abbreviations

SG	Smart Grid
SM	Smart Meter
BC	Blockchain
IoT	Internet of Things
MDMS	Meter Data Management System
AMI	Advanced Meter Infrastructure
DR	Demand Response
DSM	Demand Side Management
EMS	Energy Management System
PKI	Public Key Infrastructure
TTP	Trusted Third Party
EVM	Ethereum Virtual Machine
DApp	Decentralized Application
SC	Smart Contract
ABI	Application Binary Interface
TPS	Transaction per Second
HH	Household
# HH	Number of Households
CIA	Confidentiality, Integrity, Availability
JS	JavaScript
HTML	Hyper Text Markup Language
CSS	Cascading Style Sheet
CFG	Control Flow Graph
A.K.A	Also Know As
HEX	Hexadecimal
URL	Uniform Resource Locator

Bibliography

[1] Raphaelle1996, "raphaelle1996/Securing-Smart-Grid-Communication-Using-Ethereum-Smart-Contracts: First Release," May 2020. Thesis work.

[2] R. R. Mohassel, A. S. Fung, F. Mohammadi, and K. Raahemifar, "A survey on advanced metering infrastructure and its application in smart grids," in *2014 IEEE 27th Canadian Conference on Electrical and Computer Engineering (CCECE)*, pp. 1–8, IEEE, 2014.

[3] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system," tech. rep., Manubot, 2019.

[4] K. Christidis and M. Devetsikiotis, "Blockchains and smart contracts for the internet of things," *Ieee Access*, vol. 4, pp. 2292–2303, 2016.

[5] E. Staff, "Blockchains: The great chain of being sure about things," *The Economist*, vol. 18, 2016.

[6] D. Z. Morris, "Leaderless, blockchain-based venture capital fund raises $100 million, and counting," *Fortune (magazine)*, pp. 05–23, 2016.

[7] N. Popper, "A venture fund with plenty of virtual capital, but no capitalist," *New York Times*, vol. 21, 2016.

[8] "Ieee standard for low-rate wireless networks," *IEEE Std 802.15.4-2015 (Revision of IEEE Std 802.15.4-2011)*, pp. 1–709, 2016.

[9] S. Lee, J. Bong, S. Shin, and Y. Shin, "A security mechanism of smart grid ami network through smart device mutual authentication," in *The International Conference on Information Networking 2014 (ICOIN2014)*, pp. 592–595, IEEE, 2014.

[10] F. M. Cleveland, "Cyber security issues for advanced metering infrasttructure (ami)," in *2008 IEEE Power and Energy Society General Meeting-Conversion and Delivery of Electrical Energy in the 21st Century*, pp. 1–5, IEEE, 2008.

[11] A. Hahn and M. Govindarasu, "Cyber attack exposure evaluation framework for the smart grid," *IEEE Transactions on Smart Grid*, vol. 2, no. 4, pp. 835–843, 2011.

[12] S. S. S. R. Depuru, L. Wang, V. Devabhaktuni, and N. Gudi, "Smart meters for power grid—challenges, issues, advantages and status," in *2011 IEEE/PES Power Systems Conference and Exposition*, pp. 1–7, IEEE, 2011.

[13] B. Koay, S. Cheah, Y. Sng, P. Chong, P. Shum, Y. Tong, X. Wang, Y. Zuo, and H. Kuek, "Design and implementation of bluetooth energy meter," in *Fourth International Conference on Information, Communications and Signal Processing, 2003 and the Fourth Pacific Rim Conference on Multimedia. Proceedings of the 2003 Joint*, vol. 3, pp. 1474–1477, IEEE, 2003.

[14] M. Huczala, T. Lukl, and J. Misurec, "Capturing energy meter data over secured power line," in *2006 International Conference on Communication Technology*, pp. 1–4, IEEE, 2006.

[15] Y.-S. Son, T. Pulkkinen, K.-D. Moon, and C. Kim, "Home energy management system based on power line communication," *IEEE Transactions on Consumer Electronics*, vol. 56, no. 3, pp. 1380–1386, 2010.

[16] M. Bauer, W. Plappert, C. Wang, and K. Dostert, "Packet-oriented communication protocols for smart grid services over low-speed plc," in *2009 IEEE International Symposium on Power Line Communications and Its Applications*, pp. 89–94, IEEE, 2009.

[17] J. Diadamo, "Sip: the clear choice for smart grid communications," *Jun*, vol. 23, p. 3, 2009.

[18] T. Mander, H. Cheung, A. Hamlyn, L. Wang, C. Yang, and R. Cheung, "New network cyber-security architecture for smart distribution system operations," in *2008 IEEE Power and Energy Society General Meeting-Conversion and Delivery of Electrical Energy in the 21st Century*, pp. 1–8, IEEE, 2008.

[19] C. Bennett and D. Highfill, "Networking ami smart meters," in *2008 IEEE Energy 2030 Conference*, pp. 1–8, IEEE, 2008.

[20] S. Rusitschka, C. Gerdes, and K. Eger, "A low-cost alternative to smart metering infrastructure based on peer-to-peer technologies," in *2009 6th International Conference on the European Energy Market*, pp. 1–6, IEEE, 2009.

[21] P.-K. Cuvelier and P. Sommereyns, "Proof of concept smart metering," in *CIRED 2009-20th International Conference and Exhibition on Electricity Distribution-Part 1*, pp. 1–4, IET, 2009.

[22] C.-C. Sun, A. Hahn, and C.-C. Liu, "Cyber security of a power grid: State-of-the-art," *International Journal of Electrical Power & Energy Systems*, vol. 99, pp. 45–56, 2018.

[23] R. R. Mohassel, A. Fung, F. Mohammadi, and K. Raahemifar, "A survey on advanced metering infrastructure," *International Journal of Electrical Power & Energy Systems*, vol. 63, pp. 473–484, 2014.

[24] A. Molina-Markham, P. Shenoy, K. Fu, E. Cecchet, and D. Irwin, "Private memoirs of a smart meter," in *Proceedings of the 2nd ACM workshop on embedded sensing systems for energy-efficiency in building*, pp. 61–66, 2010.

[25] B. J. Murrill, E. C. Liu, and R. M. Thompson, "Smart meter data: Privacy and cybersecurity," 2012.

[26] G. Kalogridis, C. Efthymiou, S. Z. Denic, T. A. Lewis, and R. Cepeda, "Privacy for smart meters: Towards undetectable appliance load signatures," in *2010 First IEEE International Conference on Smart Grid Communications*, pp. 232–237, IEEE, 2010.

[27] A. Pfitzmann and M. Hansen, "A terminology for talking about privacy by data minimization: Anonymity, unlinkability, undetectability, unobservability, pseudonymity, and identity management," 2010.

[28] S. McLaughlin, D. Podkuiko, and P. McDaniel, "Energy theft in the advanced metering infrastructure," in *International Workshop on Critical Information Infrastructures Security*, pp. 176–187, Springer, 2009.

[29] F. Skopik, Z. Ma, T. Bleier, and H. Grüneis, "A survey on threats and vulnerabilities in smart metering infrastructures," *International Journal of Smart Grid and Clean Energy*, vol. 1, no. 1, pp. 22–28, 2012.

[30] R. Mahmud, R. Vallakati, A. Mukherjee, P. Ranganathan, and A. Nejadpak, "A survey on smart grid metering infrastructures: Threats and solutions," in *2015 IEEE International Conference on Electro/Information Technology (EIT)*, pp. 386–391, IEEE, 2015.

[31] Y. Guo, C.-W. Ten, S. Hu, and W. W. Weaver, "Preventive maintenance for advanced metering infrastructure against malware propagation," *IEEE Transactions on Smart Grid*, vol. 7, no. 3, pp. 1314–1328, 2015.

[32] Y. Park, D. M. Nicol, H. Zhu, and C. W. Lee, "Prevention of malware propagation in ami," in *2013 IEEE International Conference on Smart Grid Communications (SmartGridComm)*, pp. 474–479, IEEE, 2013.

[33] G. Liang, S. R. Weller, J. Zhao, F. Luo, and Z. Y. Dong, "The 2015 ukraine blackout: Implications for false data injection attacks," *IEEE Transactions on Power Systems*, vol. 32, no. 4, pp. 3317–3318, 2016.

[34] R. Deng, G. Xiao, R. Lu, H. Liang, and A. V. Vasilakos, "False data injection on state estimation in power systems—attacks, impacts, and defense: A survey," *IEEE Transactions on Industrial Informatics*, vol. 13, no. 2, pp. 411–423, 2016.

[35] G. Liang, J. Zhao, F. Luo, S. R. Weller, and Z. Y. Dong, "A review of false data injection attacks against modern power systems," *IEEE Transactions on Smart Grid*, vol. 8, no. 4, pp. 1630–1638, 2016.

[36] R. B. Bobba, K. M. Rogers, Q. Wang, H. Khurana, K. Nahrstedt, and T. J. Overbye, "Detecting false data injection attacks on dc state estimation," in *Preprints of the First Workshop on Secure Control Systems, CPSWEEK*, vol. 2010, 2010.

[37] M. Talebi, C. Li, and Z. Qu, "Enhanced protection against false data injection by dynamically changing information structure of microgrids," in *2012 IEEE 7th Sensor Array and Multichannel Signal Processing Workshop (SAM)*, pp. 393–396, IEEE, 2012.

[38] G. Liang, S. R. Weller, F. Luo, J. Zhao, and Z. Y. Dong, "Generalized fdia-based cyber topology attack with application to the australian electricity market trading mechanism," *IEEE Transactions on Smart Grid*, vol. 9, no. 4, pp. 3820–3829, 2017.

[39] X. Liu, Z. Bao, D. Lu, and Z. Li, "Modeling of local false data injection attacks with reduced network information," *IEEE Transactions on Smart Grid*, vol. 6, no. 4, pp. 1686–1696, 2015.

[40] X. Liu and Z. Li, "Local load redistribution attacks in power systems with incomplete network information," *IEEE Transactions on Smart Grid*, vol. 5, no. 4, pp. 1665–1676, 2014.

[41] X. Liu, Z. Li, X. Liu, and Z. Li, "Masking transmission line outages via false data injection attacks," *IEEE Transactions on Information Forensics and Security*, vol. 11, no. 7, pp. 1592–1602, 2016.

[42] O. Kosut, L. Jia, R. J. Thomas, and L. Tong, "Malicious data attacks on the smart grid," *IEEE Transactions on Smart Grid*, vol. 2, no. 4, pp. 645–658, 2011.

[43] C. Jiongcong, G. Liang, C. Zexiang, H. Chunchao, X. Yan, L. Fengji, and Z. Junhua, "Impact analysis of false data injection attacks on power system static security assessment," *Journal of Modern Power Systems and Clean Energy*, vol. 4, no. 3, pp. 496–505, 2016.

[44] L. Xie, Y. Mo, and B. Sinopoli, "Integrity data attacks in power market operations," *IEEE Transactions on Smart Grid*, vol. 2, no. 4, pp. 659–666, 2011.

[45] S. Tan, W.-Z. Song, M. Stewart, J. Yang, and L. Tong, "Online data integrity attacks against real-time electrical market in smart grid," *IEEE transactions on smart grid*, vol. 9, no. 1, pp. 313–322, 2016.

[46] S. Gong, Z. Zhang, H. Li, and A. D. Dimitrovski, "Time stamp attack in smart grid: Physical mechanism and damage analysis," *arXiv preprint arXiv:1201.2578*, 2012.

[47] Z. Zhang, S. Gong, A. D. Dimitrovski, and H. Li, "Time synchronization attack in smart grid: Impact and analysis," *IEEE Transactions on Smart Grid*, vol. 4, no. 1, pp. 87–98, 2013.

[48] X. Jiang, J. Zhang, B. J. Harding, J. J. Makela, A. D. Domı, *et al.*, "Spoofing gps receiver clock offset of phasor measurement units," *IEEE Transactions on Power Systems*, vol. 28, no. 3, pp. 3253–3262, 2013.

[49] C. Bonebrake and L. R. O'Neil, "Attacks on gps time reliability," *IEEE Security & Privacy*, vol. 12, no. 3, pp. 82–84, 2014.

[50] X. Liu and Z. Li, "Local topology attacks in smart grids," *IEEE Transactions on Smart Grid*, vol. 8, no. 6, pp. 2617–2626, 2016.

[51] J. Kim and L. Tong, "On topology attack of a smart grid: Undetectable attacks and countermeasures," *IEEE Journal on Selected Areas in Communications*, vol. 31, no. 7, pp. 1294–1305, 2013.

[52] G. Liang, S. R. Weller, J. Zhao, F. Luo, and Z. Y. Dong, "A framework for cyber-topology attacks: Line-switching and new attack scenarios," *IEEE Transactions on Smart Grid*, vol. 10, no. 2, pp. 1704–1712, 2017.

[53] A. Ashok and M. Govindarasu, "Cyber attacks on power system state estimation through topology errors," in *2012 IEEE Power and Energy Society General Meeting*, pp. 1–8, IEEE, 2012.

[54] M. A. Rahman, E. Al-Shaer, and R. Kavasseri, "Impact analysis of topology poisoning attacks on economic operation of the smart power grid," in *2014 IEEE 34th International Conference on Distributed Computing Systems*, pp. 649–659, IEEE, 2014.

[55] E. Padilla, K. Agbossou, and A. Cardenas, "Towards smart integration of distributed energy resources using distributed network protocol over ethernet," *IEEE Transactions on Smart Grid*, vol. 5, no. 4, pp. 1686–1695, 2014.

[56] M. A. P. Specification, "V1. 1b," *Modbus Organization*, 2006.

[57] R. Amoah, S. Camtepe, and E. Foo, "Securing dnp3 broadcast communications in scada systems," *IEEE Transactions on Industrial Informatics*, vol. 12, no. 4, pp. 1474–1485, 2016.

[58] G. Gilchrist, "Secure authentication for dnp3," in *2008 IEEE Power and Energy Society General Meeting-Conversion and Delivery of Electrical Energy in the 21st Century*, pp. 1–3, IEEE, 2008.

[59] G. Hayes and K. El-Khatib, "Securing modbus transactions using hash-based message authentication codes and stream transmission control protocol," in *2013 Third International Conference on Communications and Information Technology (ICCIT)*, pp. 179–184, IEEE, 2013.

[60] R. C.-W. Phan, "Authenticated modbus protocol for critical infrastructure protection," *Ieee transactions on power delivery*, vol. 27, no. 3, pp. 1687–1689, 2012.

[61] G. N. Ericsson, "Toward a framework for managing information security for an electric power utility—cigré experiences," *IEEE transactions on power delivery*, vol. 22, no. 3, pp. 1461–1469, 2007.

[62] G. N. Ericsson, "Cyber security and power system communication—essential parts of a smart grid infrastructure," *IEEE Transactions on Power Delivery*, vol. 25, no. 3, pp. 1501–1507, 2010.

[63] A. R. Metke and R. L. Ekl, "Smart grid security technology," in *2010 Innovative Smart Grid Technologies (ISGT)*, pp. 1–7, IEEE, 2010.

[64] Z. Lu, X. Lu, W. Wang, and C. Wang, "Review and evaluation of security threats on the communication networks in the smart grid," in *2010-Milcom 2010 Military Communications Conference*, pp. 1830–1835, IEEE, 2010.

[65] Q. Wang, H. Khurana, Y. Huang, and K. Nahrstedt, "Time valid one-time signature for time-critical multicast data authentication," in *IEEE INFOCOM 2009*, pp. 1233–1241, IEEE, 2009.

[66] R. Bobba, H. Khurana, M. AlTurki, and F. Ashraf, "Pbes: a policy based encryption system with application to data sharing in the power grid," in *Proceedings of the 4th international symposium on information, computer, and communications security*, pp. 262–275, 2009.

[67] H. Khurana, R. Bobba, T. Yardley, P. Agarwal, and E. Heine, "Design principles for power grid cyber-infrastructure authentication protocols," in *2010 43rd Hawaii International Conference on System Sciences*, pp. 1–10, IEEE, 2010.

[68] M. Lee, M. J. Assante, and T. Conway, "Tlp: White-analysis of the cyber attack on the ukrainian power grid-defense use case," in *2016 Electricity Information Sharing and Analysis Center (E-ISAC)*, pp. 1–29, 2016.

[69] Y. Zhang, L. Wang, W. Sun, R. C. Green II, and M. Alam, "Distributed intrusion detection system in a multi-layer network architecture of smart grids," *IEEE Transactions on Smart Grid*, vol. 2, no. 4, pp. 796–808, 2011.

[70] F. Diao, F. Zhang, and X. Cheng, "A privacy-preserving smart metering scheme using linkable anonymous credential," *IEEE Transactions on Smart Grid*, vol. 6, no. 1, pp. 461–467, 2014.

[71] C. Rottondi, G. Verticale, and A. Capone, "Privacy-preserving smart metering with multiple data consumers," *Computer Networks*, vol. 57, no. 7, pp. 1699–1713, 2013.

[72] D. Li, Z. Aung, J. Williams, and A. Sanchez, "P3: Privacy preservation protocol for automatic appliance control application in smart grid," *IEEE Internet of things Journal*, vol. 1, no. 5, pp. 414–429, 2014.

[73] F. D. Garcia and B. Jacobs, "Privacy-friendly energy-metering via homomorphic encryption," in *International Workshop on Security and Trust Management*, pp. 226–238, Springer, 2010.

[74] M. Wen, R. Lu, K. Zhang, J. Lei, X. Liang, and X. Shen, "Parq: A privacy-preserving range query scheme over encrypted metering data for smart grid," *IEEE Transactions on Emerging Topics in Computing*, vol. 1, no. 1, pp. 178–191, 2013.

[75] N. Liu, J. Chen, L. Zhu, J. Zhang, and Y. He, "A key management scheme for secure communications of advanced metering infrastructure in smart grid," *IEEE Transactions on Industrial electronics*, vol. 60, no. 10, pp. 4746–4756, 2012.

[76] Z. Wan, G. Wang, Y. Yang, and S. Shi, "Skm: Scalable key management for advanced metering infrastructure in smart grids," *IEEE Transactions on Industrial Electronics*, vol. 61, no. 12, pp. 7055–7066, 2014.

[77] M. M. Fouda, Z. M. Fadlullah, N. Kato, R. Lu, and X. S. Shen, "A lightweight message authentication scheme for smart grid communications," *IEEE Transactions on Smart grid*, vol. 2, no. 4, pp. 675–685, 2011.

[78] J. Xia and Y. Wang, "Secure key distribution for the smart grid," *IEEE Transactions on Smart Grid*, vol. 3, no. 3, pp. 1437–1443, 2012.

[79] D. Wu and C. Zhou, "Fault-tolerant and scalable key management for smart grid," *IEEE Transactions on Smart Grid*, vol. 2, no. 2, pp. 375–381, 2011.

[80] M. Nabeel, S. Kerr, X. Ding, and E. Bertino, "Authentication and key management for advanced metering infrastructures utilizing physically unclonable functions," in *2012 IEEE Third International Conference on Smart Grid Communications (SmartGridComm)*, pp. 324–329, IEEE, 2012.

[81] Z. Xiao, Y. Xiao, and D. H.-C. Du, "Non-repudiation in neighborhood area networks for smart grid," *IEEE Communications Magazine*, vol. 51, no. 1, pp. 18–26, 2013.

[82] J. Liu, Y. Xiao, and J. Gao, "Achieving accountability in smart grid," *IEEE Systems Journal*, vol. 8, no. 2, pp. 493–508, 2013.

[83] Y. Yan, Y. Qian, H. Sharif, and D. Tipper, "A survey on cyber security for smart grid communications," *IEEE Communications Surveys & Tutorials*, vol. 14, no. 4, pp. 998–1010, 2012.

[84] H. Li, R. Mao, L. Lai, and R. C. Qiu, "Compressed meter reading for delay-sensitive and secure load report in smart grid," in *2010 First IEEE International Conference on Smart Grid Communications*, pp. 114–119, IEEE, 2010.

[85] H. S. Fhom, N. Kuntze, C. Rudolph, M. Cupelli, J. Liu, and A. Monti, "A user-centric privacy manager for future energy systems," in *2010 International Conference on Power System Technology*, pp. 1–7, IEEE, 2010.

[86] C. Efthymiou and G. Kalogridis, "Smart grid privacy via anonymization of smart metering data," in *2010 first IEEE international conference on smart grid communications*, pp. 238–243, IEEE, 2010.

[87] E. Shi, A. Perrig, and L. Van Doorn, "Bind: A fine-grained attestation service for secure distributed systems," in *2005 IEEE Symposium on Security and Privacy (S&P'05)*, pp. 154–168, IEEE, 2005.

[88] K. M. Rogers, R. Klump, H. Khurana, A. A. Aquino-Lugo, and T. J. Overbye, "An authenticated control framework for distributed voltage support on the smart grid," *IEEE Transactions on Smart Grid*, vol. 1, no. 1, pp. 40–47, 2010.

[89] T. Sawa, "Blockchain technology outline and its application to field of power and energy system," *Electrical Engineering in Japan*, vol. 206, no. 2, pp. 11–15, 2019.

[90] F. Imbault, M. Swiatek, R. De Beaufort, and R. Plana, “The green blockchain: Managing decentralized energy production and consumption,” in *2017 IEEE International Conference on Environment and Electrical Engineering and 2017 IEEE Industrial and Commercial Power Systems Europe (EEEIC/I&CPS Europe)*, pp. 1–5, IEEE, 2017.

[91] E. Mengelkamp, B. Notheisen, C. Beer, D. Dauer, and C. Weinhardt, “A blockchain-based smart grid: towards sustainable local energy markets,” *Computer Science-Research and Development*, vol. 33, no. 1-2, pp. 207–214, 2018.

[92] S. Noor, W. Yang, M. Guo, K. H. van Dam, and X. Wang, “Energy demand side management within micro-grid networks enhanced by blockchain,” *Applied energy*, vol. 228, pp. 1385–1398, 2018.

[93] Z. Nehaï and G. Guerard, “Integration of the blockchain in a smart grid model,” in *Proceedings of the 14th International Conference Of Young Scientists On Energy Issues (CYSENI 2017), Kaunas, Lithuania*, pp. 25–26, 2017.

[94] A. Pieroni, N. Scarpato, L. Di Nunzio, F. Fallucchi, and M. Raso, “Smarter city: smart energy grid based on blockchain technology,” *Int. J. Adv. Sci. Eng. Inf. Technol*, vol. 8, no. 1, pp. 298–306, 2018.

[95] N. Z. Aitzhan and D. Svetinovic, “Security and privacy in decentralized energy trading through multi-signatures, blockchain and anonymous messaging streams,” *IEEE Transactions on Dependable and Secure Computing*, vol. 15, no. 5, pp. 840–852, 2016.

[96] A. Ahl, M. Yarime, K. Tanaka, and D. Sagawa, “Review of blockchain-based distributed energy: Implications for institutional development,” *Renewable and Sustainable Energy Reviews*, vol. 107, pp. 200–211, 2019.

[97] T. Winter, “The advantages and challenges of the blockchain for smart grids,” *Delft University of Technology: Delft, The Netherlands*, 2018.

[98] S. Huh, S. Cho, and S. Kim, “Managing iot devices using blockchain platform,” in *2017 19th international conference on advanced communication technology (ICACT)*, pp. 464–467, IEEE, 2017.

[99] G. Liang, S. R. Weller, F. Luo, J. Zhao, and Z. Y. Dong, “Distributed blockchain-based data protection framework for modern power systems against cyber attacks,” *IEEE Transactions on Smart Grid*, vol. 10, pp. 3162–3173, May 2019.

[100] J. Gao, K. O. Asamoah, E. B. Sifah, A. Smahi, Q. Xia, H. Xia, X. Zhang, and G. Dong, "Gridmonitoring: Secured sovereign blockchain based monitoring on smart grid," *IEEE Access*, vol. 6, pp. 9917–9925, 2018.

[101] S.-K. Kim and J.-H. Huh, "A study on the improvement of smart grid security performance and blockchain smart grid perspective," *Energies*, vol. 11, no. 8, p. 1973, 2018.

[102] Z. Guan, G. Si, X. Zhang, L. Wu, N. Guizani, X. Du, and Y. Ma, "Privacy-preserving and efficient aggregation based on blockchain for power grid communications in smart communities," *IEEE Communications Magazine*, vol. 56, no. 7, pp. 82–88, 2018.

[103] E. Reilly, M. Maloney, M. Siegel, and G. Falco, "An iot integrity-first communication protocol via an ethereum blockchain light client," in *2019 IEEE/ACM 1st International Workshop on Software Engineering Research & Practices for the Internet of Things (SERP4IoT)*, pp. 53–56, IEEE, 2019.

[104] E. Reilly, M. Maloney, M. Siegel, and G. Falco, "A smart city iot integrity-first communication protocol via an ethereum blockchain light client," in *Proceedings of the International Workshop on Software Engineering Research and Practices for the Internet of Things (SERP4IoT 2019), Marrakech, Morocco*, pp. 15–19, 2019.

[105] E. Reilly *et al.*, *An ethereum-based, integrity-first communication protocol for IoT devices*. PhD thesis, Massachusetts Institute of Technology, 2019.

[106] M. A. Khan and K. Salah, "Iot security: Review, blockchain solutions, and open challenges," *Future Generation Computer Systems*, vol. 82, pp. 395–411, 2018.

[107] D. Minoli and B. Occhiogrosso, "Blockchain mechanisms for iot security," *Internet of Things*, vol. 1, pp. 1–13, 2018.

[108] D. Fakhri and K. Mutijarsa, "Secure iot communication using blockchain technology," in *2018 International Symposium on Electronics and Smart Devices (ISESD)*, pp. 1–6, IEEE, 2018.

[109] J. Spasovski and P. Eklund, "Proof of stake blockchain: performance and scalability for groupware communications," in *Proceedings of the 9th International Conference on Management of Digital EcoSystems*, pp. 251–258, 2017.

[110] P. Urien, "Blockchain iot (biot): A new direction for solving internet of things security and trust issues," in *2018 3rd Cloudification of the Internet of Things (CIoT)*, pp. 1–4, IEEE, 2018.

[111] M. Singh, A. Singh, and S. Kim, "Blockchain: A game changer for securing iot data," in *2018 IEEE 4th World Forum on Internet of Things (WF-IoT)*, pp. 51–55, IEEE, 2018.

[112] A. S. Patil, B. A. Tama, Y. Park, and K.-H. Rhee, "A framework for blockchain based secure smart green house farming," in *Advances in Computer Science and Ubiquitous Computing*, pp. 1162–1167, Springer, 2017.

[113] K. Biswas and V. Muthukkumarasamy, "Securing smart cities using blockchain technology," in *2016 IEEE 18th international conference on high performance computing and communications; IEEE 14th international conference on smart city; IEEE 2nd international conference on data science and systems (HPCC/SmartCity/DSS)*, pp. 1392–1393, IEEE, 2016.

[114] T. Alam, "Blockchain and its role in the internet of things (iot)," *arXiv preprint arXiv:1902.09779*, 2019.

[115] K. Khacef and G. Pujolle, "Secure peer-to-peer communication based on blockchain," in *Workshops of the International Conference on Advanced Information Networking and Applications*, pp. 662–672, Springer, 2019.

[116] X. Liang, J. Zhao, S. Shetty, and D. Li, "Towards data assurance and resilience in iot using blockchain," in *MILCOM 2017-2017 IEEE Military Communications Conference (MILCOM)*, pp. 261–266, IEEE, 2017.

[117] H. Tanveer and N. Javaid, "Using ethereum blockchain technology for road toll collection on highways,"

[118] Y. N. Aung and T. Tantidham, "Review of ethereum: Smart home case study," in *2017 2nd International Conference on Information Technology (INCIT)*, pp. 1–4, IEEE, 2017.

[119] R. Wang, J. He, C. Liu, Q. Li, W.-T. Tsai, and E. Deng, "A privacy-aware pki system based on permissioned blockchains," in *2018 IEEE 9th International Conference on Software Engineering and Service Science (ICSESS)*, pp. 928–931, IEEE, 2018.

[120] T. Salman, M. Zolanvari, A. Erbad, R. Jain, and M. Samaka, "Security services using blockchains: A state of the art survey," *IEEE Communications Surveys & Tutorials*, vol. 21, no. 1, pp. 858–880, 2018.

[121] M. T. Hammi, P. Bellot, and A. Serhrouchni, "Bctrust: A decentralized authentication blockchain-based mechanism," in *2018 IEEE Wireless Communications and Networking Conference (WCNC)*, pp. 1–6, IEEE, 2018.

[122] X. Liang, J. Zhao, S. Shetty, J. Liu, and D. Li, "Integrating blockchain for data sharing and collaboration in mobile healthcare applications," in *2017 IEEE 28th Annual International Symposium on Personal, Indoor, and Mobile Radio Communications (PIMRC)*, pp. 1–5, IEEE, 2017.

[123] A. Alketbi, Q. Nasir, and M. A. Talib, "Blockchain for government services—use cases, security benefits and challenges," in *2018 15th Learning and Technology Conference (L&T)*, pp. 112–119, IEEE, 2018.

[124] A. Sudhan and M. J. Nene, "Employability of blockchain technology in defence applications," in *2017 International Conference on Intelligent Sustainable Systems (ICISS)*, pp. 630–637, IEEE, 2017.

[125] P. K. Banerjee, P. Kulkarni, and H. S. Patil, "Distributed logging of application events in a blockchain," June 11 2019. US Patent 10,320,566.

[126] N. Abdullah, A. Hakansson, and E. Moradian, "Blockchain based approach to enhance big data authentication in distributed environment," in *2017 Ninth International Conference on Ubiquitous and Future Networks (ICUFN)*, pp. 887–892, IEEE, 2017.

[127] H. Shafagh, L. Burkhalter, A. Hithnawi, and S. Duquennoy, "Towards blockchain-based auditable storage and sharing of iot data," in *Proceedings of the 2017 on Cloud Computing Security Workshop*, pp. 45–50, 2017.

[128] S. Suzuki and J. Murai, "Blockchain as an audit-able communication channel," in *2017 IEEE 41st Annual Computer Software and Applications Conference (COMPSAC)*, vol. 2, pp. 516–522, IEEE, 2017.

[129] Y. Lee and K. M. Lee, "Blockchain-based rbac for user authentication with anonymity," in *Proceedings of the Conference on Research in Adaptive and Convergent Systems*, pp. 289–294, 2019.

[130] J. Bergquist, A. Laszka, M. Sturm, and A. Dubey, "On the design of communication and transaction anonymity in blockchain-based transactive microgrids," in *Proceedings of the 1st Workshop on Scalable and Resilient Infrastructures for Distributed Ledgers*, pp. 1–6, 2017.

[131] M. Mylrea and S. N. G. Gourisetti, "Blockchain for smart grid resilience: Exchanging distributed energy at speed, scale and security," in *2017 Resilience Week (RWS)*, pp. 18–23, IEEE, 2017.

[132] Z. Lu, Q. Wang, G. Qu, and Z. Liu, "Bars: a blockchain-based anonymous reputation system for trust management in vanets," in *2018 17th IEEE International Conference On Trust, Security And Privacy In Computing And Communications/12th IEEE International Conference On Big Data Science And Engineering (TrustCom/BigDataSE)*, pp. 98–103, IEEE, 2018.

[133] S. T. Mehedi, A. A. M. Shamim, and M. B. A. Miah, "Blockchain-based security management of iot infrastructure with ethereum transactions," *Iran Journal of Computer Science*, vol. 2, no. 3, pp. 189–195, 2019.

[134] A. Bogner, M. Chanson, and A. Meeuw, "A decentralised sharing app running a smart contract on the ethereum blockchain," in *Proceedings of the 6th International Conference on the Internet of Things*, pp. 177–178, 2016.

[135] O. Novo, "Blockchain meets iot: An architecture for scalable access management in iot," *IEEE Internet of Things Journal*, vol. 5, no. 2, pp. 1184–1195, 2018.

[136] S. H. Shaheen, M. Yousaf, and M. Jalil, "Temper proof data distribution for universal verifiability and accuracy in electoral process using blockchain," in *2017 13th International Conference on Emerging Technologies (ICET)*, pp. 1–6, IEEE, 2017.

[137] C. Machado and A. A. M. Fröhlich, "Iot data integrity verification for cyber-physical systems using blockchain," in *2018 IEEE 21st International Symposium on Real-Time Distributed Computing (ISORC)*, pp. 83–90, IEEE, 2018.

[138] F. Luo, J. Zhao, Z. Y. Dong, Y. Chen, Y. Xu, X. Zhang, and K. P. Wong, "Cloud-based information infrastructure for next-generation power grid: Conception, architecture, and applications," *IEEE Transactions on Smart Grid*, vol. 7, no. 4, pp. 1896–1912, 2015.

[139] A.-H. Mohsenian-Rad and A. Leon-Garcia, "Coordination of cloud computing and smart power grids," in *2010 First IEEE International Conference on Smart Grid Communications*, pp. 368–372, IEEE, 2010.

[140] S. Rusitschka, K. Eger, and C. Gerdes, "Smart grid data cloud: A model for utilizing cloud computing in the smart grid domain," in *2010 First IEEE International Conference on Smart Grid Communications*, pp. 483–488, IEEE, 2010.

[141] H. Kim, Y.-J. Kim, K. Yang, and M. Thottan, "Cloud-based demand response for smart grid: Architecture and distributed algorithms," in *2011 IEEE international conference on smart grid communications (SmartGridComm)*, pp. 398–403, IEEE, 2011.

[142] F. Luo, Y. Chen, E. Pozorski, R. Stanic, J. Qiu, Y. Zheng, Y. Xu, and K. Meng, "Constructing a power cloud data center to deliver multi-layer it services for a smart grid," 2012.

[143] B. Bitzer and E. S. Gebretsadik, "Cloud computing framework for smart grid applications," in *2013 48th International Universities' Power Engineering Conference (UPEC)*, pp. 1–5, IEEE, 2013.

[144] M. Escalante, *Onion Router TOR, Blockchain Technology and Cybercriminals.* PhD thesis, Utica College, 2018.

[145] T. Sawa, "Blockchain technology outline and its application to field of power and energy system," *Electrical Engineering in Japan*, vol. 206, no. 2, pp. 11–15, 2019.