USING A PERMISSIONLESS BLOCKCHAIN TO BUILD A SMART DOOR LOCK

A Thesis Submitted to the
College of Graduate and Postdoctoral Studies
In Partial Fulfillment of the Requirements
For the Degree of Master of Science
In the Department of Computer Science
University of Saskatchewan
Saskatoon

By

LUCAS DE CAMARGO SILVA

ABSTRACT

Door locks connected to the internet, also known as smart locks, offer more convenience and security to control access to a place if compared to conventional locks that use physical keys or with those that use keypads. For instance, smart locks are managed remotely and even if someone once had access permission at some point, they cannot copy the key to attempt unauthorized access later. Those benefits, however, might be compromised due to the centralized system architecture offered by locks' vendors and manufacturers which allow users to control their devices - someone could gain access over the user's device and data.

This work explores how a permissionless blockchain – the public network of the Ethereum blockchain - can be leveraged to build a convenient and secure smart lock system, while giving the device owners full control over their devices by eliminating the central authority. It proposes an architecture and discusses in-depth the required components and other factors that must be taken into consideration while designing and implementing the system. Furthermore, a proof-of-concept application based on people that rent their places using hospitality services like Airbnb is implemented. The system allows hosts to remotely manage guests' permissions, delegate management rights to others, and allow guests to use a feature that blocks the owner's permission to unlock the device during their stay.

The proof-of-concept is evaluated regarding its functionalities, how long they take to be processed by the blockchain, and how much they cost to be executed. Among the findings are: (i) the proposed architecture and implementation were capable of delivering the expected behaviors for the smart lock functionalities; (ii) the delay associated with using the Ethereum blockchain are reasonable and fit the application use cases; (iii) besides the one-time-only operation to deploy the smart contract in the blockchain, the cost yielded for all other actions stayed below CAD 0.40, which is believed to be feasible considering the application context.

ACKNOWLEDGEMENTS

To my wife, Bárbara, thanks for your extraordinary support, for countless discussions about this work, and for all the revisions of this text.

To my supervisor, Dr. Ralph Deters, thanks for the unimaginable and life changing opportunity to pursue my master's degree working with you and with the colleagues from Madmuc Lab.

To my research fellow, Mayra, thanks for your help with the previous work that inspired this thesis.

To my parents, family, and friends, thanks for always being there for me.

CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

AP	Alternative Path
CAD	Canadian Dollars
DApp	Decentralized Application
EDT	Eastern Daylight Time
ETH	Ether (Ethereum currency)
EVM	Ethereum Virtual Machine
FR	Functional Requirement
HSP	Home Service Providers
HTTP	Hypertext Transfer Protocol
IDE	Integrated Development Environment
IoT	Internet of Things
IPFS	InterPlanetary File System
MSS	Main Success Scenario
P2P	Peer-to-Peer
PoC	Proof-of-Concept
PoW	Proof-of-Work
RPC	Remote Procedure Call
SHS	Smart Home Systems
Tx	Transaction
UC	Use Case
UML	Unified Modeling Language

CHAPTER 1

INTRODUCTION

Conventional door locks based on keys have some limitations, mainly due to the necessity to handle somebody else a key to allow them to open the door. Those keys can be copied, stolen, or lost, and replacing the secret is usually expensive. Door locks with keypads are an alternative to give access without the physical exchange of keys, but they face some similar issues than before. The code can also be copied, stolen, or forgotten, and although changing it is more feasible than previously, it requires physical interaction with the device. Door locks connected to the internet, also known as smart locks, solve those issues. The device is managed remotely, over the internet, and people do not have to handle physical keys or unique passcodes to others anymore. Therefore, they are a more secure and convenient alternative to manage access to places. On the other hand, the centralized system architecture used by those locks still contains some security and privacy risks.

Smart lock manufacturers provide users with an online interface to manage the devices (August, 2020; Friday, 2020; Google, 2020; Kwikset, 2020). Those solutions are entirely owned and controlled by the company, which means that they have full access over the user's data and device. Intentionally or not, someone could read, edit, or delete user's data and even acquire access to the lock without their consent. The same could happen if a hacker successfully attacked the company or even one of their IT providers, for example. In summary, a user must trust the smart lock manufacturer in order to use it.

Blockchain has been proposed to eliminate the need of central authorities – stakeholders with control power - in some applications since it allows trustless interaction between unknown parties through a decentralized architecture (Christidis and Devetsikiotis, 2016; Lu, 2018). Among hundreds of options, the Ethereum blockchain is one of the most popular choices to explore those capabilities. Ethereum supports smart contracts - pieces of codes that run in the blockchain - which enables the system to enforce custom rules or behaviors. One of the most exciting features about it is that once a smart contract is deployed, no one can change it, and it is only possible to interact with it honoring the programmed rules and behaviors. It is crucial to note, however, that Ethereum is not a single piece solution when it comes to building decentralized applications. Designing a full application to leverage it requires various components that must be chosen carefully among a complex range of possibilities.

Anyone can join and be part of the public Ethereum blockchain network, which indicates that anyone can see the transactions and other information about it. While this transparency is essential to the blockchain architecture, it also brings some privacy concerns, and it must be taken into consideration when building on the platform. A further characteristic that must be paid attention to is the economics of the Ethereum network. In order to the blockchain to properly work and avoid attacks, users must pay fees to execute transactions in it. Additionally, Ethereum needs time to process those transactions which impose a latency to actions that depend on the blockchain. Therefore, the cost to use it and latency times should be considered when designing solutions with Ethereum.

Finally, this thesis proposes a Proof-of-Concept (PoC) application that builds a smart lock system that uses the Ethereum blockchain, discussing all the relevant considerations from designing the application to building and evaluating it. This work is organized as follows: (i) Chapter two presents the problem definition and the research questions and objectives; (ii) Chapter three brings a literature review on the blockchain, and its application to smart homes and smart door locks; (iii) Chapter four defines the PoC application and discuss the system architecture to build it; (iv) Chapter five details all the considerations and decisions about the implementation of the PoC; (v) Chapter six presents the experiments to evaluate the functionalities, performance, and operating cost of the PoC; (vi) Chapter seven brings the conclusion about the work, its contributions, and discuss future work possibilities.

CHAPTER 2

PROBLEM DEFINITION

Alice needs to give Bob access to her house, but she cannot keep locking and unlocking the door for him every time he needs to enter or leave the place. The simplest solution one could think is that Alice can just handle Bob a key. Then, he can use it to enter and leave her house as needed and, once his access is no longer required, he returns the key to Alice. That solution, however, hides a much more complex set of requirements and assumptions behind the key exchange.

First, an assumption is made that Alice and Bob can physically exchange the key between them. In addition to that, Alice expects that:

- Her door can only be unlocked by her specific and unique secret, that she holds through her key;
- Bob is the only one that will use the key - i.e., the permission is individual and personalized;
- She is in control of Bob's access and that she can cease it at her will;
- No one besides her can control the access to her place – or maybe someone that she explicitly trust to do it for her. Therefore, she controls the access management.

Note that, with minor modifications, a similar set of requirements could be described if Alice would manage access to her workplace, to her school locker, to her car, among other physical places that she might be in charge of the access control.

Regular door locks that use physical keys fail to meet most of Alice's requirements because:

- Bob can make copies of that key without her consent – she is not in control over Bob's access;
- Anyone with that same key can successfully open the lock – the access is not personalized;
- Bob can provide others with access to Alice's lock – she is not in control of the access management.

Moreover, exchanging a physical key is inconvenient, and so is changing the lock's secret, which is also expensive.

Door locks that use keypads show some improvements when compared to physical keys, which are mainly related to Alice's control of one's access. However, it still fails to meet all the requirements because anyone that knows the passcode can type it in and successfully open the lock. On the other hand, exchanging the secret is more convenient than before, and so is changing the secret when needed – although it requires physical access to the device to modify it.

Door locks controlled remotely through the internet solve most of those issues found in the previous locks. It is possible, easy, and convenient to create and manage personalized permissions. However, Alice's control over managing the access might be at risk. The lock's management platforms offered by vendors and manufacturers are entirely under their control, and they could potentially override Alice's permissions – either intentionally or by suffering hacker attacks, for instance.

The blockchain has been proposed to eliminate the necessity of central authorities in various scenarios, which is precisely the problem with the current door locks connected to the internet, where the lock's provider is a central authority.

At last, this work proposes and implements a smart lock solution – partially illustrated in Figure 2.1 – which provides Alice with the following functionalities:

1. Assign individualized permission to open her lock over a time frame – e.g., let Bob unlock the door from June 15th, 3 PM, to 9:30 AM of the 18th;
2. Have full control over one's access to her lock – e.g., she can remove Bob's permission to unlock the door, and there is nothing that Bob can do to use his past authorization to unlock the device afterward;
3. Allow individuals to control the access management on her behalf – e.g., she can allow Diana to assign permissions to open the device;
4. Have full control over the access management of her lock – e.g., only Alice and people she authorizes can assign permissions to the device, and she is the only one who can create and edit authorizations;
5. Manage the lock remotely over the internet – i.e., execute the actions above without requiring physical access to the lock device.

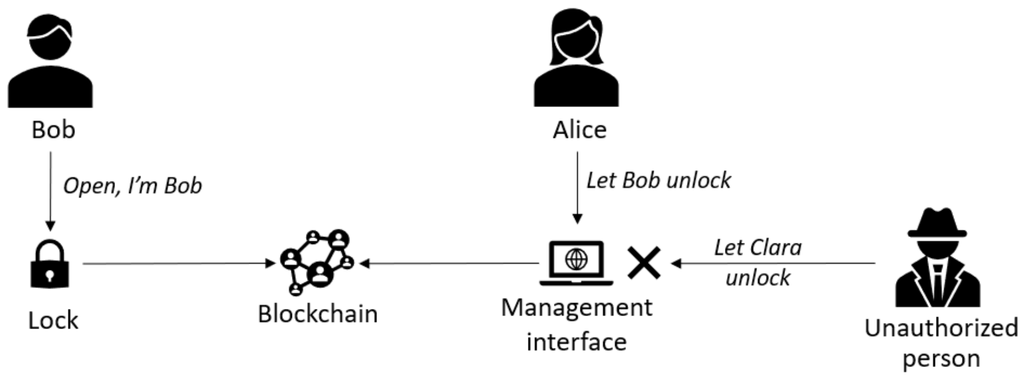


Figure 2.1 – Smart lock proposal representation

2.1 Research Questions

1. How can a smart lock solution be implemented using a permissionless blockchain to address the five functionalities presented above?
2. How much time would the proposed solution take to perform the management actions listed earlier?
3. How much money would it cost to run the proposed solution?

2.2 Research Objectives

1. Design and implement a smart lock system architecture using the blockchain
2. Regarding the five functionalities previously described:
    a. Verify that the proposed system delivers them
    b. Evaluate how long their executions are expected to take – i.e., the delay between the user sending an action and getting the corresponding response
    c. Evaluate how much money their executions are expected to cost – i.e., how much the blockchain charges the user to perform them

CHAPTER 3

LITERATURE REVIEW

This chapter presents a literature review that covers the blockchain technology, which centers on the main characteristics and applications of blockchain that are relevant to building a smart lock solution. Table 3.1, found on the last page of this chapter, shows a summary of the literature review conducted.

3.1 Blockchain

Blockchain can be explained, in summary, as a distributed ledger of transactions or, more specifically, a distributed append-only timestamped data structure (Casino et al., 2018). Satoshi Nakamoto (2008) introduced a cryptocurrency called Bitcoin, which is known as the first practical implementation of a blockchain. The proposed public peer-to-peer (P2P) network solved the double-spending problem – where one spends the same resource twice - through a consensus algorithm called proof-of-work (PoW), which enabled untrusted parties to execute transactions without relying on a central authority to validate and orchestrate it, e.g., a bank. Through asymmetric cryptography, users digitally sign their transactions, which ensures that the user indeed sent it and that the content was not exploited along the way. Following, those transactions are broadcasted to every node in the network, and inserted into blocks, with its corresponding timestamp, a process called mining. Blocks are then chained to each other in chronological order through a hash function reference. Any changes made to past transactions would alter the block's resulting hash number, causing a break in the chain, easily verifiable by anyone. In other words, all nodes in the network hold a transparent and persistent copy of the ledger. Consequently, unlike traditional centralized systems, there is not a central component – like a server or a database – that attackers could exploit to tamper with the data.

Bitcoin introduced a powerful combination of techniques that allowed the exchange and storage of cryptocurrency in a trustless and distributed environment while avoiding the previously required use of a central authority. Undoubtedly revolutionary, the application of Bitcoin was somehow limited to simple exchanges of cryptocurrency values. The architecture, on the other hand, offered the necessary characteristics to implement the idea of smart contracts proposed about a decade earlier by Szabo (1994, 1997). The author described the implementation of contracts in computer code, where a protocol would automatically enforce all the agreed terms between the parties. The inclusion of this feature into the blockchain technology enabled an extensive set of possible applications to the technology, going beyond financial to business, IoT, education, public and social services, to name a few (Casino et al., 2018; Zheng et al., 2018; Brandão et al., 2018). This movement towards smart contracts is so relevant that it created a turning point in blockchain history. Bitcoin and other cryptocurrencies are referred to as being part of the Blockchain 1.0 generation, while those with smart contracts functionalities are labeled as Blockchain 2.0 (Lu, 2018).

Generally, a blockchain will have mechanisms in place: (i) that allow participants to form a peer-to-peer network; (ii) that allow users to be uniquely addressable in the network and sign their transactions; (iii) that allow the network to verify and process transactions; (iv) that allow the network to agree on the state of the ledger; (v) optionally, it might include smart contracts features that allow the execution of computational steps given a particular transaction occurs; (vi) that allow the maintenance and upgrade of the blockchain itself; (vii) optionally, it might restrain access to the network (Christidis and Devetsikiotis, 2016; Tama et al., 2017). The last mechanism, used to control access to the network, determines if a blockchain is public, private, or consortium, also referred to as federated or hybrid (Buterin, 2015). In public blockchains, also known as permissionless, there is no restriction on who can participate in the network. Anyone can send transactions, join the consensus process, among other capabilities. On the contrary, a fully private blockchain, categorized as permissioned, centralizes the mining capability to a single node, implements a whitelist of users allowed to be part of the network, and might even constrain what kind of action each user can perform. Consortium is also a permissioned blockchain, which limits access to the network, but instead of centralizing the consensus process on a single entity, it defines a set of authorized nodes to do it. In other words, its semi-decentralized topology is a hybrid approach between public and private blockchains (Memon et al., 2018).

Finally, a summary of the key characteristics of blockchain technology is: (i) decentralization; (ii) transparency; (iii) immutability, persistency; (iv) privacy, pseudo-anonymity; (v) auditability; (vi) reliability; (vii) versatility (Seebacher and Schüritz, 2017; Zheng et al., 2018).

3.2 Ethereum

One of the leading players on blockchain 2.0 is Ethereum, which was conceptualized by Vitalik Buterin in late 2013 (Buterin, 2014a; Buterin, 2014b), and Gavin Wood launched its first specification in early 2014 (Wood, 2014). The blockchain supports smart contracts by including a built-in Turing-complete programming language, known as Ethereum Virtual Machine (EVM) code, which allows users to write arbitrary rules and create decentralized applications (Ethereum White Paper, 2019). The EVM code is a low-level bytecode language, but smart contracts are usually written in high-level languages. For instance, Solidity and Vyper are considered the most popular options for writing smart contracts, although others are available (Ethereum Developer Resources, 2019). Smart contracts deployed in the blockchain are also immutable, meaning that once they are sent there, no one can change its code, not even to fix bugs.

Ethereum blockchain contains two types of accounts: (i) externally owned accounts; (ii) contract accounts. The difference between them is that a private key controls the former, while the latter is controlled by code. However, both can hold a balance of Ether (ETH), the platform’s currency token.

Since every transaction demands some work from the network to process it, Ethereum charges fees – specified in unities of Gas - to avoid abuses and to prevent infinite loops execution in smart contracts (Wood, 2014; Ethereum White Paper, 2019). In practice, it works as a limit to how many computational steps are allowed to happen. Each of those steps corresponds to a low-level operator that has a fixed cost associated with it. If the Gas allowance assigned by a transaction is not enough to fully execute the action, all the computation is reversed, but the Gas was spent whatsoever. Miners are the nodes responsible for running the network mostly, and those fees are used to reward them for their work. They can even choose to ignore transactions if they find the Gas price set by the creator too low. In other words, the system imposes a trade-off on the transactors between Gas price and processing time.

It is possible to use Ethereum by either joining the public network or running a private/hybrid version, and each approach offers advantages and disadvantages. Zheng et al. (2018) summarize the main characteristics of public, private, and consortium blockchains. Public blockchains, which are largely decentralized, offer higher levels of immutability due to the difficulty of tampering with the data. At the same time, a large number of nodes to propagate transactions and blocks to, limits transaction throughput and raises latency times. Private and consortium blockchains, on the other hand, restrain the number of participants in the consensus process and the network. As a result, transactions are processed faster than before, but the difficulty to tamper with this centralized network architecture is lower. Besides the tread-off between security and latency, there is a third critical characteristic that must be taken into consideration, privacy. In public blockchains, all transactions are visible to the public, while permissioned blockchains control access to it. Finally, to use a public network means that there are no infrastructure costs associated with it, contrary to the private approach where one is responsible for setting up and managing the network completely (Casino et al., 2018). In addition to that, once various peers are using the network and holding value in it, there are many people with high interest in monitoring and keeping it secure, decentralizing the management as well.

It is undeniable that there are many variables involved in the decision to use a public or a hybrid blockchain. However, one can argue that decentralization is one of the main characteristics of blockchains, and partially or entirely centralizations break that premise, or at least weakness it substantially (Buterin, 2015). In fact, the Ethereum whitepaper (Ethereum White Paper, 2019) states that one of the reasons to develop Ethereum was to offer a blockchain platform that would allow people to build custom applications on top of, instead of bootstrapping their custom version of Bitcoin. The authors argue that the majority of those custom applications are too small to run their own blockchain network with a proper decentralized consensus protocol.

3.3 Privacy In Blockchain

As explained before, privacy is a crucial factor to take into consideration when using a public blockchain due to its transparency feature. Feng et al. (2019) conducted a survey on privacy issues related to blockchain and divided them into two types, identity and transaction. Identity privacy is about decoupling real identities from the transactions, as well as the links between transactions. Blockchain usually uses a random number to represent someone’s address, but it provides limited privacy. It is possible to monitor the network and reveal users’ identities, and that is why it is said to offer pseudo-anonymity (Halpin and Piekarska, 2017; Tikhomirov, 2017; Feng et al., 2019). Transaction privacy, on the other hand, means to protect the transaction content from public access, instead of broadcasting the data itself.

Different research has been conducted to study those privacy limitations in blockchains, and generally, they can be separated into two areas, they either provide tools to improve existing blockchains, or they propose creating new ones that address the issue from the design. An interested reader can find more details about these methods, privacy-aware blockchains, and further discussions in the works of Mercer (2016), Buterin (2016), Wahab (2018), and Feng et al. (2019). Wahab (2018) argues, which is corroborated by Unterweger et al. (2018), that it is not feasible to run state-of-the-art privacy technologies on blockchains that do not consider that constraint from conception. The reason is the high computation time and power that is required from them, which leads to scalability limitations, and high costs to run them in smart contract platforms, e.g., Ethereum.

The threat imposed by the data exposure will differ based on how the blockchain is being used. Generally speaking, the privacy issues related to the use case of simple cryptocurrency exchange relies mostly on people eventually learning the account owner identity, the account balance, and the detailed history of transactions executed by the account – in no particular order. In addition to that, all those transactions contain information about both parties involved, the value of the transactions, among others. On the other hand, for those applications built on blockchains that support smart contracts, the analysis is not so straightforward. Since the transactions can carry any custom data and perform custom actions, the threats will be directly related to the context of that specific use case and how the smart contract is implemented.

To illustrate those privacy concerns related to public blockchains, the e-voting application - a system that allows people to vote through the internet - is a good scenario. To build such a system, besides meeting high-security requirements, voters' privacy must be addressed. When someone casts a vote, it is not ideal that other people have knowledge of the details, but it is crucial that anyone can verify the validity of a vote, as well as that the eventual result is reliable. Ethereum is a popular choice of blockchain to create and study e-voting applications, and a variety of approaches have been proposed (McCorry et al., 2017; Yavuz et al., 2018; Li, 2019). However, more important than discussing the details, findings, and limitations of those works, is to highlight two ideas: (i) the necessity to consider privacy when building applications for blockchains; (ii) the privacy threats depend on the use case context and how it was implemented. An interested reader can find related discussions applied to auction application built with Ethereum smart contracts in the works of Galal and Youssef (2019a, 2019b).

3.4 Blockchain Applied To Smart Home

Several definitions for the term smart home is found in the literature (Schiefer, 2015). Widely, they refer to a house that incorporates devices capable of executing automated tasks, and that can be controlled remotely by the users. For instance, devices might be light bulbs, air-conditioners, house appliances, door locks, among many others. Since they are being used to control different aspects inside people's homes, they have a direct impact on their life and well-being, and successful attacks can result in severe consequences – e.g., leaking of private information, monitoring user's activity, opening doors, to name a few. For that reason, the use of blockchain has been proposed in different ways to enhance the smart home environment.

Rahman et al. (2019) present an idea to enable secure and privacy-preserving rent of Internet of Things (IoT) devices, introducing a marketplace that gathers IoT providers, travel agencies, hotels, and guests. This sharing economy of devices – e.g., lock or a light bulb - makes it possible for businesses to offer those products to guests, without owning it. Still, the system addresses some security and privacy challenges encountered when sharing such resources with third parties. It is done by using Ethereum and Hyperledger blockchains, InterPlanetary File System (IPFS) off-chain storage of multimedia data, data processing at the edge to store information more efficiently, among other components and features. However, there are a variety of ways to write smart contracts and interact with the blockchain, which can show different privacy and security levels. Therefore, a discussion on the implementation is crucial. One more element that requires investigation is the cost to run the proposed solution.

Set to explore the capabilities of combining IoT with blockchain, Joseph and Navaie (2019) propose a system to manage home appliances and resources supply in rented places. In their design, landlords can write conditions to supply resources to tenants – e.g., water and energy – and when a successful payment into the smart contract occurs, the blockchain automatically provides the resource. The system is built using a private Ethereum network customized to lower the proof-of-work (PoW) requirements, resulting in lower transaction confirmation times. The authors argue that since the participants must be authenticated and authorized into the network by the landlord, the overall security of the blockchain would not be compromised by the lower PoW values. Unfortunately, there are other vulnerabilities when adopting private blockchain networks that must be considered. First, the number of nodes is low, as probably would the number of transactions be, which makes 51% attacks easier to perform. Second, this topology requires significant technical knowledge from the landlords to properly maintain the network. Although they know the identity of all the participants, identify attacks, misconduct, and revert those actions is not easy. Third, the participants might not have many incentives to attack the network and compromise their supplies, as stated by the authors, but if any of them gets hacked, the system would be exposed. One can argue that this issue described is not relevant when dealing with water and energy supply control, once an outside attacker would hardly benefit from it. However, when dealing with door locks, as proposed in the future work section, the issue is pertinent. Lastly, all participants of the network would be able to access transaction information on the system. In which case, a privacy analysis would be interesting to reveal what kind of information is available for the participants of the network. For instance, the lock proposed as future work is said to record every entry and exit of people, which is a sensitive security piece of information to be available.

One more work exploring the intersection of blockchain and smart home IoT devices to improve on existing security and privacy issues was developed by Aung and Tantidham (2017). Their design uses a private Ethereum network, composed by a single node, to communicate and control temperature sensors and air conditioner equipment. This architecture presents similar limitations and risks to those discussed before on the work of Joseph and Navaie (2019), which also chose to run a private blockchain. In this case, however, the risk is even higher, once there is only a single node in the network. In addition to that, the proposed design includes a central database to offload some data from the blockchain, which lacks some discussion. It is not clear what information is recorded there, and how much its use can compromise the acquired privacy and security from the blockchain. The same architecture can be achieved through a variety of implementation strategies, and authors must explain the low-level system details to assess the features of their designs effectively. An example of this idea is the work done by Xu et al. (2018). Although they present a similar application, which applies the Ethereum private Blockchain to smart home systems, their detailed discussion enhances the understanding of this type of architecture and enable new analyses. Their work demonstrates a smart home application using Ethereum that monitors humidity and temperature of a room, then acts on it based on predefined thresholds. For instance, if the room temperature is found higher than set by the user, the smart contract would turn on the air conditioner. The authors describe the miner and node topology required to properly run the system, which includes the use of two computers, each one assigned to run two Ethereum miner nodes. This is done to improve security and take the computationally expensive miner responsibility from the sensor devices. It is not explicitly stated, but it seems that each place being monitored would run its own private network, or in other words, each user would have two machines running Ethereum miner nodes. There are some issues with that approach that might contradict the increase in security strategy and even the applicability of the system. First, it requires that the users keep two computers turned on twenty-four hours, which seems a waste of energy compared to the monitoring task in hand. Second, those computers are inside the same internet network, which means that if someone gains access to it, they potentially can reach all the miners – overall, the same that would happen if they were running a single miner node. Even when that is not the case, and attackers only acquire access to one of the machines, it would still represent 50% of the mining power, compromising the reliability of the blockchain.

An alternative use of a private blockchain to build smart home systems was proposed by Zhou et al. (2018). The work proposes a framework that combines a private blockchain network, presumably Hyperledger, with a public one. The topology determines that each house has its private network - with one miner node - and IoT devices, that are each a non-mining node. The devices send their data through smart contracts to be stored in the private blockchain, and they can communicate with each other to exchange information and services. The public blockchain is used to form a network of homes, that shares IoT services, and it receives the data from the private blockchains periodically. Although the architecture allows new functionalities not included in the previous works, it still faces most of the security and management issues discussed above on using private blockchains, notably when it contains a single miner node. Nonetheless, when a public blockchain is used, there are privacy concerns that must be addressed, and the authors do not provide details about it.

Following their previously discussed work on how to integrate IoT devices with blockchain to build Smart Home Systems (SHS), Aung and Tantidham (2019) present a new application inspired on their design. The authors propose an SHS, once again based on the private Ethereum blockchain, that integrates the system with external Home Service Providers (HSP) to send emergency calls given a specific condition occurs. For example, if the home security system detects a break-in, it could automatically send an emergency call to the police, and maybe to the security company hired by the household. Generally, a combination of sensors and other monitoring devices that identifies a predefined behavior could generate emergency calls to the correspondent HSP, such as the fire department, a health service agency, the hardware manufacturer, to name a few. One positive aspect of this work is the security and privacy awareness of the design, which uses a combination of encrypted messages and IPFS to prevent forgery and unintended access to emergency calls. Moreover, the authors provide a detailed explanation of how the system works and discuss the privacy and security aspects of various parts of it. Contrary to their last proposed architecture, the householders do not have to run Ethereum miner nodes, which is now the responsibility of the HSP. Although the design shows an improvement from the last application regarding the Ethereum private network usage, resulting in a potential increase in security, one can argue that the effort required to form and maintain a new network might not be worth compared to leveraging the existing public Ethereum network. In addition to that, given the smart contracts implementations, it seems that some information about the homeowners could be viewed, and even manipulated, by anyone participating in the network. Since potentially there will be different HSP collaborating in the network, this might represent an issue.

3.5 Blockchain Applied To Smart Locks

Aligned with the smart home definition introduced in section 3.4 (Schiefer, 2015), a smart lock can be described as a lock device capable of executing automated lock and unlock tasks, and that can be controlled remotely by the users.

Han et al. (2017) propose a smart door lock system, based on the blockchain, to overcome vulnerabilities to forgery and hacking encountered in smart locks. The work proposes the device’s hardware components, which include interfaces for three sensors to monitor the surroundings and aid the system in decision-making situations. The capabilities offered by this solution are lock/unlock the door given that the user is within a determined distance range from the device, detect indoor intrusion while the device is locked, and finally detect outside intrusion, like someone trying to tamper with the lock. Although the work addresses some smart lock vulnerabilities from an appealing perspective, combining blockchain and sensors, it is still in early stages, and more information and evaluation is required. First, it is not clear where the described functionalities run, embedded on the device, or deployed on the blockchain, which is not the same. Second, the proposed architecture apparently uses a custom private blockchain, either built from scratch or derived from an existing one, e.g., Ethereum. Given that, the devices and users must authenticate themselves to the network and manage their accounts in order to interact with it, processes that are not discussed. In addition to that, the system diagram suggests that the blockchain nodes are running on the users' smartphone and on the smart lock itself. Since running a proof-of-work node usually requires computationally intensive operations, more discussion about the blockchain implementation and its interface with mobiles would be required. Moreover, the authors mention a reduction in the proof-of-work requirements to increase the transaction speed, but this might also lower the security of the blockchain. Furthermore, if their implementation requires a new private network to operate each smart lock, the number of participants would be low, maybe a few people that live in the house, for instance, which also compromises the blockchain security features. Therefore, to properly apply this solution requires a further investigation of the vulnerabilities of the custom blockchain.

Zaparoli et al. (2019) identified convenience and security issues related to the exchange of keys between hosts, guests, and other stakeholders of Airbnb and other hospitality service providers. The work proposes a solution that includes a smart lock, a web platform to manage reservations, an Android app to communicate with the lock, and the use of Ethereum smart contracts to manage access. In the design, if someone wants to rent a property, they would open the reservation tool, book the place, pay for it, and when the time comes, they would access the venue using the Android app to unlock the door. An exciting feature introduced by this work is that, when a stay is active, only the guest can lock/unlock the device, not even the property managers or owner can open it. However, in many cases, people offering rent in such platforms also live in the place or have multiple bedrooms available, and the authors do not mention the support for those diversified access scenarios, which limits the application of their solution. Moreover, it is not clear if there are means to deal with guests' no shows, for example. A further limitation in the proposed solution is the reservation platform, where the user books and pays for rent. If they must use this new system to work with the lock, it means that they would not be able to use popular platforms like Airbnb, which offers many benefits to them on top of booking and payment.

Regarding the use of Ethereum and smart contracts by Zaparoli et al. (2019), some elements require further discussion. For instance, details about the smart contract implementation are not provided, which makes it hard to assess the security, privacy, and the functionalities coded. When working with a public blockchain, it is crucial to ensure that only authorized people can interact with the contract, that people cannot exploit it somehow, that the functionalities are reliable, and pay attention to how much data about the user is publicly available. For example, assume a hack successfully occurs at the system's database, it is essential to understand what the attackers could achieve with the information acquired on users and the smart contract addresses stored there. Besides security, privacy, and reliability, there is still the economic aspect of using Ethereum. The authors mention that for every new reservation, a new smart contract deploy happens in the blockchain, and this is one of the most expensive actions to execute (Wood, 2014). Therefore, further investigations about the operating costs to use the system are essential to understand its feasibility and propose alternative architectures to lower those values if necessary. About the proposed architecture, it relies on the server to lock/unlock the device, meaning that if it goes offline for any reason, the users would not be able to use the lock. For that reason, more discussion on that risk is necessary. Finally, the work is still in early stages, as discussed, but shows an interesting approach to address hospitality services challenges on access management.

Table 3.1 – Literature review summary

Topic	**Outcome**	**References**
Blockchain	It shows excellent characteristics, and it is suitable to build the proposed solution	12
Ethereum	It shows excellent characteristics, and it is suitable to build the proposed solution	8
Privacy in blockchain	One must pay attention to information transacted and recorded publicly in the blockchain	12
Blockchain applied to smart home	The application shows potential, but further investigation is needed	7
Blockchain applied to smart locks	The application shows potential, but further investigation is needed	4

CHAPTER 4

DESIGN AND ARCHITECTURE

Door locks are used worldwide to control access to places, and they are used in different situations, with particular frequencies, by certain people, in various contexts. Of course, open and close the lock are evident standard functionalities that a door lock should offer. However, it still requires the management of accesses and other auxiliary activities that might lead to context-specific requirements. Therefore, in order to explore the use of smart locks with blockchain, and guide the discussion within a clear scope, defining a proof-of-concept (PoC) use case scenario is a good idea. In the work of Zaparoli et al. (2019) discussed in the last chapter, the authors chose the hospitality service providers' use case – e.g., Airbnb and hotels – to develop their smart lock proposal, which has attractive characteristics for a PoC in this thesis. Therefore, a similar approach is used here but restricted to the Airbnb scenario. Figure 4.1 shows a high-level overview of the proposed PoC.

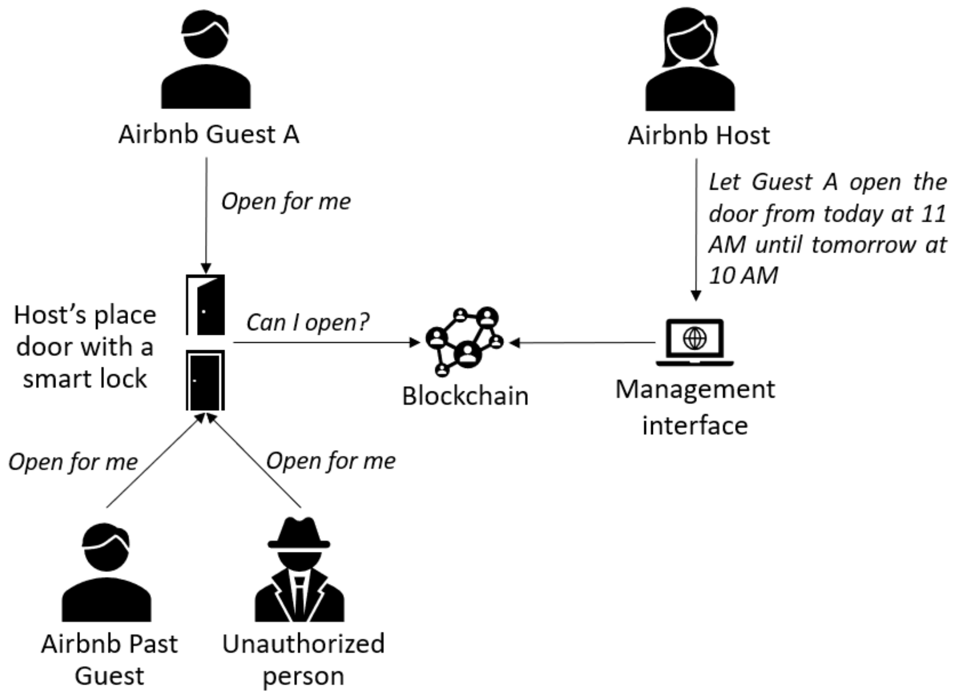


Figure 4.1 – Smart lock PoC overview

Airbnb is a peer-to-peer hospitality service platform, and it allows property owners (hosts) to find people (guests) to rent and stay in their properties – houses, apartments, or even only single rooms. Although all the booking process is facilitated by the website, where everything is done remotely and conveniently, challenges for both hosts and guests emerge when it comes to managing access to the rented places, which can compromise security, privacy, and convenience for them. First, it is significant to understand that Airbnb hosts might not be doing that as a full-time job but rather as an extra income source by renting their otherwise vacant spaces. Hosts, therefore, might have limited availability to deal with renting related tasks. Second, hosts might be using Airbnb to offer stays in their vacation properties – e.g., camping or beach houses – which they are not near to, creating even more challenges.

The security, privacy, and convenience issues addressed in this work related to managing access to the place are directly dependent on the kind of door lock that the hosts are using in their places, being either conventional locks with physical keys, locks with keypad, or smart locks – locks with internet access. The following paragraphs show the discussion for each one of them.

Conventional locks that use physical keys require hosts and guests to meet in person to exchange the door key. If their availability does not match, and they cannot agree on a meeting schedule for some reason, the host might choose to leave the key with third parties or leave it hidden and unattended somewhere. This potential sharing of keys, which can also be done by guests at any point during their stay, leads to a problem because anyone with the key can make copies of it, which can be used to access the place illegally later. An alternative for hosts would be to change the lock's secret from time to time, ideally after every guest leaves, but it is not feasible, since changing the secret of this kind of lock is usually expensive, labor-intensive, and the duration of Airbnb bookings might be for as short as a single day stay. Besides making copies, people might also just lose the key, which would incur in the same expensive costs to replace the secret. That is why conventional door locks lack convenience and compromise security and privacy for both hosts and guests.

Locks with keypad are an alternative to address some of the challenges with conventional locks. By using them, it is not mandatory that hosts and guests meet in person since they can exchange the access code remotely by various means. As a consequence, it also avoids the potential involvement of third parties that would have access to the key. However, it still faces some of the issues with the former lock. For example, people can still share the code with others, or someone can steal it by hearing, watching people type, intercepting the message where the code was exchanged, to name a few possibilities. In any case, anyone who knows the code can try to access the place illegally later. On the other hand, with keypad’s locks, it is feasible to continually change the key, even daily, which reduces the risk when compared to conventional locks, but this convenience comes with an assumption. In order to change the access code, the host must do it in person, and as stated before, they are not always close to the place. Therefore, if that is not the case, this kind of lock will face similar threats than those discussed for locks with physical keys. Moreover, even when hosts are able to change the secret daily, they still face low convenience to manage them, keeping track of the current code, and updating people about it – e.g., family members, maintenance team. That is why, to a certain degree, door locks with keypads still compromise user’s convenience, security, and privacy.

A third door lock alternative available are those equipped with an internet connection, also known as smart locks, which allow them to be controlled remotely, which brings more convenience than keypad locks. Besides the fact that it is not mandatory that hosts and guests meet in person to exchange keys, same as for keypad locks, now it is easy for hosts to manage access remotely. Accordingly, it reduces the probability that people that have had access in the past, or others that have acquired the code somehow, successfully open the lock illegally in the future. Although this solution definitely enhances convenience, security, and privacy, it does contain threats. The management platform offered to the users by the lock’s vendors or manufacturers is implemented through a centralized architecture, which they hold full control over. This control includes having access to all user’s data, having administrative power to edit or delete them, which can even mean including people in the lock’s access list, deleting someone from it, hiding information from users, or inferring when they are home or not. Therefore, employees of the company with bad intentions and hackers that successfully gain access to the company could potentially acquire control over the smart lock without the owner's knowledge. Similarly, depending on how the management platform was developed, the same could happen even if some of their IT providers are compromised – again, either an employee or hacker. Without getting into the discussion of what are the probabilities of those situations happening, the possibility itself must be enough to justify further investigations into how to enhance this architecture since door locks play a crucial security role in people's lives. Therefore, while extremely convenient, smart locks still compromise security and privacy for users.

Finally, all locks discussed above bring privacy and security issues from an Airbnb guest perspective. With hosts using any of those devices, guests can never be sure that no one will enter the place during their stay, since many might have the means to do so, and it can be either people with illegal access or even the owner.

Going back to the decision of why choosing the Airbnb use case as the PoC for this thesis, it was guided by the belief that a PoC should allow the development of a complete set of targeted requirements without being overly specialized. In that way, the discussion and knowledge gained with this study could be extensible to other use case scenarios. Generally speaking, the Airbnb scenario offers the expected lock and unlock requirements, but mixed with particular access requirements – e.g., expiry date - and business rules – e.g., ensure unique access for someone overruling even the device owner's right. Therefore, the discussion in this context could be applied to other hospitality service scenarios as hotels, to access control in the workplace or universities, to gym or school lockers, to self-storage facilities, among others. Moreover, the smart lock itself allows the study also to be extended to other IoT devices that must perform any access control over data, functionality, or any other resource.

This chapter is organized as following: section 4.1 develops the PoC use cases, and section 4.2 describes and discusses the system architecture.

4.1 PoC Use Cases

The PoC proposed has three actors, which are Host, Manager, and Guest. The Host is the smart lock owner and possibly the person who is putting the place to rent in Airbnb. A Manager is someone with management privileges over the device, which can only be provided by the Host. Managers have most of the Host's functionalities, except for register, remove, and retrieve managers. In addition to that, a Manager cannot turn on the exclusive permission feature for itself. A Guest is anyone that the Host, or any Manager, will potentially provide permission to unlock the door. The definition of the use cases to be implemented in this work started by compiling a set of functional requirements (FR). Therefore, the use cases are derived from a clear representation of expectations from all stakeholders regarding the system's functionalities. The functional requirements are presented below, where each of them contains a unique reference code, a description, and optionally details for the requirement's preconditions, postconditions, or any necessary information. All requirements are treated as mandatory for the system.

Reference	FR1
Description	The Host and Managers must be able to lock and unlock the door at any time
Required info	None
Precondition	Exclusive permission is inactive
Postcondition	None

Reference	FR2
Description	The Host and Managers must be able to remotely provide a Guest with permission to lock and unlock the door
Required info	Guest Start date and time, and expiry date and time
Precondition	None
Postcondition	None

Reference	FR3
Description	The System must automatically deactivate a Guest’s permission once the expiry date and time set are past
Required info	None
Precondition	None
Postcondition	None

Reference	FR4
Description	A Guest with permission must be able to lock and unlock the door at any time
Required info	None
Precondition 1	The attempt to unlock occurs between the Guest permission’s start and expiry dates and time
Precondition 2	Exclusive permission is inactive
Postcondition	None

Reference	FR5
Description	The Host and Managers must be able to withdraw a Guest’s permission before the expiry date remotely
Required info	None

Reference	FR5 - Continue
Precondition	The Guest's exclusive permission is inactive
Postcondition	None

Reference	FR6
Description	The Host and Managers must be able to allow one or more Guests to have simultaneous permission to lock and unlock the door
Required info	None
Precondition	None
Postcondition	None

Reference	FR7
Description	The Host and Managers must be the only ones with the power to provide or withdraw Guests' permission to lock and unlock the door
Required info	None
Precondition	None
Postcondition	None

Reference	FR8
Description	A Guest must be able to check remotely the details about its permission to lock and unlock the door
Required info	None
Precondition	None
Postcondition	Start date and time, and expiry date and time are informed

Reference	FR9
Description	The Host and Managers must be able to remotely retrieve the Guests with permission to lock and unlock the door
Required info	None
Precondition	None
Postcondition	Guests are informed along with their start date and time, and expiry date and time The list only contains Guests with the expiry date set in the future

Reference	FR10
Description	The Host must be able to remotely provide one or more people with management privileges over the device
Required info	Manager
Precondition	None
Postcondition	None

Reference	FR11
Description	The Host must be able to remove management privileges over the device from a Manager remotely
Required info	Manager
Precondition	None
Postcondition	None

Reference	FR12
Description	The Host must be able to retrieve the Managers of the device remotely
Required info	None
Precondition	None
Postcondition	Managers are informed

Reference	FR13
Description	The Host must be allowed to remotely turn on the exclusive permission to lock and unlock the door for either itself, a Manager, or a Guest
Required info	The Host, a Manager, or a Guest Expiry date and time
Precondition 1	Exclusive permission is inactive
Precondition 2	The desired exclusive permission Actor must have active permission to lock and unlock the door
Postcondition	Actor’s exclusive permission is active

Reference	FR14
Description	A Manager must be allowed to remotely turn on the exclusive permission to lock and unlock the door only for a Guest
Required info	Guest Expiry date and time
Precondition 1	Exclusive permission is inactive
Precondition 2	The desired exclusive permission Guest must have active permission to lock and unlock the door
Postcondition	Guest's exclusive permission is active

Reference	FR15
Description	The Host, Managers, and Guests must be able to remotely verify if they hold active exclusive permission to lock and unlock the door
Required info	None
Precondition	None
Postcondition	None

Reference	FR16
Description	The Host and Managers must be able to remotely verify if anyone holds active exclusive permission to lock and unlock the door
Required info	None
Precondition	None
Postcondition	Exclusive permission status is informed If any, the Actor that holds the active exclusive permission is informed, along with its related expiry date

Reference	FR17
Description	Only one Actor can hold the exclusive permission to lock and unlock the door at a given moment
Required info	None
Precondition	None
Postcondition	None

Reference	FR18
Description	Only the exclusive permission holder must be able to lock and unlock the door
Required info	None
Precondition	Exclusive permission is active for that Actor
Postcondition	None

Reference	FR19
Description	The Host, a Manager, or a Guest must be able to turn off only their own exclusive permission to lock and unlock the door
Required info	None
Precondition	The Actor must hold an active exclusive permission
Postcondition	Exclusive permission is inactive

Figure 4.2 summarizes all the use cases (UC) derived from the functional requirements detailed above. Following the picture are detailed step-by-step behavior descriptions for each one of the eleven use cases. Alternative paths (AP) to the main success scenario (MSS) are also provided when a particular step has multiple possible outcomes. However, a given step can only happen through a single behavior, either the one described in the MSS or one of the AP. For example, UC1 - unlock and lock the door - has three possible behaviors for the second step of the MSS. For a given execution, either will happen behavior MSS.2, AP.2a, or AP.2b, followed by the next steps defined by each of them. It is possible to go back from alternative paths to the main success scenario, which can lead to other alternative behaviors. For instance, a given execution for UC3 - remove Guest - could go from AP.5a to AP.6a.

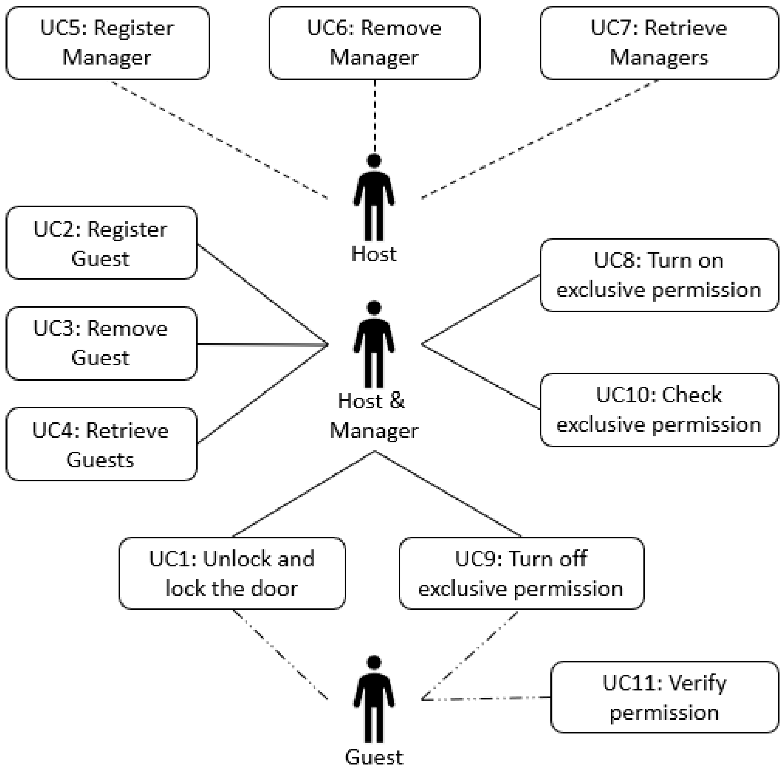


Figure 4.2 - Smart lock use cases

Reference	UC1
Name	Unlock and lock the door
Actors	Host, Manager, Guest
MSS	1. The Actor requests the smart lock to unlock or lock the door 2. The smart lock verifies that the exclusive permission feature is OFF 3. The smart lock verifies that the actor is the Host or a Manager 4. The smart lock unlocks the door
AP.1	2a.1. The smart lock verifies that the exclusive permission feature is ON 2a.2. The smart lock verifies that the actor is the same that holds the exclusive permission 2a.3. The smart lock unlocks the door
AP.2	2b.1. The smart lock verifies that the exclusive permission feature is ON 2b.2. The smart lock verifies that the actor is not the same that holds the exclusive permission 2b.3. The smart lock ignores the request

Reference	UC1 - Continue
AP.3	3a.1. The smart lock verifies that the actor is a Guest 3a.2. The smart lock verifies that the permission start date and time are in the past 3a.3. The smart lock verifies that the permission expiry date and time are in the future 3a.4. The smart lock unlocks the door
AP.4	3b.1. The smart lock verifies that the actor is a Guest 3b.2. The smart lock verifies that the permission start date and time are in the future 3b.3. The smart lock ignores the request
AP.5	3c.1. The smart lock verifies that the actor is a Guest 3c.2. The smart lock verifies that the permission start date and time are in the past 3c.3. The smart lock verifies that the permission expiry date and time are in the past 3c.4. The smart lock ignores the request

Reference	UC2
Name	Register Guest
Actors	Host, Manager
MSS	1. The Actor selects “Register Guest” in the smart lock management website 2. The Actor informs the Guest, the permission start date and time, and the permission expiry date and time 3. The Actor clicks “Register” 4. The System verifies that the request comes from either the Host or a Manager 5. The System verifies that the Guest does not have permission registered 6. The System registers the Guest permission
AP.1	4a.1. The System verifies that the request does not come from either the Host or a Manager 4a.2. The System ignores the request 4a.3. The System informs the error
AP.2	5a.1. The System verifies that the Guest has permission registered 5a.2. The System overrides the existing permission with the data from step 2

Reference	UC3
Name	Remove Guest
Actors	Host, Manager
MSS	1. The Actor selects “Remove Guest” in the smart lock management website 2. The Actor informs the Guest 3. The Actor clicks “Remove” 4. The System verifies that the request comes from either the Host or a Manager 5. The System verifies that the exclusive permission feature is OFF 6. The System verifies that the Guest has permission registered 7. The System terminates the Guest permission
AP.1	4a.1. The System verifies that the request does not come from either the Host or a Manager 4a.2. The System ignores the request 4a.3. The System informs the error
AP.2	5a.1. The System verifies that the exclusive permission feature is ON 5a.2. The System verifies that the Guest is not the exclusive permission holder 5a.3. Go back to step 6 of the main success scenario
AP.3	5b.1. The System verifies that the exclusive permission feature is ON 5b.2. The System verifies that the Guest is the exclusive permission holder 5b.3. The System ignores the request 5b.4. The System informs the error
AP.4	6a.1. The System verifies that the Guest does not have permission registered 6a.2. The System ignores the request 6a.3. The System informs the error

Reference	UC4
Name	Retrieve Guests
Actors	Host, Manager
MSS	1. The Actor selects “See Guests” in the smart lock management website 2. The System verifies that the request comes from either the Host or a Manager 3. The System finds all Guests registered with expiry date and time set in the future of the request 4. The System retrieves all Guests found in step 3 with their correspondent start date and time, and expiry date and time
AP.1	2a.1. The System verifies that the request is not from either the Host or a Manager 2a.2. The System ignores the request 2a.3. The System informs the error
AP.2	3a.1. The System does not find any Guests registered with expiry date and time set in the future of the request 3a.2. The System informs that there are not Guests registered

Reference	UC5
Name	Register Manager
Actors	Host
MSS	1. The Host selects “Register Manager” in the smart lock management website 2. The Host informs the Manager 3. The Host clicks “Register” 4. The System verifies that the request comes from the Host 5. The System verifies that the informed Manager is not registered 6. The System registers the Manager
AP.1	4a.1. The System verifies that the request does not come from the Host 4a.2. The System ignores the request 4a.3. The System informs the error
AP.2	5a.1. The System verifies that the Manager is already registered 5a.2. The System ignores the request 5a.3. The System informs the error

Reference	UC6
Name	Remove Manager
Actors	Host
MSS	1. The Host selects “Remove Manager” in the smart lock management website 2. The Host informs the Manager 3. The Host clicks “Remove” 4. The System verifies that the request comes from Host 5. The System verifies that the exclusive permission feature is OFF 6. The System verifies that the Manager is registered 7. The System removes the Manager
AP.1	4a.1. The System verifies that the request does not come from the Host 4a.2. The System ignores the request 4a.3. The System informs the error
AP.2	5a.1. The System verifies that the exclusive permission feature is ON 5a.2. The System verifies that the Manager is not the exclusive permission holder 5a.3. Go back to step 6 of the main success scenario
AP.3	5b.1. The System verifies that the exclusive permission feature is ON 5b.2. The System verifies that the Manager is the exclusive permission holder 5b.3. The System ignores the request 5b.4. The System informs the error
AP.4	6a.1. The System verifies that the Manager is not registered 6a.2. The System ignores the request 6a.3. The System informs the error

Reference	UC7
Name	Retrieve Managers
Actors	Host
MSS	1. The Host selects “See Managers” in the smart lock management website 2. The System verifies that the request comes from the Host 3. The System retrieves all Managers registered

Reference	UC7 - Continue
AP.1	2a.1. The System verifies that the request does not come from the Host 2a.2. The System ignores the request 2a.3. The System informs the error
AP.2	3a.1. The System does not find any Managers registered 3a.2. The System informs that there are not Managers registered

Reference	UC8
Name	Turn on exclusive permission
Actors	Host, Manager
MSS	1. The Actor selects “Exclusive Permission – Switch ON” in the smart lock management website 2. The Actor informs the Host, a Manager, or a Guest, and the exclusive permission expiry date and time 3. The Actor clicks “Switch ON” 4. The System verifies that the request comes from the Host 5. The System verifies that the exclusive permission feature is OFF 6. The System verifies that the desired exclusive permission holder currently has active permission to lock and unlock the door 7. The System turns the exclusive permission ON
AP.1	4a.1. The System verifies that the request comes from a Manager 4a.2. Executes step 5 of the main success scenario 4a.3. The System verifies that the desired exclusive permission holder is a Guest 4a.4. Go back to step 6 of the main success scenario
AP.2	4b.1. The System verifies that the request comes from a Manager 4b.2. Executes step 5 of the main success scenario 4b.3. The System verifies that the desired exclusive permission holder is either the Host or a Manager 4b.4. The System ignores the request 4b.5. The System informs the error

Reference	UC8 - Continue
AP.3	4c.1. The System verifies that the request does not come from either the Host or a Manager 4c.2. The System ignores the request 4c.3. The System informs the error
AP.4	5a.1. The System verifies that the exclusive permission feature is ON 5a.2. The System ignores the request 5a.3. The System informs the error
AP.5	6a.1. The System verifies that the desired exclusive permission holder currently does not have active permission to lock and unlock the door 6a.2. The System ignores the request 6a.3. The System informs the error

Reference	UC9
Name	Turn off exclusive permission
Actors	Host, Manager, Guest
MSS	1. The Actor selects "Exclusive Permission – Switch OFF" in the smart lock management website 2. The System verifies that the exclusive permission feature is ON 3. The System verifies that the request comes from the exclusive permission holder 4. The System turns the exclusive permission OFF
AP.1	2a.1. The System verifies that the exclusive permission feature is OFF 2a.2. The System ignores the request 2a.3. The System informs the error
AP.2	3a.1. The System verifies that the request does not come from the exclusive permission holder 3a.2. The System ignores the request 3a.3. The System informs the error

Reference	UC10
Name	Check exclusive permission
Actors	Host, Manager
MSS	1. The Actor selects “See Exclusive Permission Details” in the smart lock management website 2. The System verifies that the request comes from either the Host or a Manager 3. The System verifies that the exclusive permission feature is ON 4. The System informs that the feature is active, the Actor who holds it, and its expiry date
AP.1	2a.1. The System verifies that the request comes from neither Host nor a Manager 2a.2. The System ignores the request 2a.3. The System informs the error
AP.2	3a.1. The System verifies that the exclusive permission feature is OFF 3a.2. The System informs that the feature is OFF

Reference	UC11
Name	Verify permission
Actors	Guest
MSS	1. The Guest selects “See My Permission Details” in the smart lock management website 2. The System verifies that the exclusive permission feature is OFF 3. The System informs the Guest permission’s start date and time, and expiry date and time
AP.1	2a.1. The System verifies that the exclusive permission feature is ON 2a.2. The System verifies that the Guest holds the exclusive permission feature 2a.3. The System informs that the Guest has an exclusive permission active, along with its expiry date 2a.4. The System informs the Guest permission’s start date and time, and expiry date and time

Reference	UC11 - Continue
AP.2	2b.1. The System verifies that the exclusive permission feature is ON 2b.2. The System verifies that the Guest is not who holds the exclusive permission feature 2b.3. Executes step 3 of the main success scenario
AP.3	3a.1. The System does not find permission registered for the Guest 3a.2. The System informs that the Guest is not registered

Table 4.1 shows a matrix mapping the relationship between all the functional requirements and the use cases, where the code in the intersection means that the use case scenario in the column addresses the functional requirement in the row. Drawing this relationship is helpful for several reasons. First, if the matrix is constructed before detailing the use cases, it supports the development of its steps focused on the desired requirements to meet, and the Actors involved. Furthermore, the matrix helps to identify alternative paths to the main success scenario and to make sure no requirement is left unaddressed. Second, the matrix can be used to understand the quality of the use case design. For example, if a use case has intersections with too many requirements, it is either because they are too general, and its scope or domain is not well defined, or the requirements provided are overly specialized, probably being minor deviations of the same behavior, and could be grouped under a common goal. The inverse relationship, where a requirement has intersections with too many use cases, also follows that same logic.

On the other hand, a use case that does not connect with any requirement means either that it is not essential in the system to achieve the desired goals, or that the requirements provided are somehow incomplete. Third, the matrix serves as a traceability tool. Accordingly, in the event of any changes being made after the design phase to any of the use cases, or requirements, it is straightforward to determine what is potentially impacted by the modification.

Table 4.1 – Functional requirements versus use cases matrix

	UC1	UC2	UC3	UC4	UC5	UC6	UC7	UC8	UC9	UC10	UC11
FR1	MSS										
FR2		MSS AP2									
FR3	AP4 AP5										
FR4	AP3										
FR5			MSS AP2 AP3 AP4								
FR6		MSS									
FR7		AP1	AP1								
FR8											MSS AP3
FR9				ALL							
FR10					ALL						
FR11						ALL					
FR12							ALL				
FR13								MSS AP3 AP5			
FR14								AP1 AP2 AP3			
FR15										ALL	ALL
FR16										ALL	
FR17								AP4			
FR18	AP1 AP2										
FR19									ALL		

4.2 System Architecture

Figure 4.3 shows the system's components, and a discussion about each of them is presented in the following pages. Please note that the architectural design goal is to discuss possibilities, concerns, limitations, and other aspects surrounding the components, rather than narrowing into specific solutions for each of them, besides the use of the Ethereum blockchain public network. The architectural discussion shows, however, how complex it is to design applications that leverage the blockchain and why it matters. In summary, it focuses on showing how it is possible to build a system around Ethereum, what are the implications of doing it, but the main emphasis of this thesis is to explore the implementation and behavior of the use cases in the blockchain.

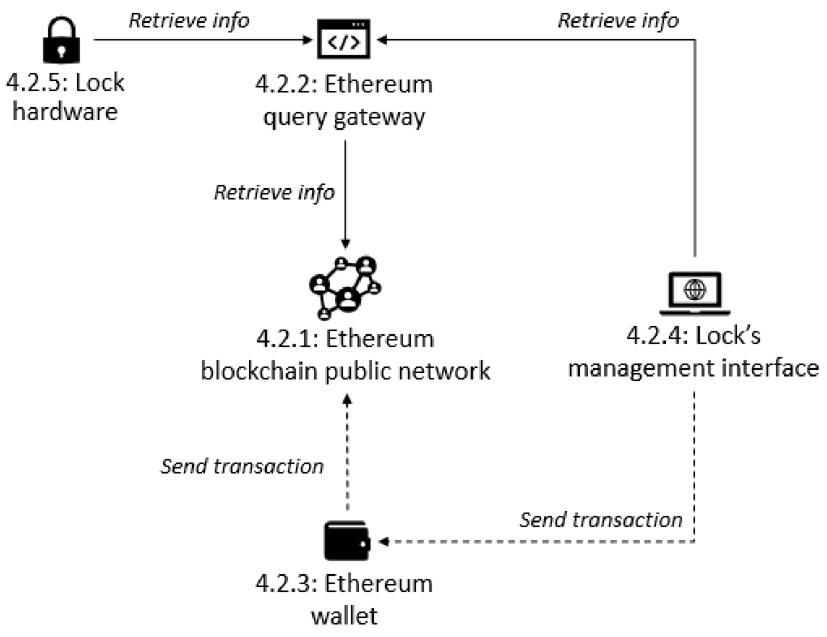


Figure 4.3 – System Architecture

4.2.1 Ethereum blockchain public network

As explained at the beginning of this chapter, smart locks that use centralized management platforms, although offering a convenient manner to remotely control the device, face some issues regarding user's security and privacy. Those issues are mostly due to the full control over the system that manufacturers or vendors have, which potentially enable user's data inspection and manipulation.

In order to address that concern, the main component of the proposed system’s architecture is the Ethereum Blockchain, which has suitable characteristics to develop a decentralized application (DApp) to manage the door lock. Blockchain is promising to overcome that limitation due to its capabilities of allowing uniquely addressable users, in a peer-to-peer network, to send, verify, and process signed transactions while agreeing in the state of the ledger. Ethereum, while leveraging the blockchain general attributes, still allows users to write arbitrary rules – programs - and deploy them to be executed on the blockchain. Among other features, it means that the code is immutable, so no one can change it once it has been published. Thus, implementing all the use cases in Ethereum smart contracts means that Host, Managers, and Guests still have the convenience to control the smart lock remotely, but through a decentralized tool in which they can trust to execute the rules as intended – assuming they were implemented correctly in the first place - without the interference of unauthorized parties. It is important to emphasize that all the rules detailed by the use cases are executed in the blockchain, written in smart contracts, so they are fully decentralized.

Ethereum can be used by joining its public network or by running a private instance of it, and it must be clearly stated since each one brings its advantages and disadvantages that must be taken into consideration at the design phase. Deciding between the two options involves a tradeoff between mainly security, transaction throughput, and managing the network infrastructure. The security aspect is directly related to the decentralization of the network. In the public environment, largely decentralized in terms of the number of nodes and their geographic locations, tampering with the data is extremely difficult. Besides the high number of malicious nodes needed to perform an attack, a large number of peers holding value in the network also means that many people are monitoring it. Private networks, slightly decentralized because they usually have only a few nodes participating in comparison with the public network, are generally less robust against attacks. The number of nodes, however, has an impact on transaction throughput and latency times. The higher the number of participants, the higher is the latency times, and fewer transactions can be processed. Finally, to run a private network, the user takes the responsibility of arranging the infrastructure equipment and of configuring, monitoring, and maintaining it, which requires financial resources, time, and technical knowledge.

Having expressed that, the system proposed will leverage the public Ethereum network due to the higher security features and less network management responsibilities that it brings, outweighing the lower transaction latency. Moreover, only a few use cases are compromised by having a latency time higher than a few seconds, namely, unlock and lock the door (UC1), retrieve guests (UC4), retrieve managers (UC7), check exclusive permission (UC10), and verify permission (UC11). However, Ethereum allows all of them to be implemented as a query to the blockchain. A query means that one is only retrieving information from the blockchain, not writing information in it – that is what a transaction does and, therefore, only transactions need to be processed by the network. For all other use cases, a latency limitation is consequently imposed, where the user must wait for the network to process their transactions. Even so, it is not believed that this limitation compromises the use cases in question since they are not sensitive to real-time execution. For example, the Host can register a guest at any point before the stay, as a start date is provided. Thus, even if the transaction takes more than 60 minutes, which is not usual, the functionality is not damaged. Of course, there could be exceptions to those use cases that would require faster responses from the blockchain, and Ethereum has a mechanism in place to address that need – it is possible to offer a higher fee price to the miners to incentivize them to process that transaction before others. However, it is out of the scope of this work to account for them, though advanced Ethereum users would still be able to take advantage of this option using the same smart contracts developed for this architecture, even if the smart contract is already in regular use.

In summary, Ethereum acts as a decentralized database while also responsible for storing and executing the smart lock business rules – in other words, the smart contracts. Given that, note that Ethereum by itself does not make up for a full application but rather its back-end structure. In order to build that, it still requires some communication mean between the blockchain and the door lock device, as well as an interface where users would manage the lock functionalities. Therefore, it is mandatory to include other components in the architecture, considering that Ethereum neither provides an API endpoint to interact with it nor is it suitable to host files. Moreover, it demands a mechanism through which users can sign their transactions before sending them to the blockchain.

4.2.2 Ethereum query gateway

For the purpose of communicating with the Ethereum network, one must have access to an Ethereum node, i.e., a client participating in the network. Those nodes offer a remote procedure call (RPC) interface that enables data exchange to occur. Therefore, the first option to build this bridge is by running a node and use it to interact with the blockchain. On the other hand, it demands infrastructure – a machine to run, internet connection, to name a few -, high technical skills to start and maintain the node, and introduces a single point of failure in the system – if the infrastructure fails, e.g., either the internet or the power at the place is down, the access to Ethereum is lost. Note, also, that this structure requires that the users trust who is hosting the node, acting as a central authority, similarly as required by the vendor of the fully centralized smart lock solution. Therefore, based on that premise only, it would not make sense to change the existing solution, i.e., it would not make sense to run an Ethereum node. Of course, it is possible to propose an architecture where each door lock owner would run its own Ethereum node, but the overhead of work, knowledge, and resources it demands, make it extremely impractical.

Due to the high demands and constraints associated with running a node, companies started offering the Ethereum infrastructure as a service, where they take care of setting up and maintaining it, plus offering simple API options to exchange data with the blockchain. While it significantly lowers the barrier of entry into the network, it implicates in placing a middleman hosting an interface between users and the blockchain, which brings back trust and centralization concerns that the blockchain was set to address in the first place. For example, a malicious API could ignore or delay relaying any transaction to Ethereum, could interrupt the service entirely, could relay information to parties other or rather than the blockchain, among others. Despite that, a valid workaround to weaken those issues would be using multiple Ethereum API providers to form the Ethereum gateway component. Consequently, the trust is divided between them – the majority wins strategy - and this creates some redundancy to access the blockchain – if any service is interrupted, there are other paths available. Some API providers are Infura (2020), Nodesmith (2020), Alchemy (2020), and Etherscan (2020).

As a consequence of the undesired reintroduction of a central authority in the blockchain application stack, while at the same time recognizing the importance of their role in abstracting the intensive demands of building a blockchain infrastructure, decentralized API networks projects and companies were developed. Since the API network itself behaves like a blockchain, similar characteristics are present, e.g., decentralization and trustless environment. Therefore, they do not break the design principle that has led to the use of Ethereum in the first place. Slock.it Incubed Client (2020) and Pocket Network (2020) are two available API providers of this nature.

Finally, note that the query word is present in the Ethereum gateway name, and the main reason is to constrain its scope of use in the architecture. In Ethereum, when someone wants to write data in the blockchain, the call is referred to as a transaction. Once they change the state of the chain, transactions are paid and take effect only after the network has processed them and confirmed their validity. On the other hand, when someone wants only to retrieve data from Ethereum, the call is named a query, which is free of charges and does not require network confirmation. The distinction in the component's name is necessary considering that, in practice, this gateway is capable of handling both transactions and queries. However, in the proposed architecture, a different component is used to perform transactions, and the reason is elaborated next.

4.2.3 Ethereum wallet

An Ethereum wallet is a software that holds and manages the user's accounts and their respective tokens. It allows users to sign transactions with their private key, and to interact with the network and applications built on top of it. The main advantage of including an existing wallet in the system architecture is because Ethereum users are already familiar with those tools, and they can choose the one that better fits their needs. There are a variety of wallet options available, and each of them addresses different requirements, e.g., desktop or mobile use, browser add-ons, hardware wallet, multiple token support, robust security features, among others. Generally, Ethereum wallets provide APIs to connect to and interact with the blockchain, not different than what was discussed for the Ethereum query gateway. Popular software libraries, e.g., JavaScript's Web3.js (2020) and Java's Web3j (2020), used to write software to interact with Ethereum can work with different providers already.

However, the trust relationship required from the user on the wallet solution is not only comparable to that discussed for APIs but even worse. A malicious wallet could acquire the means to sign transactions of an account, so an attacker could legitimately control the account. Wallet providers will address this trust issue in a particular manner. For instance, some providers have their wallet code available open-source, while others are built entirely like that. Potentially, by doing so, the software gets reviewed - or developed - by a community of users, which then reinforces the trust on the tool. The trust concern is one more reason for the architecture to allow the use of a variety of wallets, leaving users free to choose the solution which they feel comfortable using. In addition to that, when using not only Ethereum but any blockchain, the lesser the places that one must provide their account's secret key, the better it is for their security.

In any way, the proposed architecture does not leave those risks imposed by the wallet completely unaddressed. The strategy adopted is restricting the use of the wallet solely to submitting transactions to the blockchain, and then use the Ethereum query gateway to retrieve the information. Therefore, to successfully control the lock without the user's knowledge, multiple pieces must be compromised.

4.2.4 Lock's management interface

The management interface is the tool that allows Host, Managers, and Guests to control the lock functionalities, check for relevant information, and receive guidance and feedback on the actions performed. Note that this interface presents similar centralization and trust issues than those discussed for other components of this architecture, namely the Ethereum query gateway and wallet. Indeed, it acts as a middleman between the user and both the wallet and the gateway. Accordingly, it is a powerful tool once it controls not only the data flow but, more importantly, what the users see. For instance, a malicious management interface in the proposed architecture would not allow attackers to sign users' transactions, as could happen in a compromised wallet. On the other hand, they could mislead them to sign tampered ones by controlling what the interface shows to them. Ethereum wallets will usually allow users to see the transaction's data before they sign it. However, they are encoded in a low abstract level form, which is not very comprehensible for reading. In addition to that, depending on where one hosts the interface, attackers could drop its availability, prohibiting the stakeholders from managing the locks. For those reasons, the architecture must have mechanisms in place to diminish the trust requirements and centralization risks on this component.

An interesting approach is by leveraging blockchain-like storage platforms, i.e., distributed file systems, e.g., IPFS (2020) and Swarm (2020). Thus, instead of having the interface code, also referred to as the application's front-end, hosted on a specific server controlled by someone, a peer-to-peer network hosts it. Typically, a file deployed in the network is addressable by its hash string, which is accepted as unique and unpredictable. Consequently, if one makes any changes to that particular file, it will cause a change of its hash string, which then results in a different address to access it in the network. Hence, note that the trust imposed on a file is limited up to the moment that its upload to the network occurs, which takes care of keeping it unaltered throughout time. In contrast, for regular central party authority hosting, even though at some point a file might have been trusted, changes can happen. That is why trust is continuously required.

Regarding the faith still demanded on the original files, making the source code publicly available so anyone interested can review them, as some blockchain wallets are doing, is an attractive option. As a result, the trust does not lie on a single actor anymore. Instead, a community of people and organizations anchors it.

Using the strategy for hosting and managing the interface source code, as detailed above, does not impose the use of a specific design pattern, language, or framework to build the front-end tool. However, it favors having a web page rather than other types of applications. First, the management interface runs simply by accessing the file address through a web browser. Thus, it does not demand users to download and run the file somewhere, which could introduce new concerns into the architecture. Second, a web page can be accessed by any device that supports a web browser and an Ethereum wallet, entirely different from building smartphone apps or computer installed programs, which are often platform-specific. As a result, a single solution fits a broader range of users without compromising the system's use cases.

4.2.5 Lock hardware

The Lock hardware component is an abstraction of the physical infrastructure required to operate the door lock device. However, provide technical details about this piece is out of the scope of this thesis, which is limited to outline how the component fits in the proposed architecture, and what it must have to support the system's use cases.

Of course, a full hardware solution development would require discussions about security, about alternative means to guarantee the system's availability even without internet access or power supply at the place, among other needs not considered in this work. However, in the interest of the use cases, the discussion is surrounding two steps that the door lock must be able to complete: (i) receive the open and lock command from the users; (ii) verify the person's permission on the blockchain.

First, regarding the communication between the user and the lock device, this architecture presumes that the interaction ensures that an access code can only be received from its owner. For example, if they are exchanging an Ethereum account address, this communication must ensure that the person sending the information is the actual account holder. To make it clear, assume that Alice holds permission to open the lock, but Bob knows about it. There should be means in place to avoid Bob from providing Alice's permission at the door and successfully open it, and it must be at that layer. Otherwise, the blockchain will find valid permission registered. In the work of Zaparoli et al. (2019), introduced earlier in this thesis, they present an appealing message exchange protocol between a smartphone and a device that could be adapted to suit this functionality. In addition to that, for the user's convenience, the lock hardware could have physical means to lock and unlock the door from the inside without requiring to go through the blockchain to check permissions.

The system's architecture imposes two other requirements on the lock hardware: (i) it must be able to handle HTTP requests for the purpose of interacting with the Ethereum query gateway; (ii) it must be fully operational without the need to send transactions to the blockchain, only query data from it; (iii) ideally, it must ask the blockchain for a permission status every time someone tries to lock or unlock the door from the outside. The word ideally is because, for implementing a system robust against power and internet connection losses, it demands a workaround strategy on that requirement, at least for exceptional operation environment conditions.

Note that the system's architecture allows for locks to share a smart contract, being merely a matter of pointing to the same address. Therefore, in places where two or more devices are needed to operate under the same rules, no extra work is required.

CHAPTER 5

IMPLEMENTATION

Designing and developing for blockchain, as noticeable from the system's architecture discussion, involves many particular concerns not necessarily present when dealing with other types of applications. The implementation of Ethereum smart contracts also includes many specificities that can lead to different, unexpected, and undesired behaviors. Solidity offers multiple ways to code to achieve a particular outcome, not differently from many general-purpose programming languages. However, when dealing with Ethereum smart contracts, one must pay attention to aspects like cost and privacy, for instance, surrounding the development.

Therefore, this chapter presents the implementation process and relevant discussions about it. It starts with section 5.1, which details the PoC architecture to be implemented. Section 5.2 brings the development tools used throughout the process. Then, section 5.3 details the implementation for each iteration. Finally, section 5.4 shows the lock's management interface.

5.1 PoC Architecture

The discussion on section 4.2 focused on the generalized architecture components is hugely relevant to understand the available options to build the parts, how they might impact the system purpose, how they fit with one another, among other aspects. Section 4.2 addresses one of the research questions for this work about how the Ethereum blockchain can be leveraged to create a smart door lock solution. However, to answer the remaining research questions, defining a PoC lower-level architecture is required – see Figure 5.1.

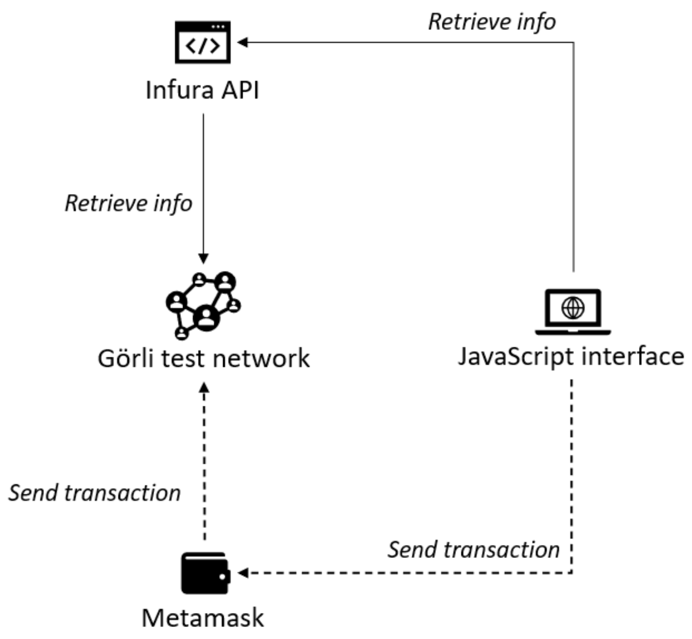


Figure 5.1 – PoC Components

Ethereum test networks are designed to emulate the Ethereum network behavior, and they are useful during the development and testing phases of smart contracts and decentralized applications. Among the reasons for their use are: (i) Ethereum test networks are free to use. They still charge the Ether associated with the transactions happening, but no money is required to acquire them there. On the other hand, Ether costs money in the public Ethereum network. Therefore, using test networks cut development costs; (ii) Ethereum test networks usually process transactions faster than the Ethereum public network. Although they emulate the Ethereum blockchain behavior, they use different mechanisms to process transactions, which might result in shorter confirmation times. Therefore, using test networks accelerates development; (iii) Ethereum test networks offload beta and development traffic from the Ethereum public network, which improves its performance; (iv) public Ethereum test networks leave its records open to anyone – transparency - like the main network does, allowing developers to inspect their smart contracts' data flow behavior. Finally, Görli (2020) is one of the Ethereum public test networks available, and it is used in this architecture to emulate the public Ethereum network present in the system's architecture – see section 4.2.1.

The Ethereum query gateway discussed in section 4.2.2 is implemented in this architecture through the use of Infura (2020). As discussed before, using a single centralized API provider for the system's architecture is not ideal. However, it works great for the context of validating the architecture and running the experiments - i.e., to build the PoC. The knowledge acquired using only Infura applies to the Ethereum query gateway component as a whole. Therefore, it provides the primary gateway functionality of bridging the management interface to the blockchain through an API, which allows for the validation of the component in the system's architecture.

As explained in section 4.2.3, the choice of the Ethereum wallet is left to the user's preference. Metamask (2020) is the choice of Ethereum wallet for the PoC because besides offering all the standard features of a wallet, it still offers a browser extension that improves the development experience.

For the lock's management interface, a web page is used in accordance with the discussion in section 4.2.4. The technology stack follows a standard web development stack combining JavaScript, HTML, and CSS, plus the use of Web3.js (2020) to interact with the wallet, which fits within the system's architecture component. As explained before, the ideal solution for this part is hosting the tool in a distributed file system. Note, however, that the interface's functionalities have the same behavior regardless of the hosting strategy. The files are the same, and the only change is from where they are being served. Therefore, for the purpose of architecture validation and user case testing of the PoC, the web page is served from a centralized host since it cuts development time and improves debugging capabilities.

Finally, as stated before, the lock hardware implementation is out of the scope of this work. However, since the query strategy is the same that the management interface tool will perform through the Infura API - providing that the lock is capable of performing the HTTP calls as the requirement presented in section 4.2.5 -, the component architecture is also validated.

5.2 Ethereum Development and Evaluation Tools

First is Remix (2020), a web-based integrated development environment (IDE) that enables writing, testing, debugging, and deploying Ethereum smart contracts. From Remix, it is possible to use Metamask to deploy and interact with the smart contract in various Ethereum networks – e.g., the Görli test network. Also, it offers a built-in sandbox blockchain where transactions happen instantly, perfect for early development stages and quick testing of smart contracts. Furthermore, Remix enables automating tests by writing scripts to run against smart contracts.

Second is Etherscan (2020a, 2020b), an Ethereum block explorer, which can search and show all data recorded in the blockchain about the blocks, transactions, and more. Etherscan supports not only the main Ethereum network but also public test networks - e.g., the Görli test network -, which is perfect to evaluate what information is made publicly available by the blockchain.

The third tool is Postman (2020), an API development tool that facilitates testing HTTP calls, among other functionalities. During the development phase, Postman is useful to simulate the calls to the smart contract coming from the lock hardware or the lock's management interface - which pass through the Infura API - and investigate their behavior early in the process.

5.3 Use Cases Implementation

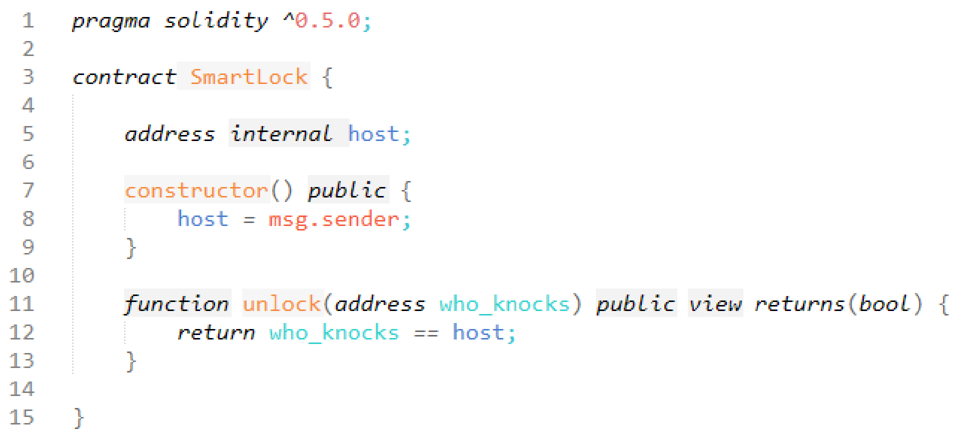
The development follows an incremental approach. It means that the use cases are gradually implemented, tested, and evaluated against different criteria, e.g., the information publicly available in the blockchain, access through Infura API, among others. If it meets the requirements and functionalities proposed, the development continues. On the contrary, it allows for early intervention, which potentially saves time and resources. This in-depth discussion is relevant to show the different aspects surrounding design decisions for writing smart contracts in Ethereum.

5.3.1 Unlock and lock the door - Host

The most basic functionality that works as the backbone structure for the application is the Host locking and unlocking the door – see UC1 in section 4.1. Therefore, it is a great place to start. The smart contract of Figure 5.2 shows the implementation of this functionality. It is essential to highlight some aspects of the smart contract:

- It is set to work with the latest full developed version of the Solidity language, v0.5;
- The *address* variable type refers to an Ethereum account public key;
- The *internal* keyword imposes that only the smart contract itself, or contracts derived from it, can read and modify the host variable;
- The constructor is called only once in the smart contract life cycle, only when it is deployed;

- The *msg.sender* value refers to the address of the account that makes the call to the smart contract. Therefore, the constructor sets as host the Ethereum address that deploys the smart contract;
- The *view* modifier on the *unlock* function means that it does not make changes to state variables, only read them. A call to this function, then, is free and does not require processing from the blockchain – also known as a query;
- The *unlock* function is meant to be called by both the lock device and the management tool, which then informs the address to be verified. Note that anyone can call it for any address and check its unlock ability. Although it enables anyone to potentially find addresses that can unlock the door, if the lock hardware follows the requirement detailed in section 4.2.5, it should not be an issue there. Also, it is not an issue for the management tool since the secret key would also be required to change anything in the contract.



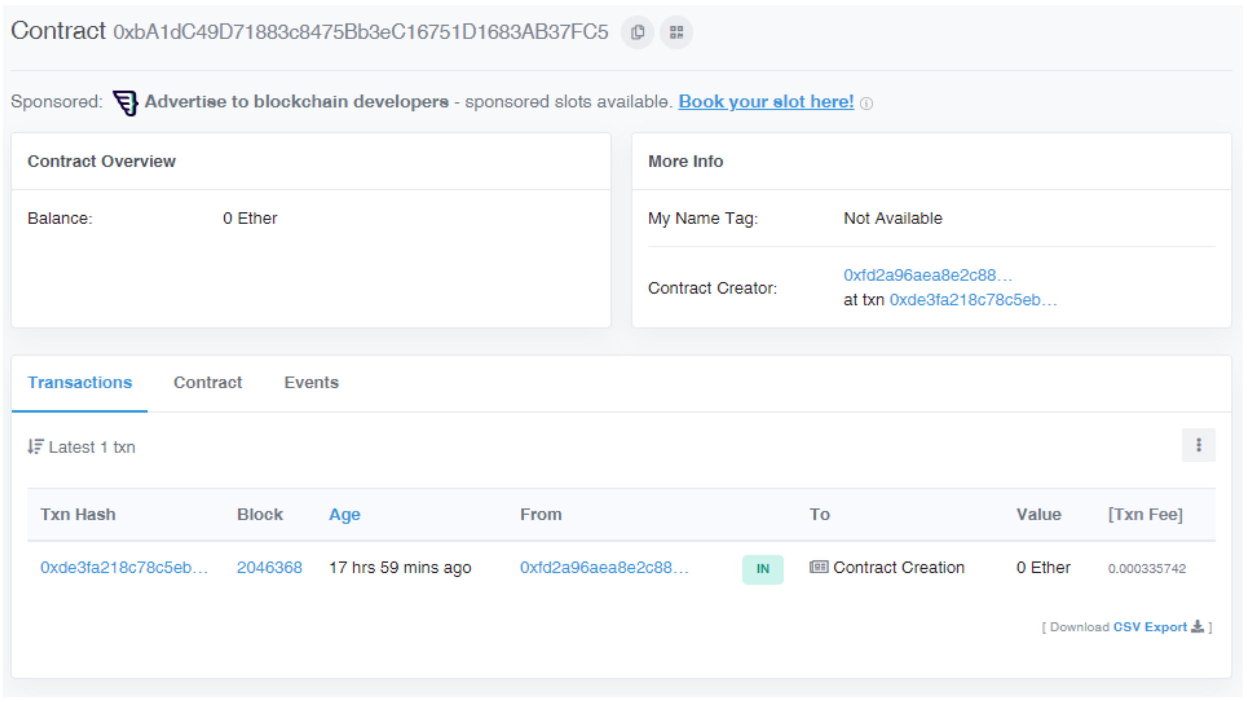
```
pragma solidity ^0.5.0;

contract SmartLock {

    address internal host;

    constructor() public {
        host = msg.sender;
    }

    function unlock(address who_knocks) public view returns(bool) {
        return who_knocks == host;
    }

}
```

Figure 5.2 – Smart Contract - Host unlock the door

Along with the smart contract, following a good programming practice, a test script to run in Remix is also developed. It automates the contract tests to ensure that past implemented functionalities still behave as expected, regardless of the current development, while saving the time that would be spent by manually testing the contract every once in a while.

As explained earlier, Metamask can be used with Remix to deploy and interact with smart contracts in many Ethereum networks, and Görli is one of them. The deployment of the contract above happens to enable the evaluation of the public information available about it, and how it behaves when called through the Infura API.

The smart contract is deployed in the Görli network with the address 0xbA1dC49D71883c8475Bb3eC16751D1683AB37FC5. Anyone can see the details about it by using the Etherscan website to inspect the blockchain. Figure 5.3 shows the information available about the contract, while Figure 5.4 brings what is available about the transaction that originated it. At this point, no other interaction happened with the contract besides its creation.



Contract 0xbA1dC49D71883c8475Bb3eC16751D1683AB37FC5

Sponsored: Advertise to blockchain developers - sponsored slots available. Book your slot here!

Contract Overview

Balance: 0 Ether

More Info

My Name Tag: Not Available

Contract Creator: 0xfd2a96aea8e2c88... at txn 0xde3fa218c78c5eb...

Transactions | Contract | Events

Latest 1 txn

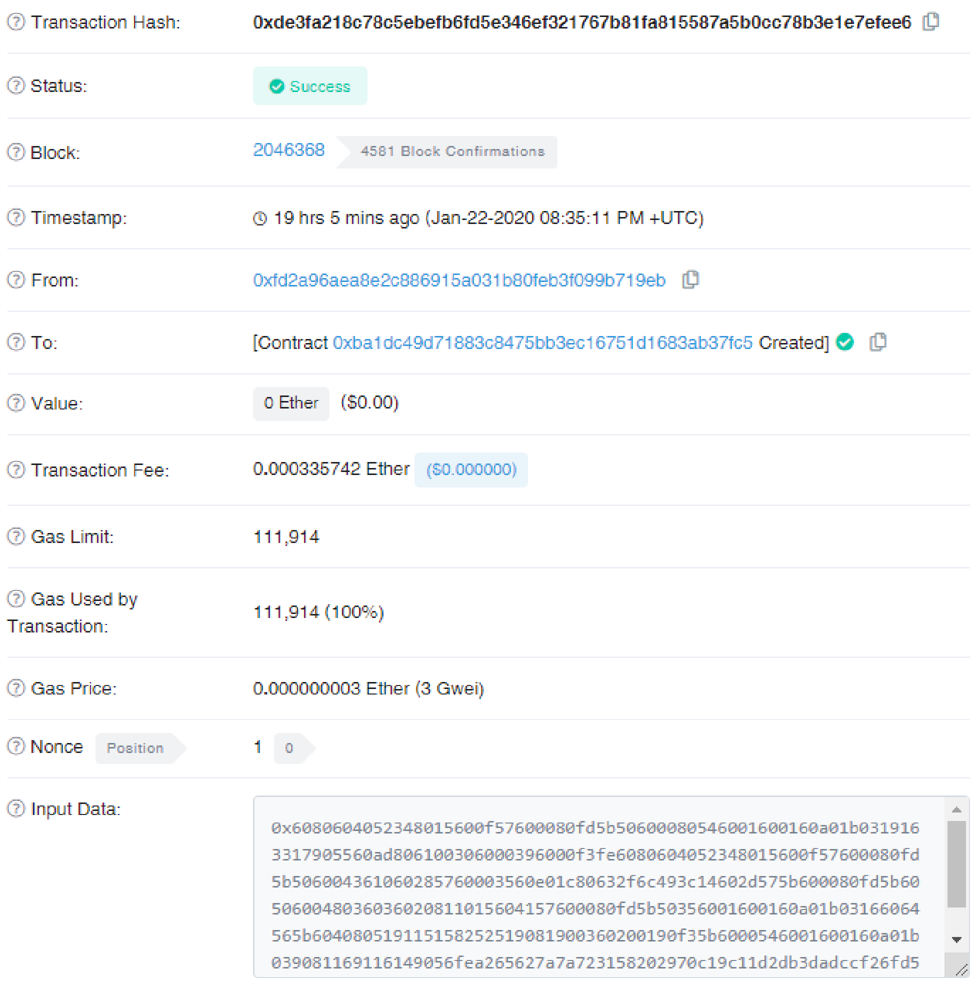
Txn Hash	Block	Age	From		To	Value	[Txn Fee]
0xde3fa218c78c5eb...	2046368	17 hrs 59 mins ago	0xfd2a96aea8e2c88...	IN	Contract Creation	0 Ether	0.000335742

[Download CSV Export]

Figure 5.3 – Contract’s publicly available data

First, it is worth highlighting that to have access to those pages shown, at this point, one must know either the transaction hash that originated the contract, the contract address, or the Host’s account address. At this stage, though, no one else besides the Host has reasons to hold that knowledge, which would make the contract hard to find. On the other hand, by the time the contract is completely written and in use, all those required information can be found or inferred by a variety of means, since all stakeholders interact with it. In conclusion, all the evaluations through Etherscan in this work assumes that any interested party can have access to it. Furthermore, remember that the stakeholders must know their real identities to make the arrangements of a stay. Consequently, the pseudo-anonymity provided by the Ethereum address is compromised. Even if the implementation design were to separate the functionalities in different contracts by stakeholder, eventually, they would require exchanging data between themselves, and it would be equally recorded in the blockchain.

Going back to the data available, the most important information one can learn is the Host's address. In Ethereum, however, no one will be able to use it to interact with the smart contract since the private key – secret key - is also required for that. For that reason, it represents no risk to the application. At this point, that is all to inspect, once the only function available to call is *unlock*, which is called as a query and, therefore, does not leave any records in the blockchain.



Transaction Hash:	0xde3fa218c78c5ebefb6fd5e346ef321767b81fa815587a5b0cc78b3e1e7efee6
Status:	Success
Block:	2046368 4581 Block Confirmations
Timestamp:	19 hrs 5 mins ago (Jan-22-2020 08:35:11 PM +UTC)
From:	0xfd2a96aea8e2c886915a031b80feb3f099b719eb
To:	[Contract 0xba1dc49d71883c8475bb3ec16751d1683ab37fc5 Created]
Value:	0 Ether ($0.00)
Transaction Fee:	0.000335742 Ether ($0.000000)
Gas Limit:	111,914
Gas Used by Transaction:	111,914 (100%)
Gas Price:	0.000000003 Ether (3 Gwei)
Nonce Position	1 0
Input Data:	

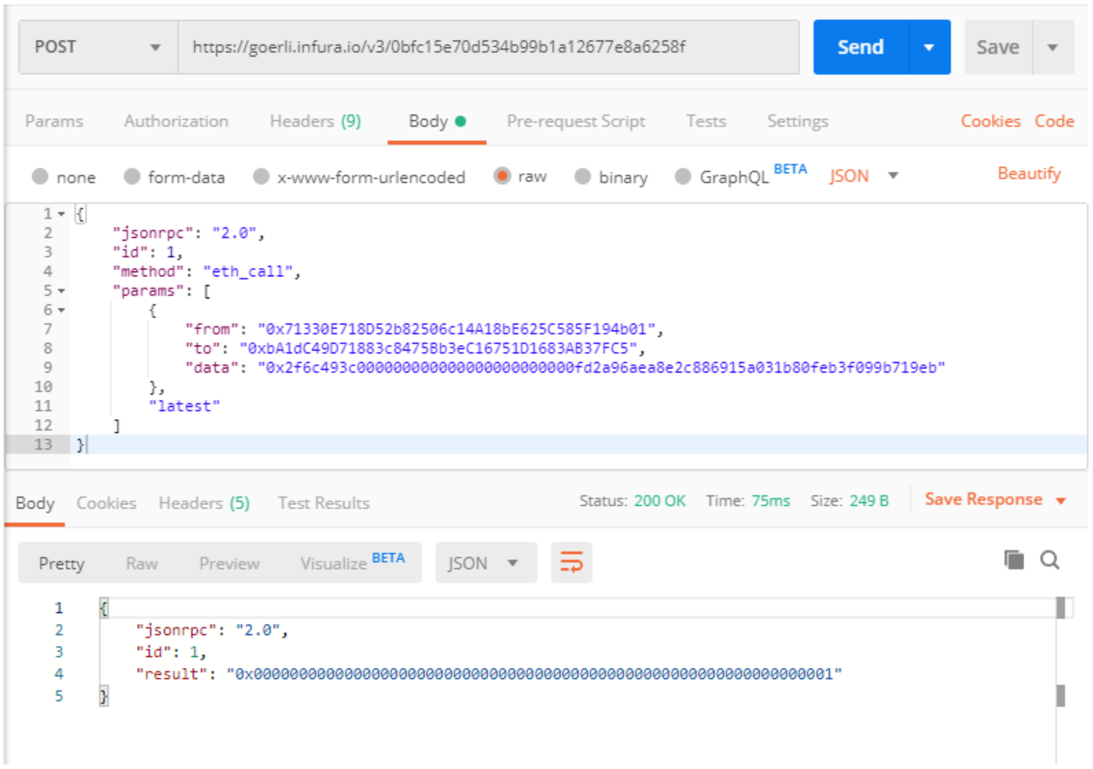
```
0x6080604052348015600f57600080fd5b506000805460016001600160a01b031916
3317905560ad806100306000396000f3fe6080604052348015600f57600080fd
5b5060043610602857600035600e01c80632f6c493c14602d575b600080fd5b60
50600480360360208110156041576000080fd5b50356001600160a01b03166064
565b6040805191151582525190819003602001900f35b6000546001600160a01b
0390811691161490056fea265627a7a723158202970c19c11d2db3dadccf26fd5
```

Figure 5.4 – Contract's deploy transaction - Publicly available data

Finally, the last aspect of evaluating this contract is how it behaves when called from the Infura API. Ideally, the calls to Infura would come from either the lock hardware or the management tool. However, as discussed before, they both communicate with Infura in the same way, through HTTP requests. To test and evaluate the architecture with those HTTP calls during development, applying less effort than building the management interface along with the smart contract would require, Postman is used. As a result, the management tool implementation can be postponed and safely start after the smart contract is fully developed.

Figure 5.5 shows a request from Postman to the Infura API endpoint. It asks Infura to call the *unlock* function of the smart contract deployed on the Görli network, using the Host address as a parameter. The image also shows the response from Infura, where the *result* field means that the unlock function returned true – the address can unlock the door. It is worth highlighting the following:

- The parameter *from* corresponds to which account is making the call. In this case, a random account address is used since anyone can call the function;
- The parameter *to* is which address to call. I.e., the smart contract address at the Görli network;
- The parameter *data* is encoded with the smart contract function to be called and the corresponding parameters – if any. The encoded message can be obtained in Remix. By visually inspecting the call, though, it is possible to identify the Host address – that is the function expected parameter.



```
https://goerli.infura.io/v3/0bfc15e70d534b99b1a12677e8a6258f
```

```
{
    "jsonrpc": "2.0",
    "id": 1,
    "method": "eth_call",
    "params": [
        {
            "from": "0x71330E718D52b82506c14A18bE625C585F194b01",
            "to": "0xbA1dC49D71883c8475Bb3eC16751D1683AB37FC5",
            "data": "0x2f6c493c000000000000000000000000fd2a96aea8e2c886915a031b80feb3f099b719eb"
        },
        "latest"
    ]
}
```

Status: 200 OK Time: 75ms Size: 249 B

```
{
    "jsonrpc": "2.0",
    "id": 1,
    "result": "0x0000000000000000000000000000000000000000000000000000000000000001"
}
```

Figure 5.5 – Infura API request and response to the contract *unlock* function

To continue the development is extremely important to bear in mind that to make a query call to the smart contract does not require signing the message with the account's secret key. In other words, anyone can use that address to query the function. That is not the case for transactions, however. No one can submit a transaction without signing it with the account's private key. Of course, this function could be written to require a transaction call, but that would bring two issues to the architecture. First, transactions are paid, meaning that the lock hardware and the management interface would have to pay every time that information was needed, which appears to be unreasonable. Second, the network must process those transactions, which can take a while. It does not seem reasonable to impose this wait at someone that is at the door waiting to get in.

In conclusion, it is better to work with the query constraint in mind than paying and waiting for transactions.

5.3.2 Add support for Guests

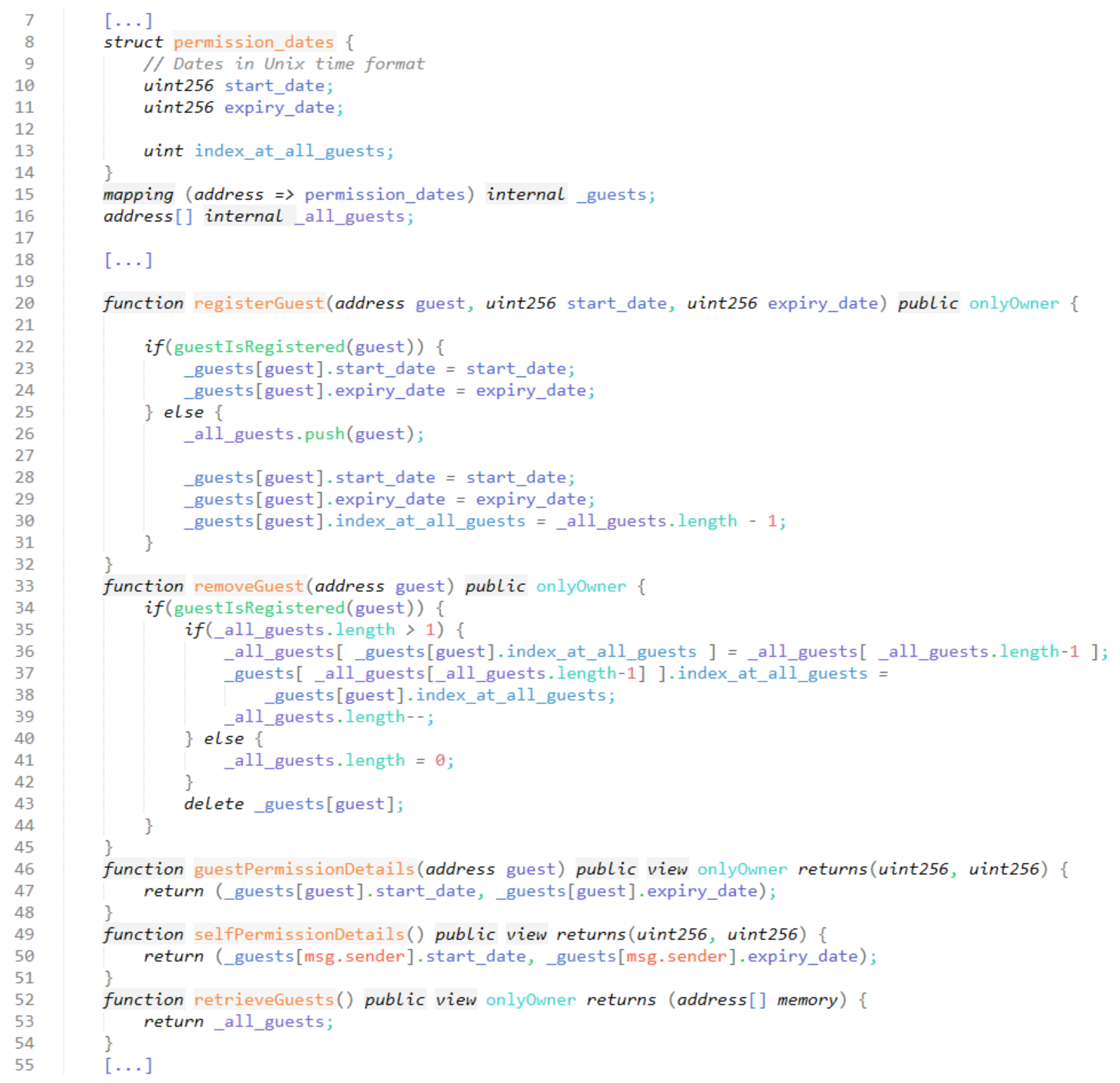
Once Managers are fundamentally a special case of the Host, more precisely a constrained one, it makes sense to implement first all the use cases considering only the existence of Host and Guests. For that reason, the next increment builds the necessary functionalities to begin supporting Guests. More specifically, they correspond to the use cases: (i) UC2 – Register Guest; (ii) UC3 – Remove Guest; (iii) UC4 - Retrieve Guests; (iv) UC11 – Verify Permission; (v) UC1 – Unlock and lock the door.

Similar to section 5.3.1, following are some significant highlights about the smart contract implementation partially shown by Figure 5.6:

- The smart contract requires the Ethereum account address of anyone to be registered as a Guest. It demands, therefore, that Host and Guest exchange that information outside the application, e.g., through Airbnb after booking the place;
- The functions *registerGuest()* and *removeGuest()* alter the data in the smart contract, which means they require a transaction to be called. The modifier *onlyOwner* included at the functions' declaration enforces that, in order to call them, one must be the registered owner of the smart contract. If anyone else tries to use them, none of the code insides is executed. Furthermore, the Guest is only added, or removed, after the blockchain process the transaction, as explained before;

- In Solidity, dates are represented in the Unix Epoch time (Wikipedia, 2020), which is the number of seconds that have elapsed since 00:00:00 UTC of 1 January 1970. For instance, the Unix Epoch time for 09:30:00 GMT of 30 January 2020 is 1580376600;
- In solidity, the most efficient data type to record relationships between entities is a mapping, which works similarly as a key-value dictionary. However, unlike dictionaries of popular object-oriented languages, Solidity's mapping does not allow iteration through its keys. In other words, one must know the key to access its corresponding value. Therefore, a second data type, an array, is demanded to allow for Guests retrieval, i.e., keep track of the mapping keys;
- The function *retrieveGuests()* returns the addresses of all Guests registered in the mapping. Once smart contract functions do not return structs – the data structure used to record Guests' permission dates -, the management interface must call *guestPermissionDetails()* for each Guest address retrieved to meet the UC4 scenario correctly. Furthermore, *retrieveGuests()* includes past Guests with expired permissions. Again, to meet the UC4 scenario, the management interface must use the permission details to hide past Guests from the user.

The UC4 specifies in the second step of the main success scenario that the system verifies that the retrieve request comes from either the Host or a Manager and, if that is not the case, the system should ignore the request, i.e., not respond. However, as shown in the previous section 5.3.1, query calls to smart contract functions - as it is the case for *retireveGuests()* – do not require signing the message with the accounts' private key. In other terms, anyone could make a query call defining which address it should be made from. Nevertheless, the transactions to register Guests are open in the blockchain with the permission details –more details are following -, and any interested party could derive a Guest list from the smart contract calls. In any case, both *retireveGuests()* and *guestPermissionDetails()* functions have the modifier that constraints who can call them for two reasons: (i) to avoid casual users of retrieving the Guest list and acquiring other Guests' permission details through the management interface – which is assumed to be honest and do not allow calls from accounts not owned by the person; (ii) to make it more complex to acquire that information. As discussed before regarding the host address, the fact that Guest addresses and their permission details are public, it does not represent a significant threat due to the private key required to use those accounts to interact with the system.



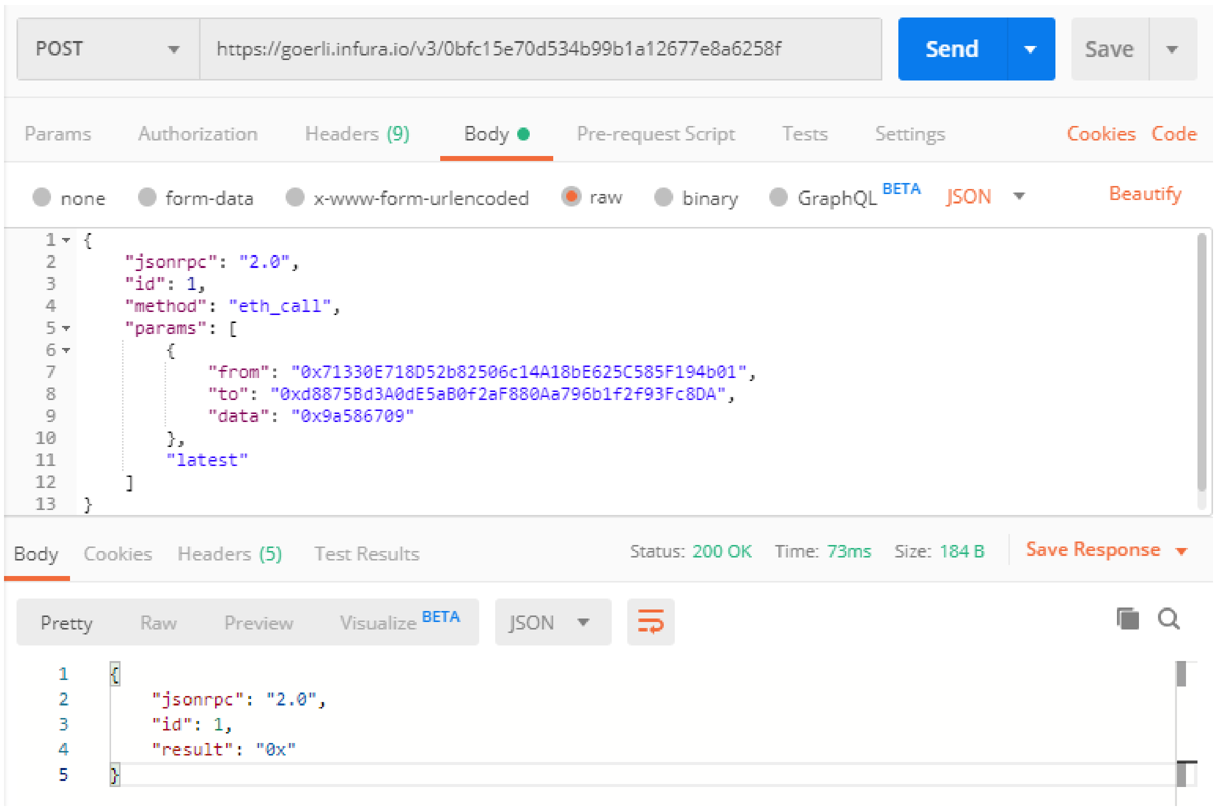
```solidity
[...]
struct permission_dates {
    // Dates in Unix time format
    uint256 start_date;
    uint256 expiry_date;

    uint index_at_all_guests;
}
mapping (address => permission_dates) internal _guests;
address[] internal _all_guests;

[...]

function registerGuest(address guest, uint256 start_date, uint256 expiry_date) public onlyOwner {

    if(guestIsRegistered(guest)) {
        _guests[guest].start_date = start_date;
        _guests[guest].expiry_date = expiry_date;
    } else {
        _all_guests.push(guest);

        _guests[guest].start_date = start_date;
        _guests[guest].expiry_date = expiry_date;
        _guests[guest].index_at_all_guests = _all_guests.length - 1;
    }
}
function removeGuest(address guest) public onlyOwner {
    if(guestIsRegistered(guest)) {
        if(_all_guests.length > 1) {
            _all_guests[ _guests[guest].index_at_all_guests ] = _all_guests[ _all_guests.length-1 ];
            _guests[ _all_guests[_all_guests.length-1] ].index_at_all_guests =
                _guests[guest].index_at_all_guests;
            _all_guests.length--;
        } else {
            _all_guests.length = 0;
        }
        delete _guests[guest];
    }
}
function guestPermissionDetails(address guest) public view onlyOwner returns(uint256, uint256) {
    return (_guests[guest].start_date, _guests[guest].expiry_date);
}
function selfPermissionDetails() public view returns(uint256, uint256) {
    return (_guests[msg.sender].start_date, _guests[msg.sender].expiry_date);
}
function retrieveGuests() public view onlyOwner returns (address[] memory) {
    return _all_guests;
}
[...]
```

Figure 5.6 – Smart Contract Snippet – some Guests related functionalities

The following experiment is executed - at the given order - to illustrate the query call issue better:

- Using Remix and Metamask, the smart contract is deployed in the Görli network at the address 0xd8875Bd3A0dE5aB0f2aF880Aa796b1f2f93Fc8DA by the account 0xFd2a96AEA8E2C886915a031b80FEb3f099b719eb;

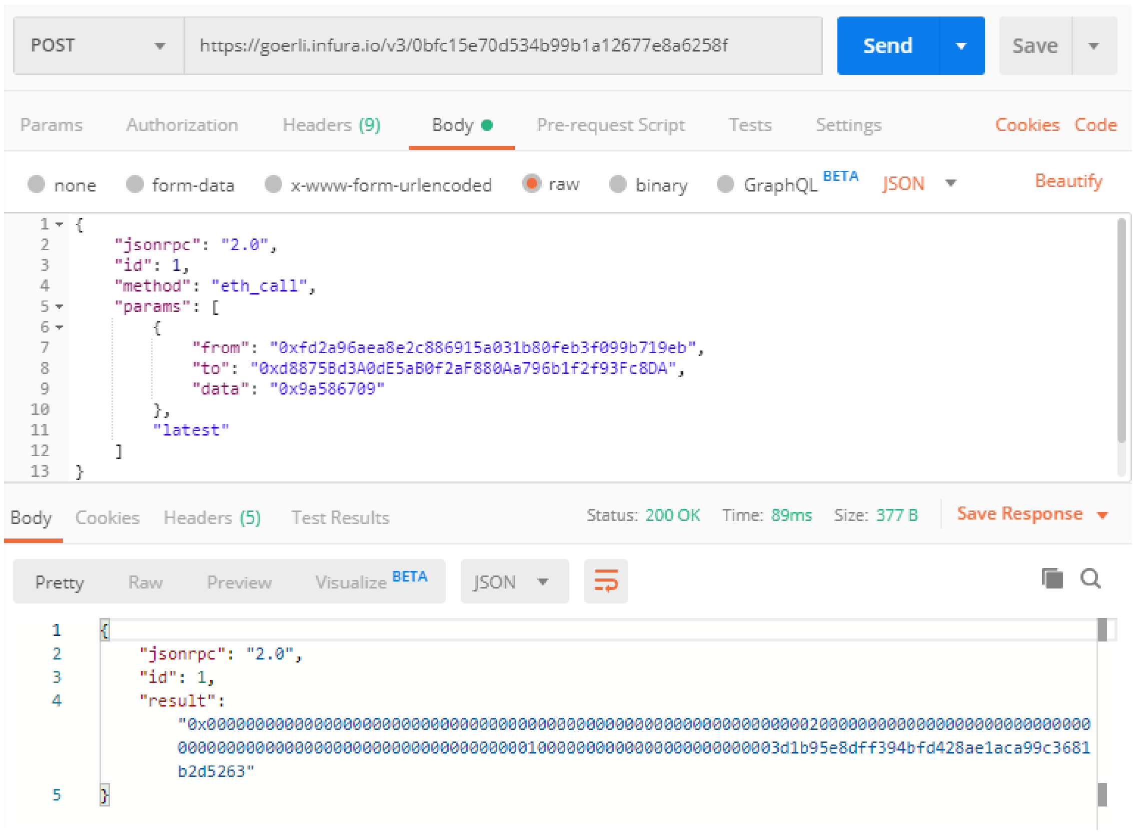
- The Host account registers the address 0x3d1b95E8Dff394bfD428ae1aCA99C3681B2d5263 as Guest, with start date as 1580376600 and expiry date as 2590376699;
- Using Postman to call the Infura API, a request is made to the function *retrieveGuests()* using a random address – which simulates the honest call made from the management interface. The request and the response can be found in Figure 5.7, which resulted in a failed call as expected;
- The last step was repeated but using the Host address. The request and the response can be found in Figure 5.8, which resulted in a successful call containing the registered Guest's address;



```
{
    "jsonrpc": "2.0",
    "id": 1,
    "method": "eth_call",
    "params": [
        {
            "from": "0x71330E718D52b82506c14A18bE625C585F194b01",
            "to": "0xd8875Bd3A0dE5aB0f2aF880Aa796b1f2f93Fc8DA",
            "data": "0x9a586709"
        },
        "latest"
    ]
}
```

```
{
    "jsonrpc": "2.0",
    "id": 1,
    "result": "0x"
}
```

Figure 5.7 – Infura API request and response to *retrieveGuests* function – call from a random account



https://goerli.infura.io/v3/0bfc15e70d534b99b1a12677e8a6258f

```
{
    "jsonrpc": "2.0",
    "id": 1,
    "method": "eth_call",
    "params": [
        {
            "from": "0xfd2a96aea8e2c886915a031b80feb3f099b719eb",
            "to": "0xd88758d3A0dE5aB0f2aF880Aa796b1f2f93Fc8DA",
            "data": "0x9a586709"
        },
        "latest"
    ]
}
```

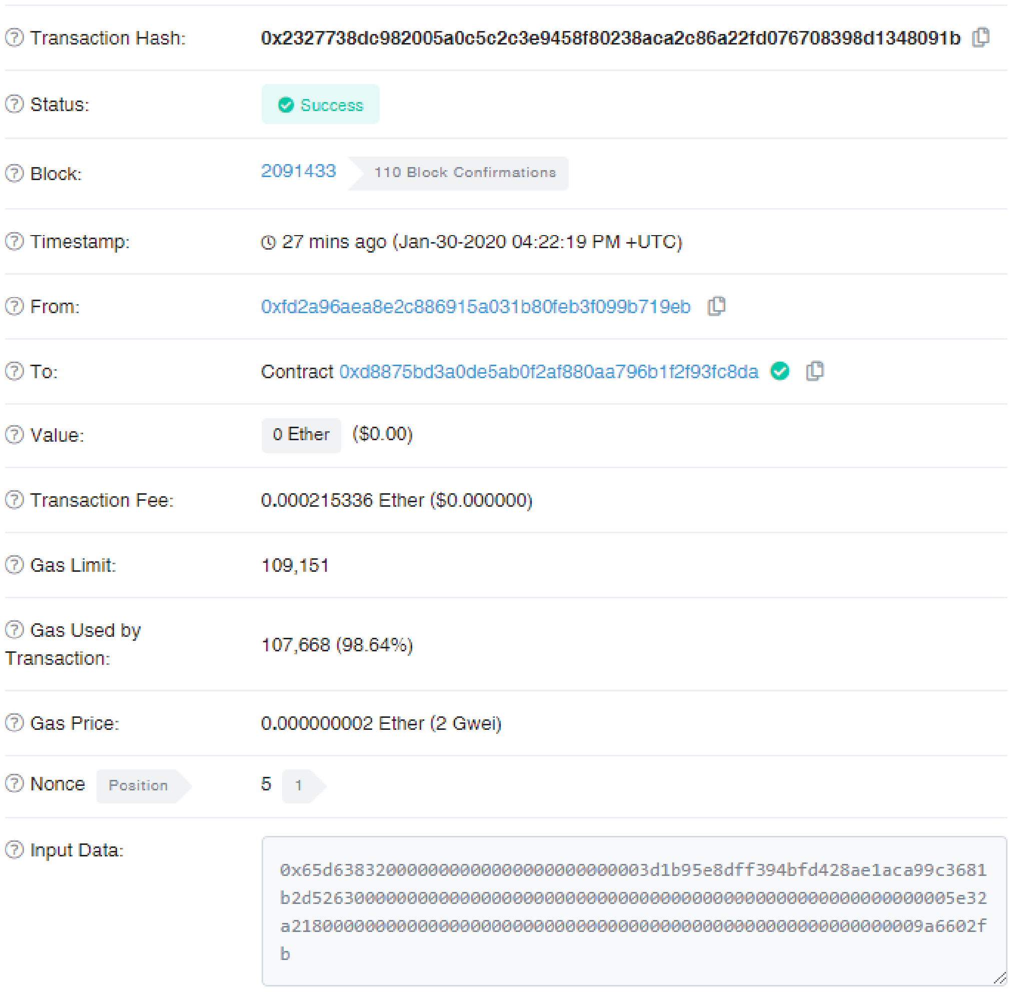
Status: 200 OK Time: 89ms Size: 377 B

```
{
    "jsonrpc": "2.0",
    "id": 1,
    "result":
        "0x00000000000000000000000000000000000000000000000000000000000000200000000000000000000000000000
        00000000000000000000000000000000000010000000000000000000000003d1b95e8dff394bfd428ae1aca99c3681
        b2d5263"
}
```

Figure 5.8 – Infura API request and response to *retrieveGuests* function – call from Host account

Next, the Host account removes the Guest previously added. The reason is to investigate the publicly available information related to registering and removing a Guest. Once again, Etherscan is used to inspect the smart contract address, which displays all the transactions in chronological order. Figure 5.9 shows the transaction details of registering a Guest. Note that the input data field has four hexadecimal numbers encoded between zeros, and it contains all the permission information as following:

- 0x65d63832: This number encodes the function to call - in this case, *registerGuest*. Each function in the smart contract has its identifier, that can be easily obtained from different sources. Remix, for instance, is one of them;
- 3d1b95e8dff394bfd428ae1aca99c3681b2d5263: The Guest address;
- 5e32a218: Hexadecimal representation of number 1580376600;
- 9a6602fb: Hexadecimal representation of number 2590376699.



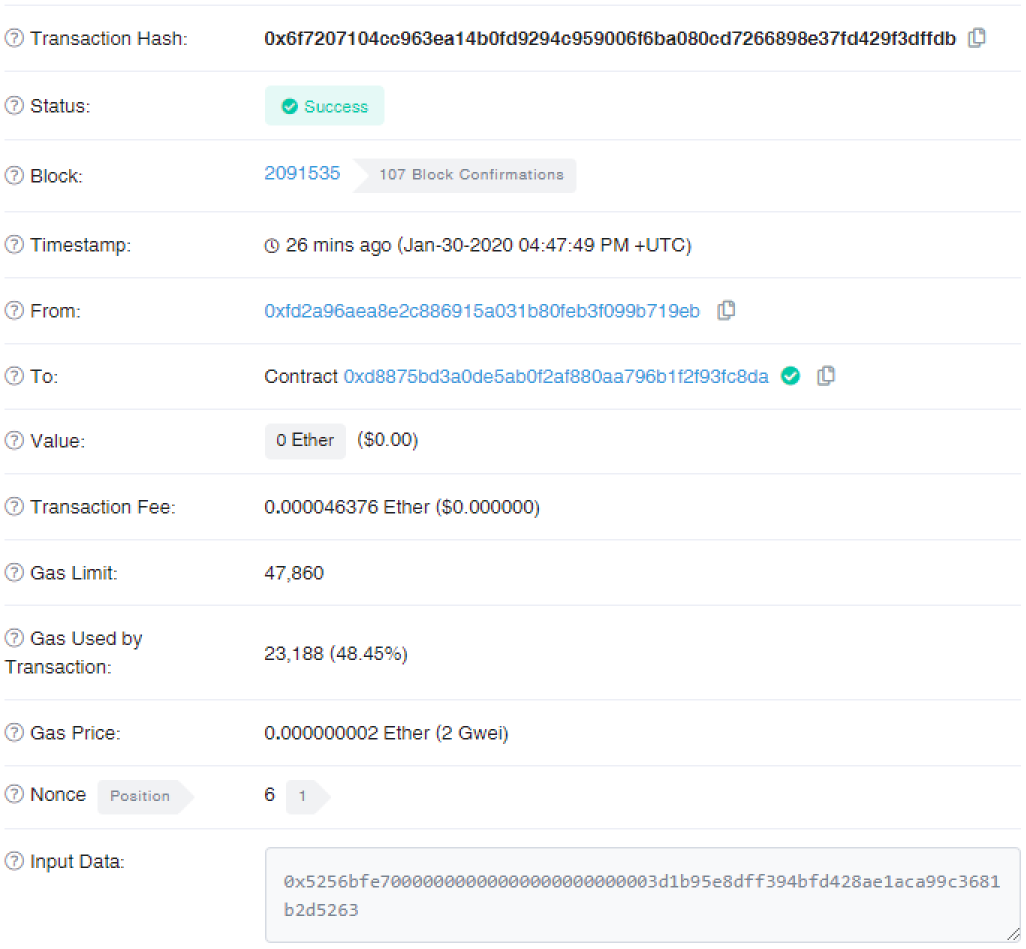
Transaction Hash:	0x2327738dc982005a0c5c2c3e9458f80238aca2c86a22fd076708398d1348091b
Status:	Success
Block:	2091433 110 Block Confirmations
Timestamp:	27 mins ago (Jan-30-2020 04:22:19 PM +UTC)
From:	0xfd2a96aea8e2c886915a031b80feb3f099b719eb
To:	Contract 0xd8875bd3a0de5ab0f2af880aa796b1f2f93fc8da
Value:	0 Ether ($0.00)
Transaction Fee:	0.000215336 Ether ($0.000000)
Gas Limit:	109,151
Gas Used by Transaction:	107,668 (98.64%)
Gas Price:	0.000000002 Ether (2 Gwei)
Nonce Position	5 1
Input Data:	

```
0x65d638320000000000000000000000003d1b95e8dff394bfd428ae1aca99c3681
b2d526300000000000000000000000000000000000000000000000000000000005e32
a21800000000000000000000000000000000000000000000000000000000009a6602f
b
```

Figure 5.9 – Publicly available transaction details to register a Guest

The transaction details of removing a Guest are in Figure 5.10. Observe that the input data field reveals the *removeGuest* function call, encoded as 0x5256bfe7, for the Guest 3d1b95e8dff394bfd428ae1aca99c3681b2d5263, analogously to what happened for *registerGuest*.

In conclusion, all data sent to the smart contract is open to the public. Although not ideal, the knowledge of the Host, Guests, and eventually Managers accounts does not allow unauthorized manipulations as, for example, register and remove Guests or unlocking the door – assuming that the lock hardware follows the architecture guidelines. The knowledge obtained from the transactions also does not enable people to monitor or infer if someone is currently at the place or not. The query calls to verify permissions from the lock hardware are not recorded in the blockchain and, even if someone knows that there is no Guest registered at a given time, Host and Managers are still capable of unlocking the door.



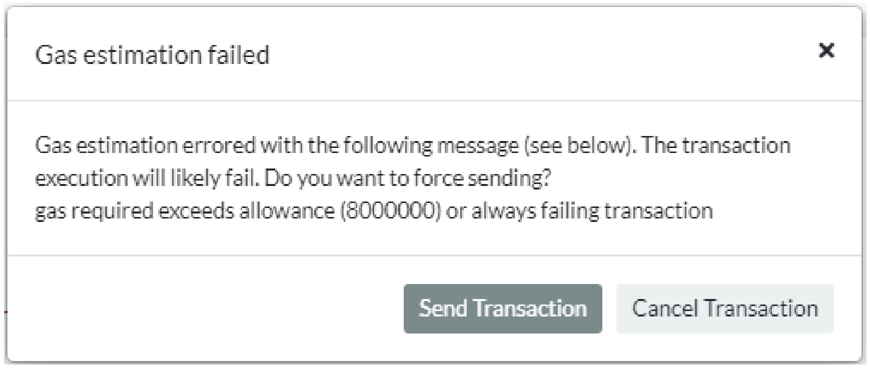
Transaction Hash:	0x6f7207104cc963ea14b0fd9294c959006f6ba080cd7266898e37fd429f3dffdb
Status:	Success
Block:	2091535 107 Block Confirmations
Timestamp:	26 mins ago (Jan-30-2020 04:47:49 PM +UTC)
From:	0xfd2a96aea8e2c886915a031b80feb3f099b719eb
To:	Contract 0xd8875bd3a0de5ab0f2af880aa796b1f2f93fc8da
Value:	0 Ether ($0.00)
Transaction Fee:	0.000046376 Ether ($0.000000)
Gas Limit:	47,860
Gas Used by Transaction:	23,188 (48.45%)
Gas Price:	0.000000002 Ether (2 Gwei)
Nonce Position	6 1
Input Data:	0x5256bfe70000000000000000000000003d1b95e8dff394bfd428ae1aca99c3681b2d5263

Figure 5.10 – Publicly available transaction details to remove a Guest

5.3.3 Exclusive permission feature

The next increment builds the exclusive permission feature. More specifically, it covers the use cases that follow: (i) UC8 – Turn on exclusive permission; (ii) UC9 – Turn off exclusive permission; (iii) UC10 – Check exclusive permission. However, this feature also has an impact on many other use cases, as described by their main success scenarios and alternative paths. Therefore, reviewing the following use cases, partially implemented before, is necessary: (i) UC1 – Unlock and lock the door; (ii) UC3 – Remove Guest; (iii) UC11 – Verify Permission.

As the implementation leverages the same structure and elements used for coding the previous versions, it is not necessary to discuss the functions in depth. However, it is worth mentioning a new approach used to enforce functions' preconditions. Figure 5.6 shows the *removeGuest* function starting with an *if* condition to check if the Guest is registered. Otherwise, it does not make sense to remove it. However, in those cases where the Guest was not registered, the Host would still be able to make a successful call to the function, useless but successful. The danger is that a typo could give the Host the wrong impression that the Guest's permission was removed when, in fact, it was still there. This behavior was changed using the *require* function, which forces a transaction to fail if its condition evaluates to false. Going back to the Guest remove example, if the Host tries to remove an inexistent Guest, the transaction will fail, and they will know that something went wrong. The Metamask wallet can inform users that the transaction at hand is likely going to fail even before they approve and send it. As an example, Figure 5.11 displays the warning gave by Metamask when the Host tries to call the *removeGuest* function to remove an inexistent address. Assumes that the Host ignores this warning, sends the transaction, and it fails. Using Etherscan to inspect the transaction reveals the custom message written in the *require* function that explains the error, which in this case is "*The Guest is not registered*" – see Figure 5.12.

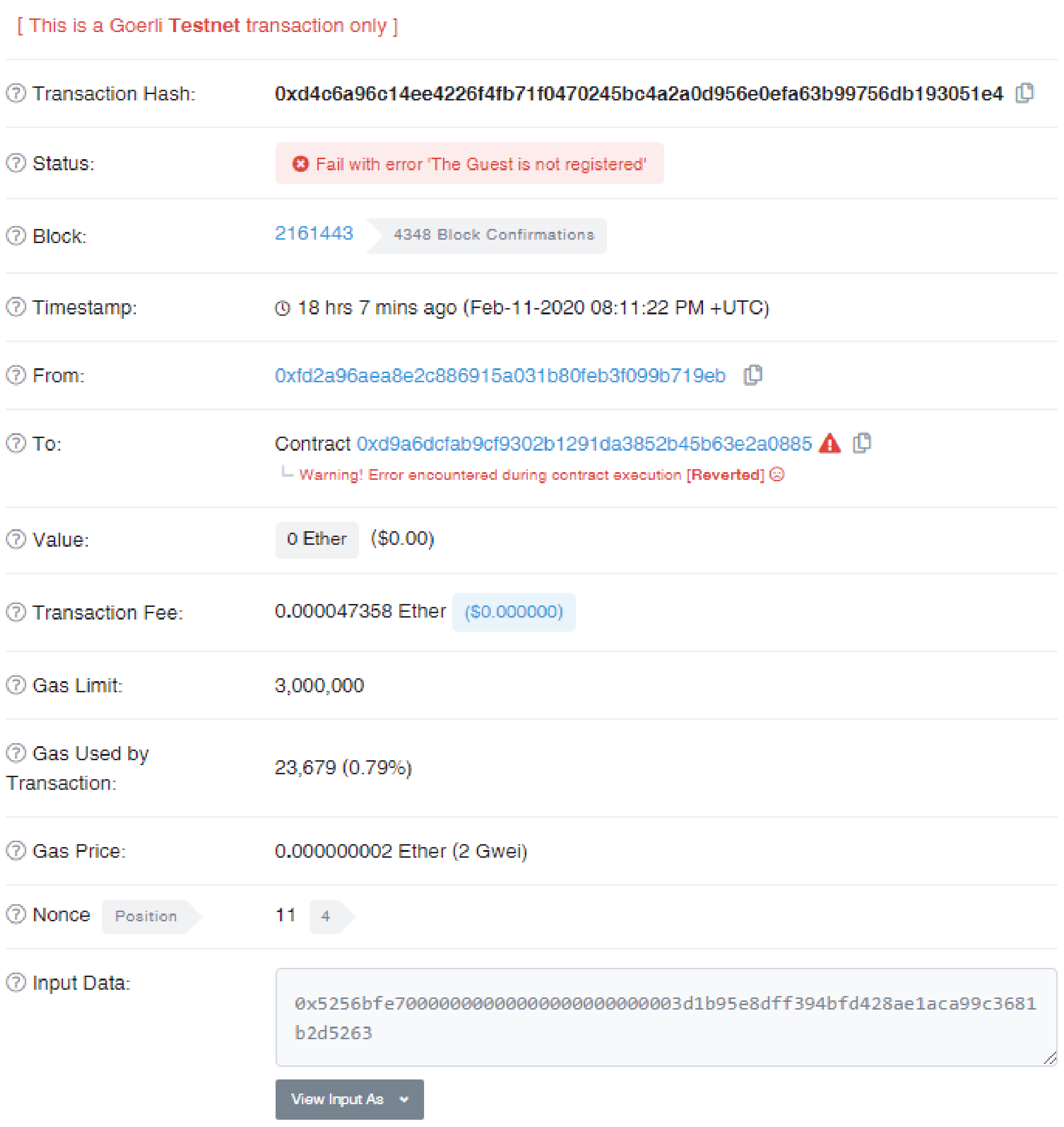

Gas estimation failed

Gas estimation errored with the following message (see below). The transaction execution will likely fail. Do you want to force sending?
gas required exceeds allowance (8000000) or always failing transaction

Figure 5.11 – Metamask warning before sending a transaction that does not meet an expected precondition

Figure 5.13 presents three functions using this strategy to enforce pre-conditions - e.g., turnExclusiveFeatureOn() will reject a transaction if the feature is already on.

One more relevant characteristic to emphasize is that details about the exclusive permission feature – the address that holds the permission and when it expires – are available in the blockchain. As demonstrated before, although the functions responsible for retrieving the information only answer calls from the host address, advanced users can emulate that call, e.g., using Infura API, or find the information using block explorers to find and view transactions details. Despite that, this knowledge does not enable anyone to illegally control the smart contract and, therefore, the smart lock.



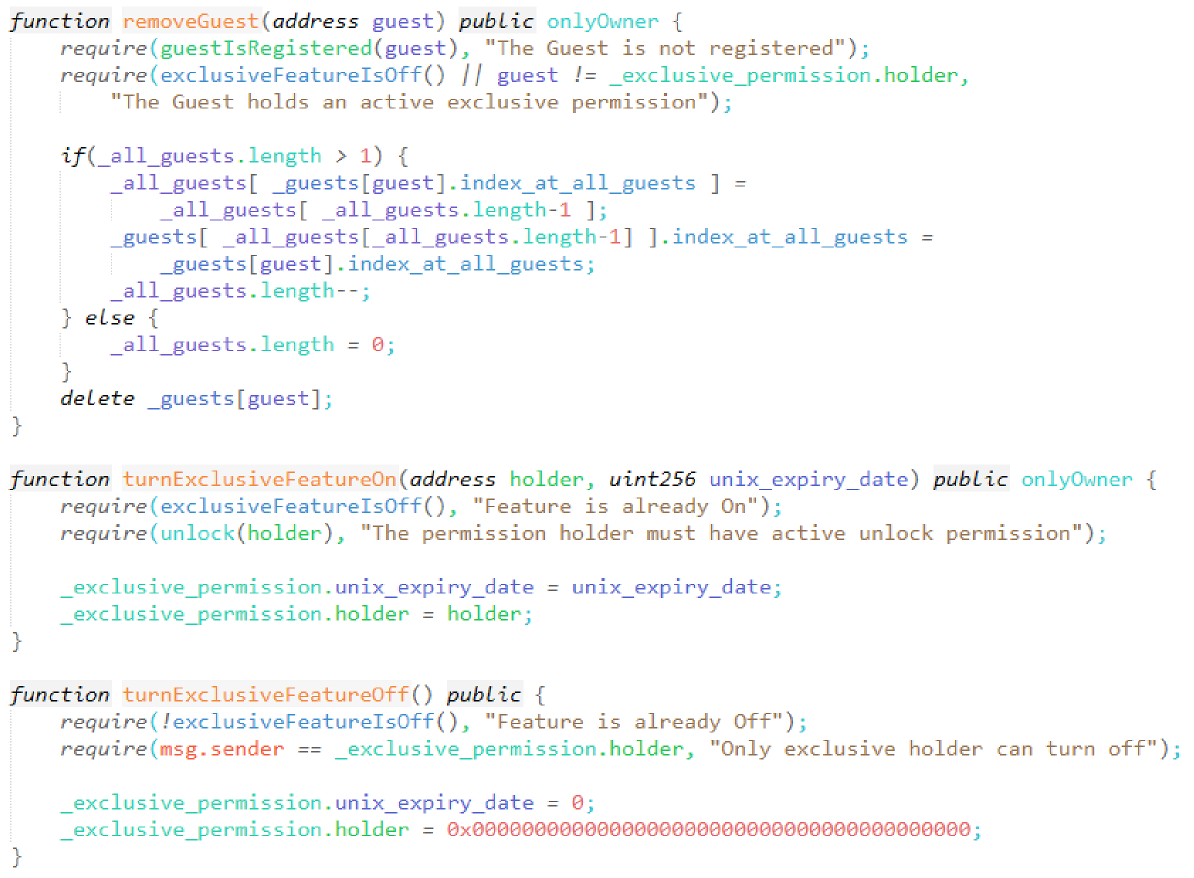
[This is a Goerli **Testnet** transaction only]

Transaction Hash:	0xd4c6a96c14ee4226f4fb71f0470245bc4a2a0d956e0efa63b99756db193051e4
Status:	Fail with error 'The Guest is not registered'
Block:	2161443 4348 Block Confirmations
Timestamp:	18 hrs 7 mins ago (Feb-11-2020 08:11:22 PM +UTC)
From:	0xfd2a96aea8e2c886915a031b80feb3f099b719eb
To:	Contract 0xd9a6dcfab9cf9302b1291da3852b45b63e2a0885 Warning! Error encountered during contract execution **[Reverted]**
Value:	0 Ether ($0.00)
Transaction Fee:	0.000047358 Ether ($0.000000)
Gas Limit:	3,000,000
Gas Used by Transaction:	23,679 (0.79%)
Gas Price:	0.000000002 Ether (2 Gwei)
Nonce Position	11 4
Input Data:	0x5256bfe70000000000000000000000003d1b95e8dff394bfd428ae1aca99c3681b2d5263

Figure 5.12 – Host removes inexistent Guest - Failed transaction details

5.3.4 Add support for Managers

The next increment includes support for Managers, which represent a Host with some functionality constraints. Therefore, it implements the following use cases: (i) UC5 – Register Manager; (ii) UC6 – Remove Manager; (iii) UC7 – Retrieve Managers. In addition to that, all the use cases that contain the Manager as one of the actors must be revisited. In other words, all the use cases besides UC11 – Verify permission.



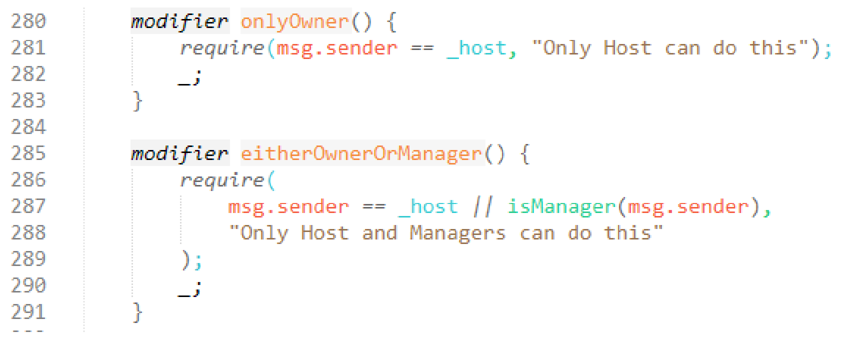
```
function removeGuest(address guest) public onlyOwner {
    require(guestIsRegistered(guest), "The Guest is not registered");
    require(exclusiveFeatureIsOff() || guest != _exclusive_permission.holder,
        "The Guest holds an active exclusive permission");

    if(_all_guests.length > 1) {
        _all_guests[ _guests[guest].index_at_all_guests ] =
            _all_guests[ _all_guests.length-1 ];
        _guests[ _all_guests[_all_guests.length-1] ].index_at_all_guests =
            _guests[guest].index_at_all_guests;
        _all_guests.length--;
    } else {
        _all_guests.length = 0;
    }
    delete _guests[guest];
}

function turnExclusiveFeatureOn(address holder, uint256 unix_expiry_date) public onlyOwner {
    require(exclusiveFeatureIsOff(), "Feature is already On");
    require(unlock(holder), "The permission holder must have active unlock permission");

    _exclusive_permission.unix_expiry_date = unix_expiry_date;
    _exclusive_permission.holder = holder;
}

function turnExclusiveFeatureOff() public {
    require(!exclusiveFeatureIsOff(), "Feature is already Off");
    require(msg.sender == _exclusive_permission.holder, "Only exclusive holder can turn off");

    _exclusive_permission.unix_expiry_date = 0;
    _exclusive_permission.holder = 0x0000000000000000000000000000000000000000;
}
```

Figure 5.13 – Smart contract functions using *require()* to enforce preconditions

The implementation of Managers follows precisely the same data structure strategy used for Guests, which is a combination of a mapping variable with an array of Manager addresses. As a consequence of that, the operations of registering, removing, and retrieving Managers are highly similar to those of Guests, which implicates that all the discussions in section 5.3.2 - Add support for Guests – related to those actions are equally applicable for Managers. For instance, inspecting the blockchain record of transactions reveals Manager addresses registered as it does for Guests. Also, it is possible to successfully call the function to retrieve the list of Managers without the need to sign the request, as shown for Guests previously.

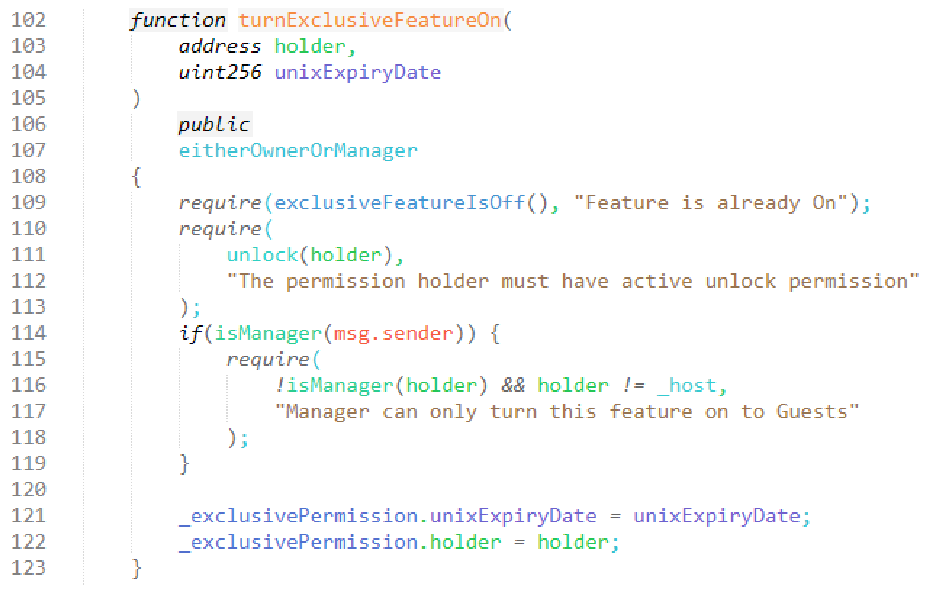
Besides the introduction of the data structure and of those operations to control Managers, use cases already implemented must accommodate the newly introduced actor. Note from the use cases, however, that when they allow both Manager and Host to act, they expect an identical behavior from both actors – there is a single exception to that, UC8 Turn on exclusive permission. It means that most of the changes necessary to support Managers are simply changing the functions' custom access modifier from *onlyOwner* to the new *eitherOwnerOrManager* - see

Figure 5.14. Regarding UC8, where the Manager has a limitation in functionality compared to the Host, the modification besides changing the modifier was checking for whom the Manager tries to turn the feature on – compare Figure 5.15 with Figure 5.13 to see the difference. As a result of those minimal changes, the discussion presented in previous sections of this chapter remain valid and can be applied to Managers.



```
modifier onlyOwner() {
    require(msg.sender == _host, "Only Host can do this");
    _;
}

modifier eitherOwnerOrManager() {
    require(
        msg.sender == _host || isManager(msg.sender),
        "Only Host and Managers can do this"
    );
    _;
}
```

Figure 5.14 – *onlyOwner* and *eitherOwnerOrManager* access modifiers implementation



```
function turnExclusiveFeatureOn(
    address holder,
    uint256 unixExpiryDate
)
    public
    eitherOwnerOrManager
{
    require(exclusiveFeatureIsOff(), "Feature is already On");
    require(
        unlock(holder),
        "The permission holder must have active unlock permission"
    );
    if(isManager(msg.sender)) {
        require(
            !isManager(holder) && holder != _host,
            "Manager can only turn this feature on to Guests"
        );
    }

    _exclusivePermission.unixExpiryDate = unixExpiryDate;
    _exclusivePermission.holder = holder;
}
```

Figure 5.15 – *turnExclusiveFeatureOn* function implementation

5.3.5 Add events

At this stage, the smart contract implements all the use cases proposed. However, it works as a single way of communication. For instance, when the management tool wants to interact with the smart contract to retrieve information or send transactions, it reaches the smart contract with the desired action. The smart contract, on the other hand, does not have the means to reach external components and update them on some changes that have happened. For example, say that Bob, a Guest holding the exclusive permission feature, decides to turn off its exclusive access. This modification is relevant to Alice, the Host, but unless she checks the exclusive feature status with the smart contract every other time, she would not know about it. A second illustration for this behavior has Alice, the Host, registering a new Guest through the management interface. She approves the transaction in her wallet, and a few moments later, she receives the confirmation of the transaction's successful execution. For Alice to check if the Guest was registered correctly, she would need to manually ask for it, which works but is not user-friendly. If a change in the Guests' list could be notified to the management tool, it could automatically fetch the updated information. Fortunately, Ethereum offers smart contract events to address this concern.

Ethereum events work as a publish-subscribe design pattern. Custom events are declared inside the smart contract and are published – emitted - when some desired condition is met – e.g., when a function executes. External clients can subscribe to listen to particular events of specific contracts. As a consequence, any external stakeholder can be notified by the smart contract when something of their interest happens. Events can include tailored data, but it is crucial to understand that, similarly to what happens with transactions, they are kept open in the blockchain to anyone to see. In other words, there is not a mechanism to protect who can subscribe to a particular event. Another significant characteristic of events is that only transactions can emit them since they have a cost. Free query calls to functions that retrieve information cannot use events - e.g., get the Guests' list.

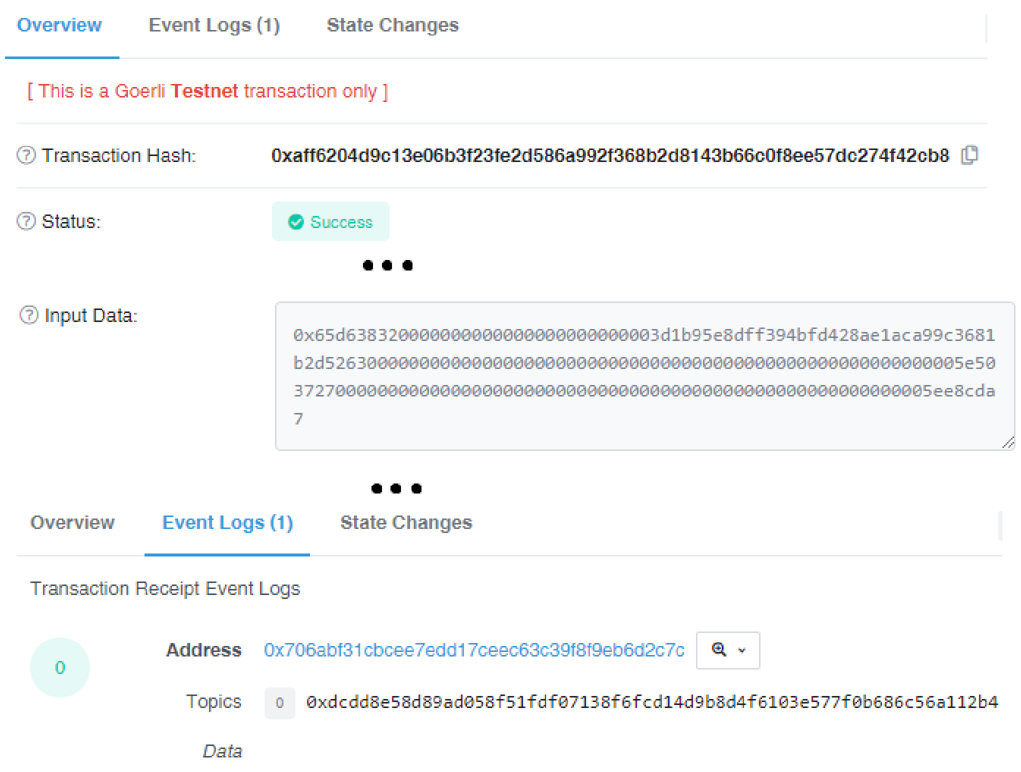
In summary, while events are convenient to enhance the user experience and the applications using Ethereum smart contracts, it brings limitations in terms of privacy and costs. For those reasons, the system's smart contract implements only three events, and none of them publishes additional information. Those events are:

- GuestChange: Emitted every time a Guest is registered or removed;
- ManagerChange: Emitted every time a Manager is registered or removed;
- ExclusiveFeatureChange: Emitted every time the exclusive permission feature is either turned on or off.

Going back to the lock's management interface example, now it can subscribe to those events and retrieve the related data automatically when they happen, they do not require any action from the user. It provides the desired functionality without exposing too much information about the smart lock management.

Using Etherscan, it is possible to see the events emitted by a particular transaction. Figure 5.16 shows the event log for the register Guest transaction 0xaff6204d9c13e06b3f23fe2d586a992f368b2d8143b66c0f8ee57dc274f42cb8. As usual, the topic is encoded but corresponds to the GuestChange event. Note that this topic matches the one shown by the event log in Figure 5.17, which corresponds to the remove Guest transaction 0xa2eb4ff8b541e6b89a10dd083847e4de11c6cfcab0ac2d13050bb3e5bac93fb5. Also, observe that both events have the data field empty.

Finally, the smart contract implementation is complete and Figure 5.18 shows the contract's UML class diagram representation.



[This is a Goerli **Testnet** transaction only]

Transaction Hash: 0xaff6204d9c13e06b3f23fe2d586a992f368b2d8143b66c0f8ee57dc274f42cb8

Status: Success

• • •

Input Data:

```
0x65d63832000000000000000000000000003d1b95e8dff394bfd428ae1aca99c3681
b2d526300000000000000000000000000000000000000000000000000000000005e50
3727000000000000000000000000000000000000000000000000000000005ee8cda
7
```

• • •

Transaction Receipt Event Logs

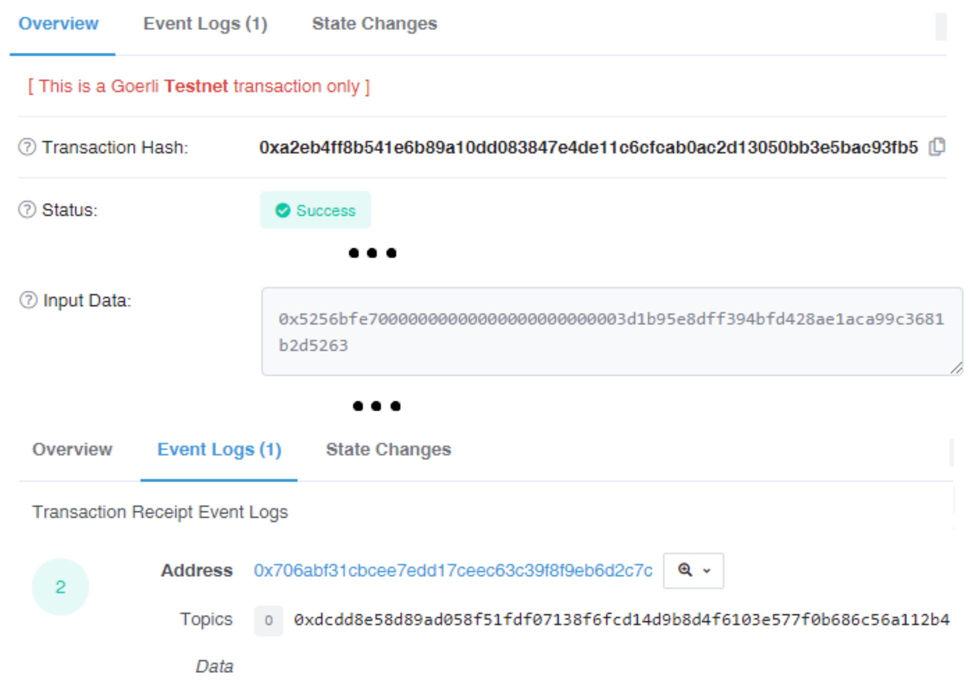
0

Address 0x706abf31cbcee7edd17ceec63c39f8f9eb6d2c7c

Topics 0 `0xdcdd8e58d89ad058f51fdf07138f6fcd14d9b8d4f6103e577f0b686c56a112b4`

Data

Figure 5.16 – Event log for a register guest transaction



Overview Event Logs (1) State Changes

[This is a Goerli **Testnet** transaction only]

Transaction Hash: 0xa2eb4ff8b541e6b89a10dd083847e4de11c6cfcab0ac2d13050bb3e5bac93fb5

Status: Success

• • •

Input Data:

```
0x5256bfe70000000000000000000000003d1b95e8dff394bfd428ae1aca99c3681
b2d5263
```

• • •

Overview Event Logs (1) State Changes

Transaction Receipt Event Logs

2 **Address** 0x706abf31cbcee7edd17ceec63c39f8f9eb6d2c7c

Topics 0 0xdcdd8e58d89ad058f51fdf07138f6fcd14d9b8d4f6103e577f0b686c56a112b4

Data

Figure 5.17 – Event log for a remove guest transaction

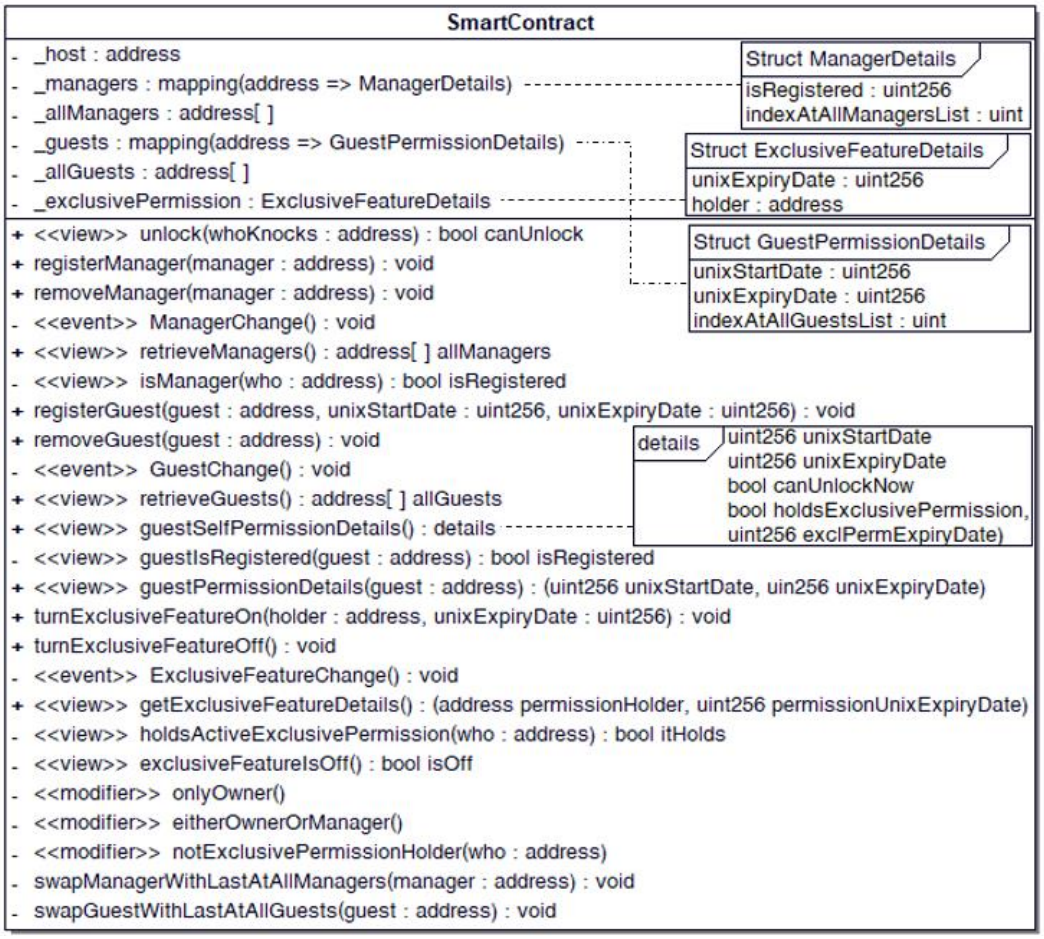
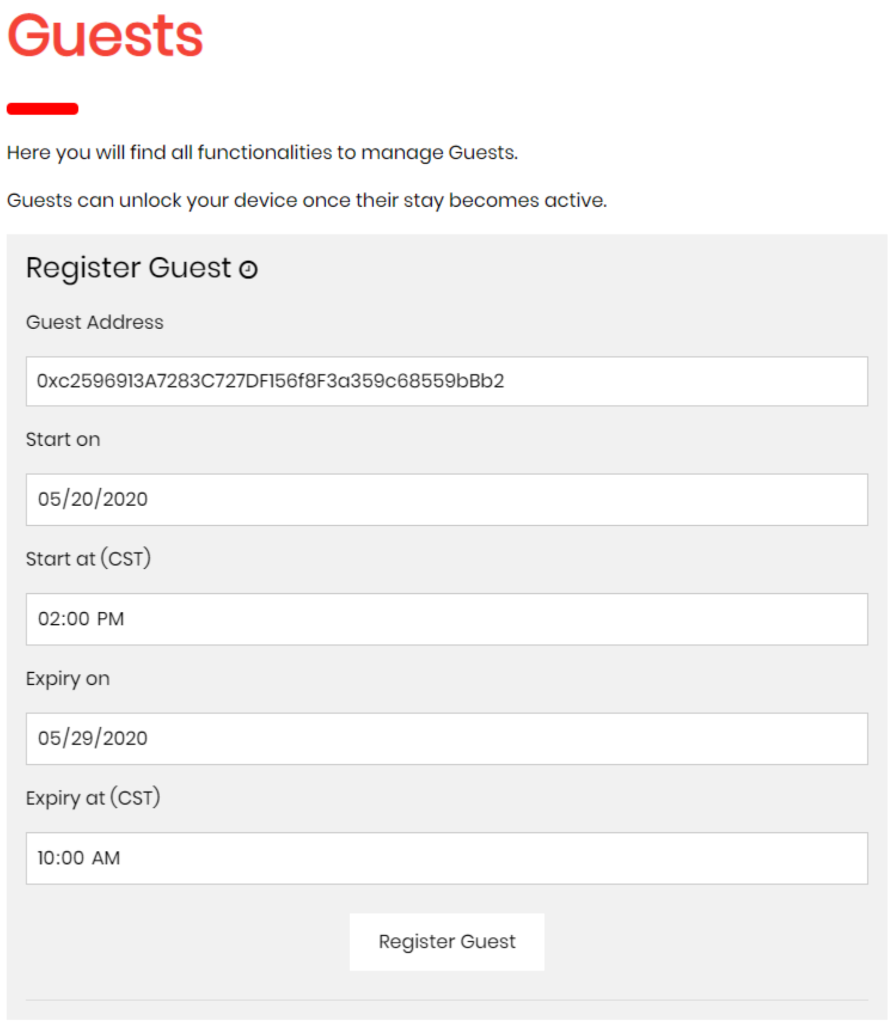

SmartContract
- _host : address
- _managers : mapping(address => ManagerDetails)
- _allManagers : address[ ]
- _guests : mapping(address => GuestPermissionDetails)
- _allGuests : address[ ]
- _exclusivePermission : ExclusiveFeatureDetails
+ <<view>> unlock(whoKnocks : address) : bool canUnlock
+ registerManager(manager : address) : void
+ removeManager(manager : address) : void
- <<event>> ManagerChange() : void
+ <<view>> retrieveManagers() : address[ ] allManagers
- <<view>> isManager(who : address) : bool isRegistered
+ registerGuest(guest : address, unixStartDate : uint256, unixExpiryDate : uint256) : void
+ removeGuest(guest : address) : void
- <<event>> GuestChange() : void
+ <<view>> retrieveGuests() : address[ ] allGuests
+ <<view>> guestSelfPermissionDetails() : details
- <<view>> guestIsRegistered(guest : address) : bool isRegistered
+ <<view>> guestPermissionDetails(guest : address) : (uint256 unixStartDate, uin256 unixExpiryDate)
+ turnExclusiveFeatureOn(holder : address, unixExpiryDate : uint256) : void
+ turnExclusiveFeatureOff() : void
- <<event>> ExclusiveFeatureChange() : void
+ <<view>> getExclusiveFeatureDetails() : (address permissionHolder, uint256 permissionUnixExpiryDate)
- <<view>> holdsActiveExclusivePermission(who : address) : bool itHolds
- <<view>> exclusiveFeatureIsOff() : bool isOff
- <<modifier>> onlyOwner()
- <<modifier>> eitherOwnerOrManager()
- <<modifier>> notExclusivePermissionHolder(who : address)
- swapManagerWithLastAtAllManagers(manager : address) : void
- swapGuestWithLastAtAllGuests(guest : address) : void
Struct ManagerDetails
isRegistered : uint256
indexAtAllManagersList : uint
Struct ExclusiveFeatureDetails
unixExpiryDate : uint256
holder : address
Struct GuestPermissionDetails
unixStartDate : uint256
unixExpiryDate : uint256
indexAtAllGuestsList : uint
details
uint256 unixStartDate
uint256 unixExpiryDate
bool canUnlockNow
bool holdsExclusivePermission,
uint256 exclPermExpiryDate)

Figure 5.18 – Smart contract's UML class diagram

5.4 Lock's Management Interface

As discussed in section 5.1, this interface is built using JavaScript, HTML, CSS, and Web3.js (2020). This tool allows Host, Managers, and Guests to execute all the actions as described by each of the eleven use cases. For instance, Figure 5.19 shows the interface to register a Guest. Note in the figure that the title brings a clock icon, which communicates to the user that the action – a transaction to the blockchain – might take some time to complete.


Guests

Here you will find all functionalities to manage Guests.

Guests can unlock your device once their stay becomes active.

Register Guest

Guest Address

0xc2596913A7283C727DF156f8F3a359c68559bBb2

Start on

05/20/2020

Start at (CST)

02:00 PM

Expiry on

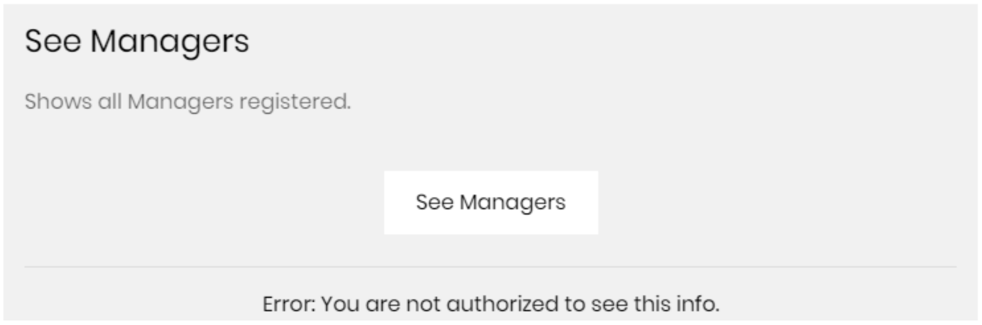
05/29/2020

Expiry at (CST)

10:00 AM

Figure 5.19 – Management Interface: Register Guest

The PoC version of the management interface shows all the functions regardless of the Actor type – Host, Managers, and Guests – even though only the Host can use all of them. The main reason for implementing the tool like that is to be able to test all the use cases properly, which includes Actors trying to execute actions that they are not allowed to do and that the system should handle at the smart contract level, as discussed before. Figure 5.20 illustrates a scenario where the Guest tries to retrieve the list of Managers, which they are not allowed to access.



Error: You are not authorized to see this info.

Figure 5.20 – Management Interface: Guest retrieve Managers list

The management interface includes two functionalities not specified by the use cases because they are related to the use of Ethereum smart contracts. First, the interface provides an option to deploy a new smart contract in the blockchain. Second, it enables users to inform the address of an existing contract. It is only after completion of one of those two options that the interface is set for use. Observe that users must keep their smart contract addresses saved somewhere once the management interface does not store any data. In addition to that, Hosts must share that address with their Managers and Guests to enable then to interact with their smart contract.

CHAPTER 6

EXPERIMENTS AND EVALUATIONS

Transactions on Ethereum are not free and, therefore, the system requires Host, Managers, and Guests, to spend money to manage the lock. Hence, it is critical to evaluate the system for operating costs. In addition to that, all the use cases must be systematically assessed to ensure that the system displays behaviors that match the description of the scenarios. Moreover, there is the necessity to measure the latency times encountered to manage the lock that is imposed by the Ethereum network. As a result, it is then possible to understand the impact that the blockchain has on the overall system, and what kind of limitation it imposes, if any. This chapter first presents the methodology developed for each experiment in section 6.1, followed by the results and their corresponding discussions in section 6.2.

6.1 Methodology

Initially, it is fundamental to observe that all the evaluations presented in this section are connected to the eleven use cases developed earlier in chapter four. Sets of those test cases were selected to validate the system functionalities, measure its performance, as well as evaluate its running costs.

This section is organized as follows: 6.1.1 is the test case details; 6.1.2 presents the methodology for three experiments designed to evaluate performance; and 6.1.3 details the methodology for three experiments designed to evaluate the cost. Table 6.1 shows how those six experiments are related to the research objectives presented in chapter two.

Table 6.1 – Relationship between the experiments and the research objectives from chapter two

Research Objective	**Experiments**
2.a – Verify functionalities	6.1.2.1
2.b – Evaluate delay	6.1.2.1, 6.1.2.2, and 6.1.2.3
2.c – Evaluate costs	6.1.3.1, 6.1.3.2, and 6.1.3.3

6.1.1 Test cases

Since all use cases already show systematically detailed descriptions for their functionalities, they can be treated as test cases. Every use case has a main success scenario and some alternative paths, where each of them represents a test case for that functionality. Test cases are referred to by the use case number plus either the letters “MSS,” for the main success scenario, or the letters “AP” followed by a number, for the individual alternative path. For instance, the test case UC1.AP3 represents the unlock and lock the door use case executed through the alternative path 3, and UC1.MSS refers to its main success scenario. Moreover, multiple executions of a single test case might happen in some evaluations– e.g., a Host registering several Guests. The test case sets defined for each experiment will be described in the according section.

Besides the test cases extracted from the use cases, the system requires one further test. In order to set up the system, the smart contract must be deployed in the Ethereum blockchain, which adds costs and performance considerations to the system. This test case is called “Deploy.”

Although the use cases are described systematically in terms of functionality, they lack relevant information about their corresponding implementation. For instance, the test cases must carefully describe where they initiate the action, what are the components and Actors involved, which smart contract functions are called, among others. To address those matters, figures 6.1 to 6.21 are UML sequence diagrams that illustrates the use cases flow.

The sequence diagrams make some assumptions:

- The Actor is logged in to the Metamask wallet;
- The smart contract is deployed in the blockchain – except for the Deploy use case of Figure 6.21;
- The lock’s interface management tool is set with both the smart contract address and the Actor’s Ethereum account address;
- The Actor always sends the transaction to the smart contract, even in those cases where they get a warning saying that a transaction might fail;
- Metamask obtains the transaction confirmation from the blockchain. How this process happens internally in Metamask, however, is not relevant to the diagrams.

Once the implementation of the lock's hardware is out of the scope of this work, the test case for UC1 emulates the hardware call through the lock's management interface – as explained in section 5.1.

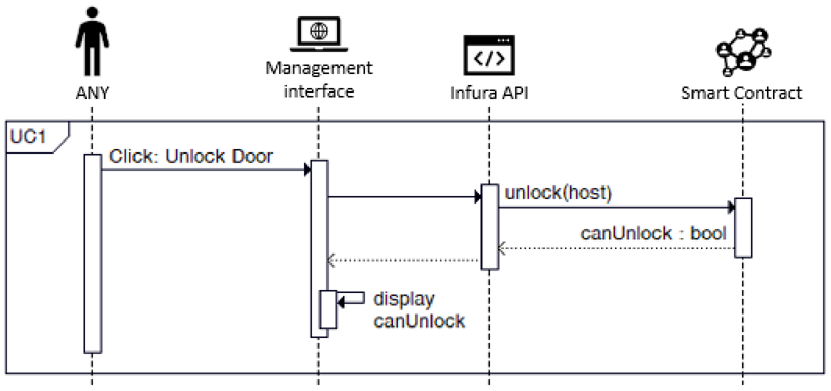


Figure 6.1 – UC1 sequence diagram – Unlock and lock the door

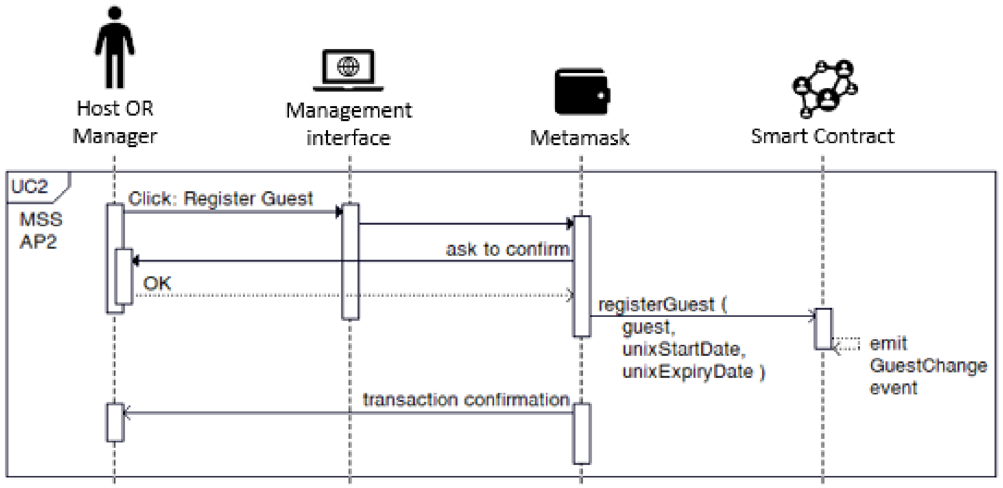


Figure 6.2 – UC2.MSS, AP2 sequence diagram – Register Guest

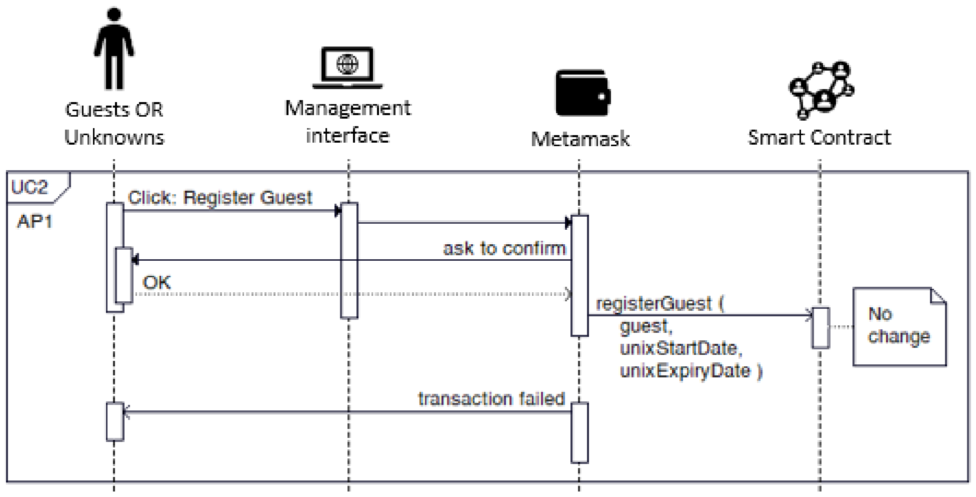


Figure 6.3 – UC2.AP1 sequence diagram – Register Guest

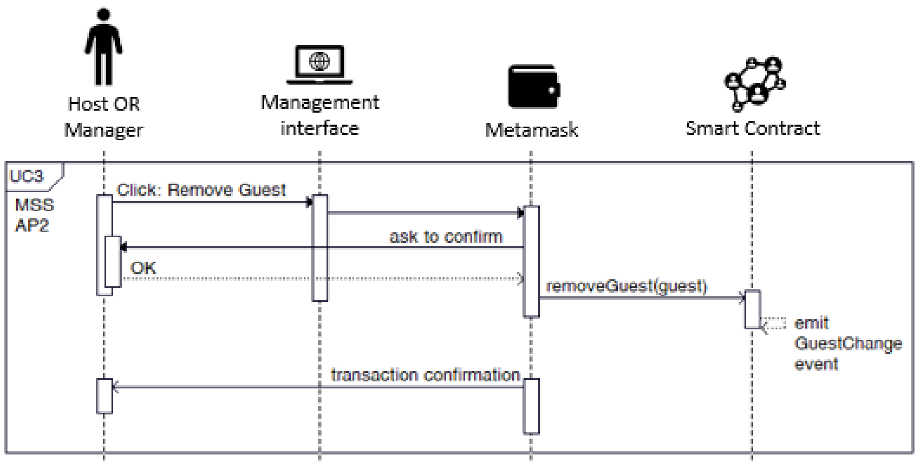


Figure 6.4 – UC3.MSS, AP2 sequence diagram – Remove Guest

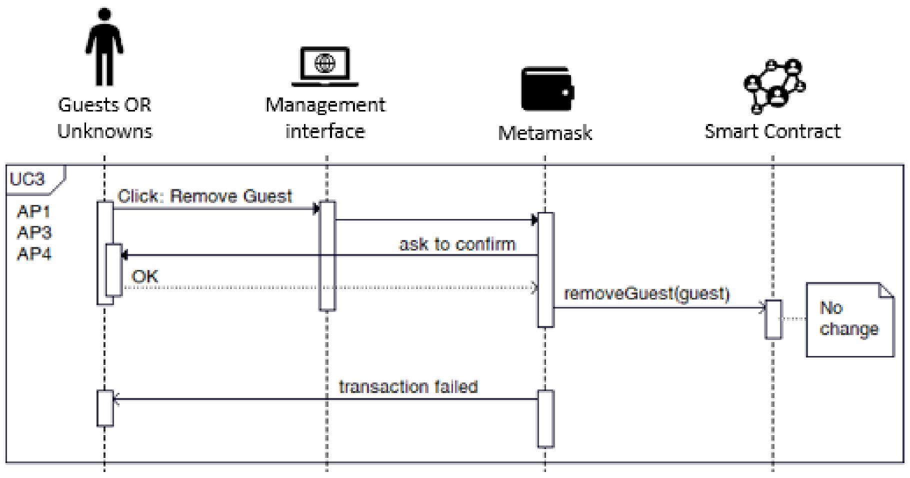


Figure 6.5 – UC3.AP1, AP3, AP4 sequence diagram – Remove Guest

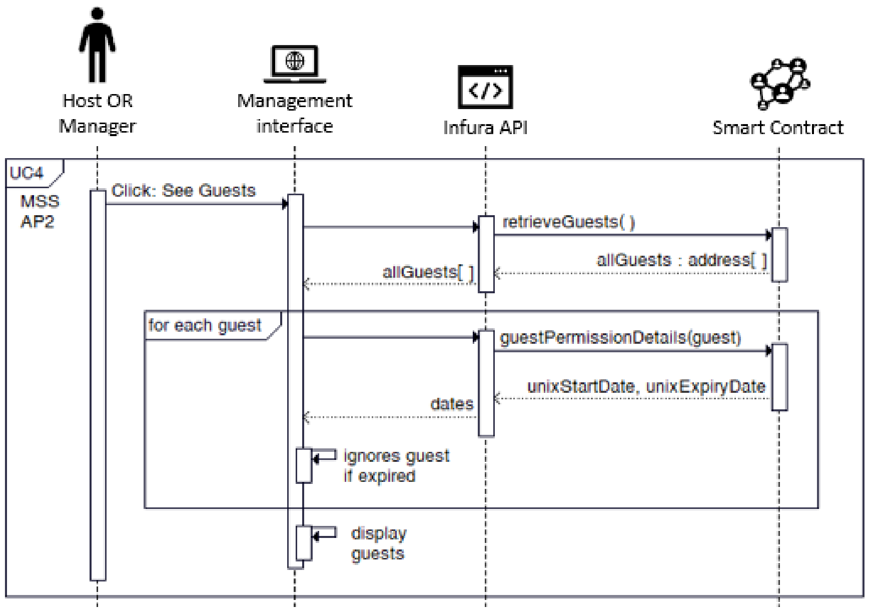


Figure 6.6 – UC4.MSS, AP2 sequence diagram – Retrieve Guests

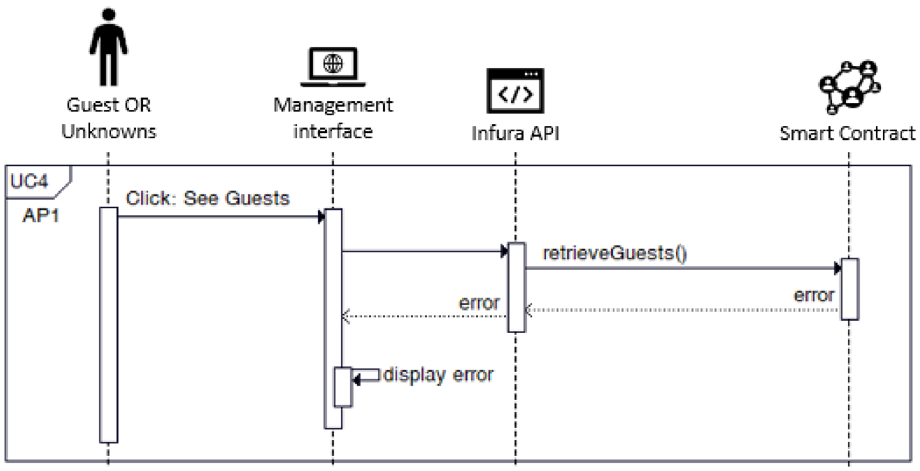


Figure 6.7 – UC4.AP1 sequence diagram – Retrieve Guests

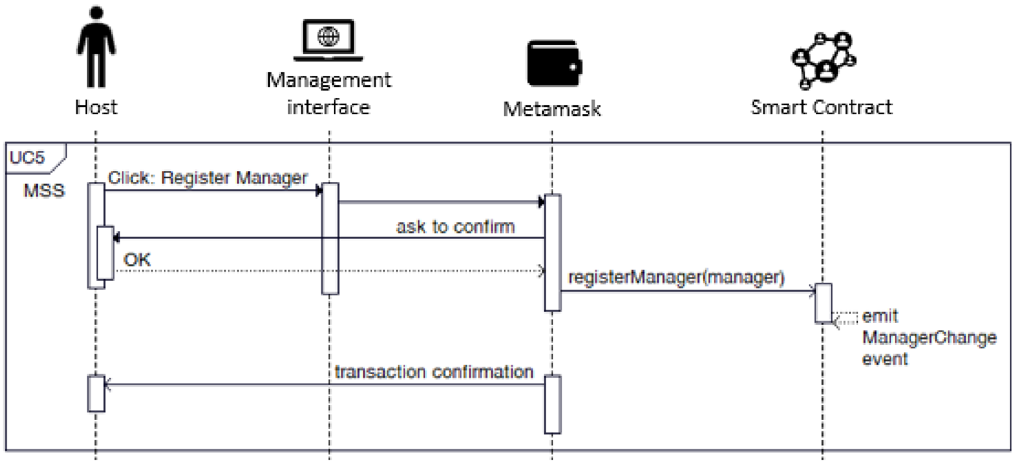


Figure 6.8 – UC5.MSS sequence diagram – Register Manager

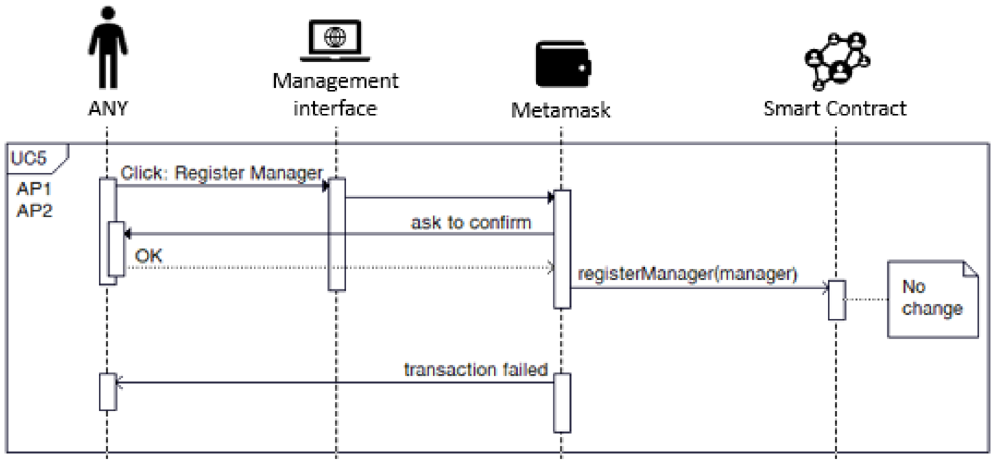


Figure 6.9 – UC5.AP1, AP2 sequence diagram – Register Manager

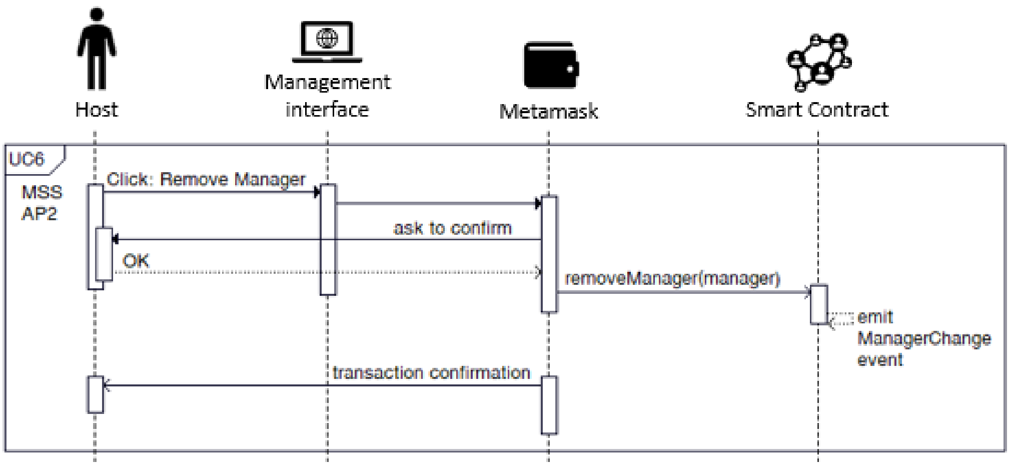


Figure 6.10 – UC6.MSS, AP2 sequence diagram – Remove Manager

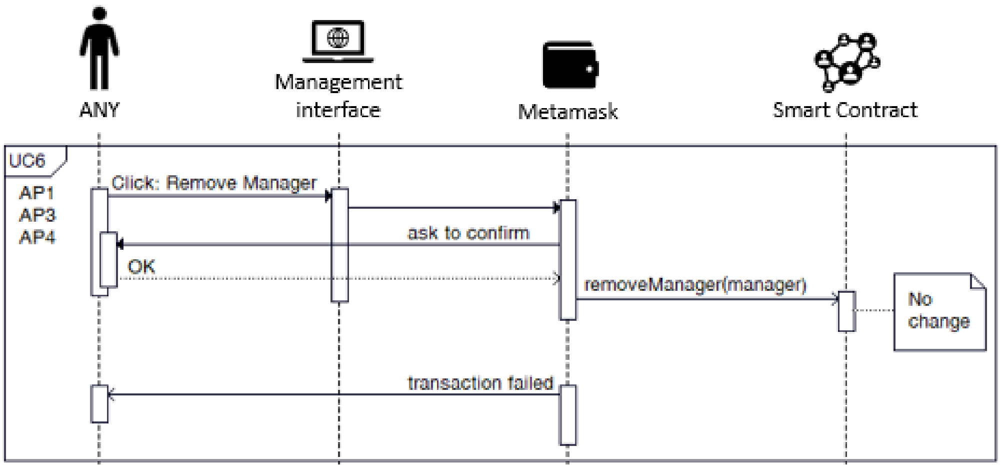


Figure 6.11 – UC6.AP1, AP3, AP4 sequence diagram – Remove Manager

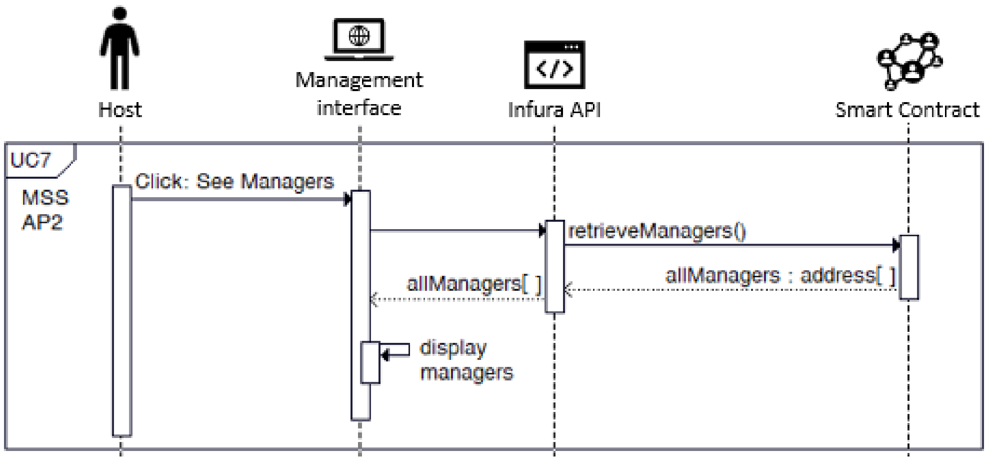


Figure 6.12 – UC7.MSS, AP2 sequence diagram – Retrieve Managers

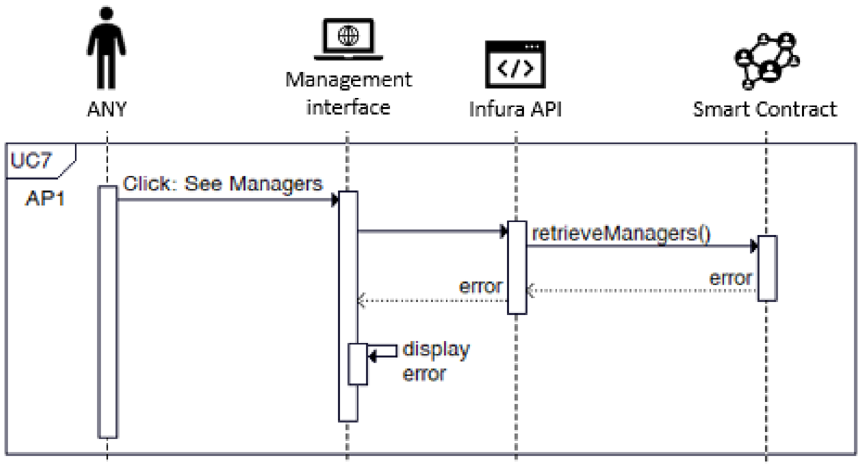


Figure 6.13 – UC7.AP1 sequence diagram – Retrieve Managers

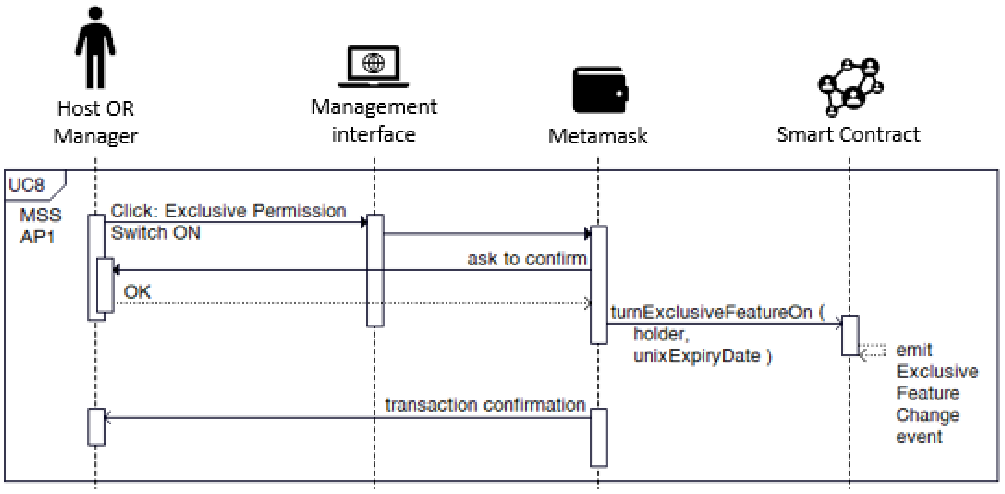

Host OR Manager
Management interface
Metamask
Smart Contract
UC8
MSS
AP1
Click: Exclusive Permission Switch ON
ask to confirm
OK
turnExclusiveFeatureOn (holder, unixExpiryDate)
emit Exclusive Feature Change event
transaction confirmation

Figure 6.14 – UC8.MSS, AP1 sequence diagram – Turn on exclusive permission

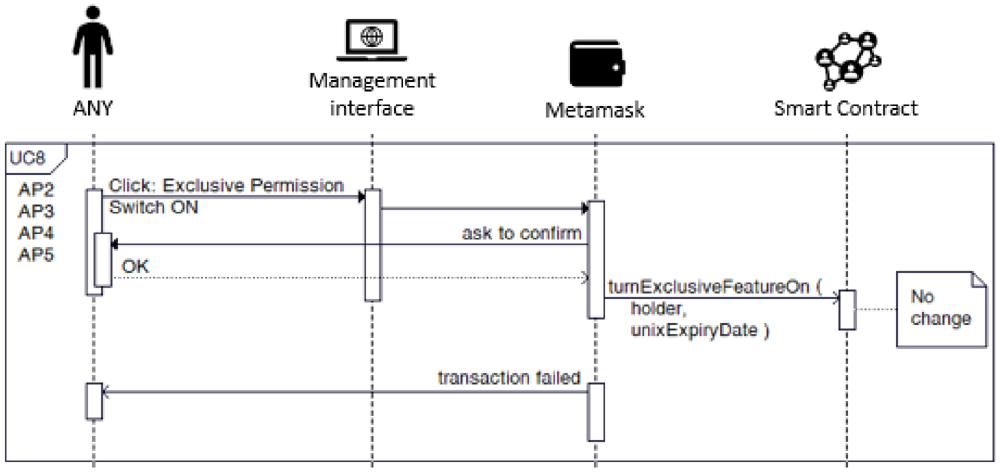


Figure 6.15 – UC8.AP2, AP3, AP4, AP5 sequence diagram – Turn on exclusive permission

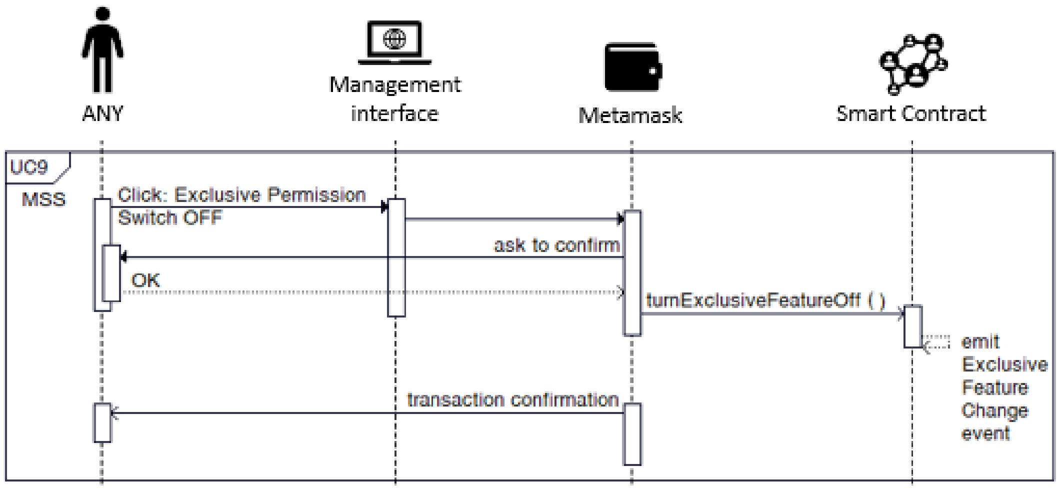

ANY
Management interface
Metamask
Smart Contract
UC9
MSS
Click: Exclusive Permission Switch OFF
ask to confirm
OK
turnExclusiveFeatureOff ( )
emit Exclusive Feature Change event
transaction confirmation

Figure 6.16 – UC9.MSS sequence diagram – Turn off exclusive permission

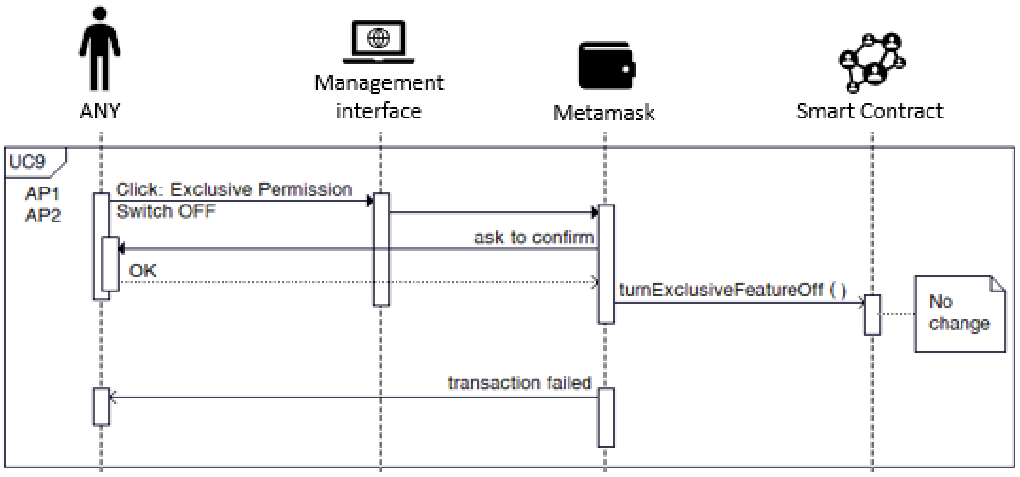


Figure 6.17 – UC9.AP1, AP2 sequence diagram – Turn off exclusive permission

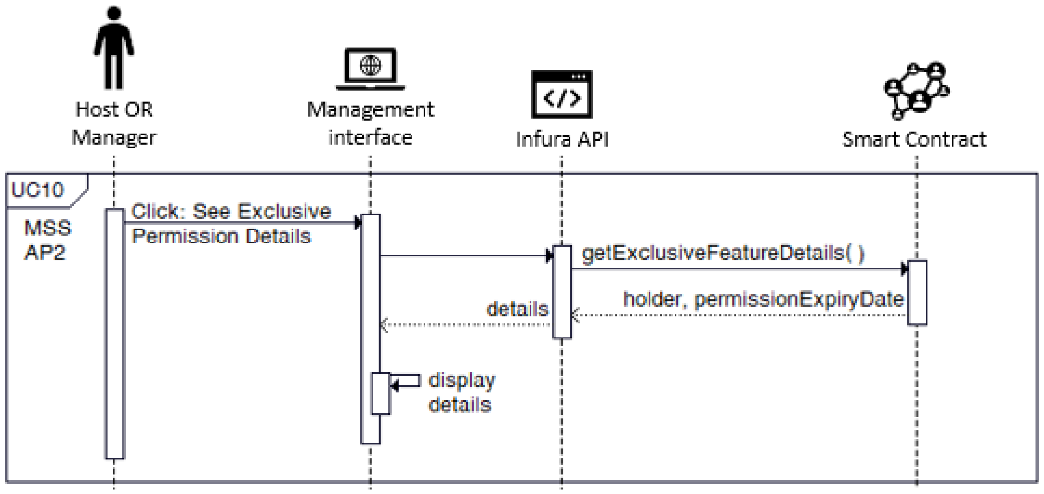

Host OR Manager
Management interface
Infura API
Smart Contract
UC10
MSS
AP2
Click: See Exclusive Permission Details
getExclusiveFeatureDetails( )
holder, permissionExpiryDate
details
display details

Figure 6.18 – UC10.MSS, AP2 sequence diagram – Check exclusive permission

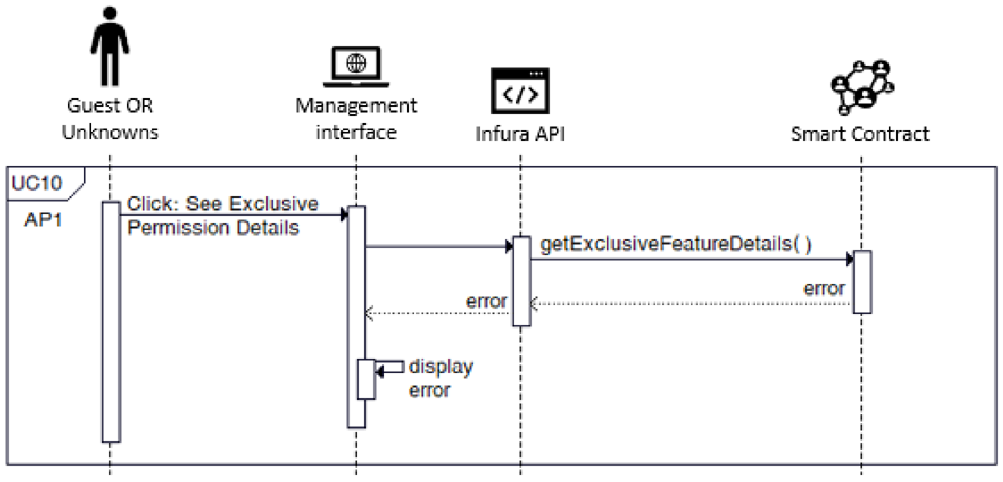


Figure 6.19 – UC10.AP1 sequence diagram – Check exclusive permission

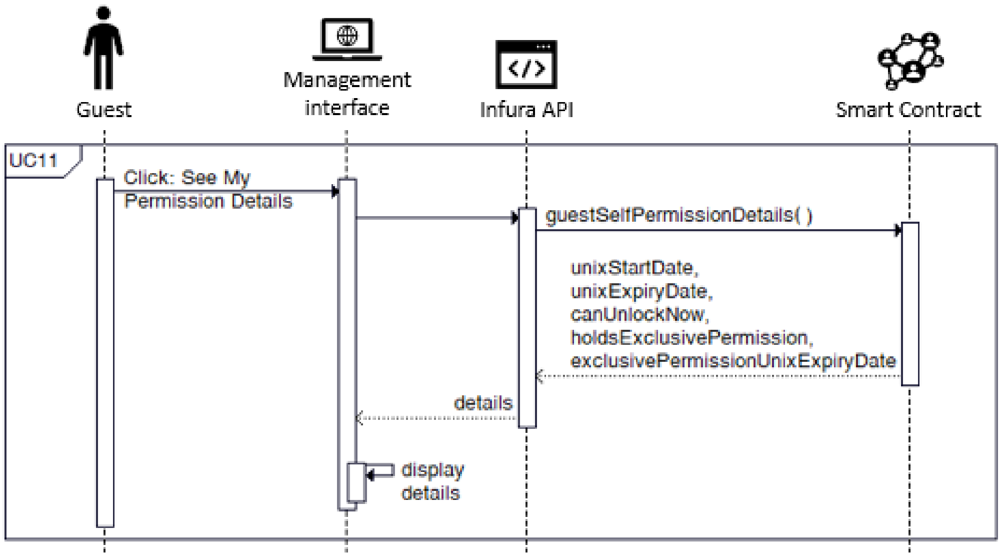

Guest
Management interface
Infura API
Smart Contract
UC11
Click: See My Permission Details
guestSelfPermissionDetails( )
unixStartDate,
unixExpiryDate,
canUnlockNow,
holdsExclusivePermission,
exclusivePermissionUnixExpiryDate
details
display details

Figure 6.20 – UC11 sequence diagram – Verify permission

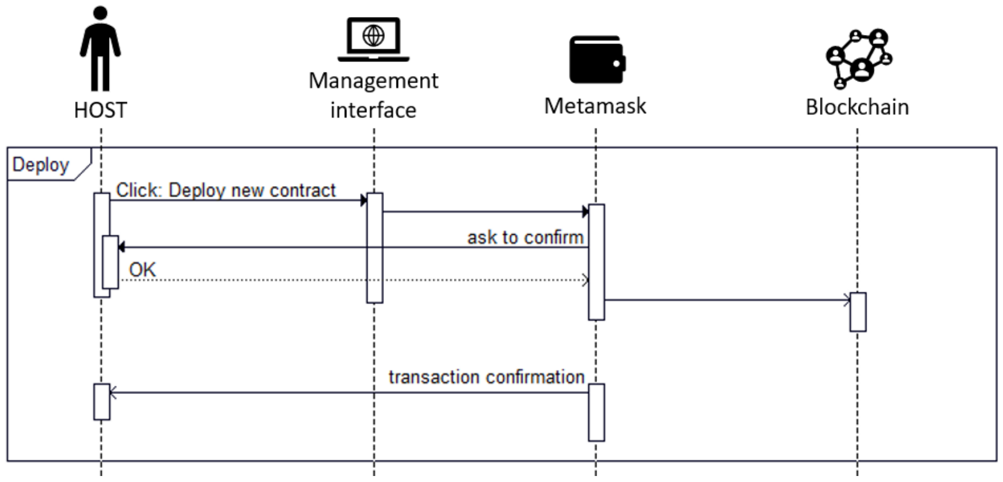


Figure 6.21 – Deploy sequence diagram

6.1.2 Evaluating performance

Although the Ethereum test network Görli is an excellent choice to emulate the Ethereum blockchain functional behavior, it is not suitable to evaluate performance metrics and extrapolate the findings to the Ethereum main network – also referred to as *mainnet*. For this work, performance is measured by how long the blockchain takes to process a transaction and to reply to a query call. This time is referred to as transaction latency, and it might vary according to a diverse set of factors, e.g., by the number of transactions at a given time. The Gas price offered by users also has an impact on that considering that, the higher the value, the more attractive it is to miners to process the transaction and collect the fee. This may lead to lower latency times in comparison to other transactions since the Ethereum network does not impose an order that the transactions must be processed, i.e., the miners are free to choose among them. Therefore, there is not a direct relationship between the performance of Ethereum test networks and the *mainnet*. In short, in order to properly evaluate the system's performance, the smart contract must be deployed in the production environment, the main Ethereum network.

In total, this thesis proposes three experiments to measure performance: (i) a base case; (ii) a network dynamics case; (iii) a Gas price case. The three of them enable a discussion about what the use cases latency are, how they can be affected by the network dynamics, and how they can be manipulated by the Gas price setting. The next sections detail each one of these experiments.

6.1.2.1 The base case experiment

This experiment follows the methodology presented below:

1. Define a methodical set of test cases chronologically arranged – see Table 6.2;
2. Define a fixed set of Ethereum accounts and the actors they correspond to – see Table 6.3;
3. Define a fixed Gas price to be offered by every transaction;
4. Execute the test cases in the mainnet, using a single Metamask wallet account and the same browser;
    4.1 Record the transaction number – if applicable;
    4.2 Record the smart contract address;

4.3 Record if the behavior matches the expectation;

4.4 For transactions:

- Record both the submission and confirmation times obtained from the Metamask wallet log;

4.5 For queries – calls to the smart contract's *view* functions:

- Record the submission time obtained from the management interface tool;
- Record the response time obtained from the management interface tool;

4.6 Calculate and record the latency as the response time minus the submission time.

To address the third step of the methodology, the definition of the Gas price, Ethereum Gas Station website (2020) is used. This website monitors Gas prices in the network and provides real-time recommendations of prices based on the expected confirmation time for a transaction. It offers three price recommendations: (i) the ”Safe Low” price represents an expected confirmation time up to 30 minutes; (ii) the "Standard" price, up to 5 minutes; (iii) the "Fast" price, up to 2 minutes. Those prices regularly change according to the network usage. For this experiment, the "Fast" recommended price when starting the experiment execution must be taken as the fixed value for step three. In other words, immediately before starting the execution of this experiment, the "Fast" price recommended by the Ethereum Gas Station at that moment must be used by all the transactions defined by step two. Even if the recommended price changes during the experiment, the Gas price offered should not change.

The Actor "Guest Past" represents a past Guest of the Host, meaning that the person has had permission at some point to open the lock, but now it has expired. In practice, to simulate that scenario, that Guest is registered with an expired permission date - start and expiry date set anywhere in the past. Similarly, the "Guest Future" represents a Guest who will stay at the place at some point. Therefore, it is a Guest that is registered with both start and expiry permission dates set anywhere in the future.

Table 6.2 – Set of tests for the performance base case experiment

ID	**Short Description**	**Use Case**	**Call Arguments**	**Caller**
T1	Smart contract deploy	Deploy		Host
T2	Host can open the lock	UC1.MSS	Host	Host
T3	Host adds new Manager	UC5.MSS	Manager 1	Host
T4	Manager can open the lock	UC1.MSS	Manager 1	Manager 1

Table 6.2 Continue				
ID	**Short Description**	**Use Case**	**Call Arguments**	**Caller**
T5	Host tries to add existing Manager	UC5.AP2	Manager 1	Host
T6	Manager tries to add a new Manager	UC5.AP1	Manager 2	Manager 1
T7	Unknown tries to add new Manager	UC5.AP1	Manager 2	Unknown
T8	Host adds new Guest	UC2.MSS	Guest 1, 1583049600, 1591027200	Host
T9	Manager adds new Guest	UC2.MSS	Guest Future, 1591027200, 1593619200	Manager 1
T10	Guest can open the lock	UC1.AP3	Guest 1	Guest 1
T11	Guest verifies its permission details	UC11.MSS		Guest 1
T12	Future Guest tries to open the lock	UC1.AP4	Guest Future	Guest Future
T13	Guest Future verifies its permission details	UC11.MSS		Guest Future
T14	Unknown verifies its permission details	UC11.AP3		Unknown
T15	Guest tries to add a new Manager	UC5.AP1	Manager 2	Guest 1
T16	Guest tries to add Guest	UC2.AP1	Guest 1, 1583049600, 1592027220	Guest 1
T17	Unknown tries to add Guest	UC2.AP1	Guest 1, 1583049600, 1592027220	Unknown
T18	Host adds existing Guest	UC2.AP2	Guest 1, 1583049600, 1593027240	Host
T19	Host adds new Manager	UC5.MSS	Manager 2	Host
T20	Manager adds new Guest	UC2.MSS	Guest Past, 1580554800, 1581554820	Manager 2
T21	Past Guest tries to open the lock	UC1.AP5	Guest Past	Guest Past
T22	Host retrieves Guests	UC4.MSS		Host
T23	Manager retrieves Guests	UC4.MSS		Manager 2
T24	Guest tries to retrieve Guests	UC4.AP1		Guest 1
T25	Unknown tries to retrieve Guests	UC4.AP1		Unknown
T26	Host retrieves Managers	UC7.MSS		Host
T27	Manager tries to retrieve Managers	UC7.AP1		Manager 1
T28	Guest tries to retrieve Managers	UC7.AP1		Guest Future
T29	Unknown tries to retrieve Managers	UC7.AP1		Unknown
T30	Host turns exclusive on to self	UC8.MSS	Host, 1591027200	Host
T31	Manager checks exclusive details	UC10.MSS		Manager 1
T32	Host exclusive ca n open the lock	UC1.AP1	Host	Host

Table 6.2 Continue				
ID	**Short Description**	**Use Case**	**Call Arguments**	**Caller**
T33	Manager tries to open the lock when exclusive is on to other	UC1.AP2	Manager 1	Manager 1
T34	Host exclusive turns it off	UC9.MSS		Host
T35	Host checks exclusive details when off	UC10.AP2		Host
T36	Manager tries to turn exclusive on to self	UC8.AP2	Manager 1, 1591027200	Manager 1
T37	Host turns exclusive on to Manager	UC8.AP2	Manager 1, 1591027200	Host
T38	Host checks exclusive details	UC10.MSS		Host
T39	Guest tries to check exclusive details	UC10.AP1		Guest 1
T40	Unknown tries to check exclusive details	UC10.AP1		Unknown
T41	Manager exclusive can open the lock	UC1.AP1	Manager 1	Manager 1
T42	Host tries to remove Manager that holds exclusive permission	UC6.AP3	Manager 1	Host
T43	Host adds new Manager while exclusive is on	UC5.MSS	Manager 3	Host
T44	Host removes Manager while exclusive is on to another Manager	UC6.AP2	Manager 3	Host
T45	Guest verifies its permission details when exclusive is on to other	UC11.AP2		Guest 1
T46	Host tries to open the lock when exclusive is on to other	UC1.AP2	Host	Host
T47	Guest tries to open the lock when exclusive is on to other	UC1.AP2	Guest 1	Guest 1
T48	Host tries to turn exclusive off when on to other	UC9.AP2		Host
T49	Manager exclusive turns it off	UC9.MSS		Manager 1
T50	Host tries to turn exclusive off when off	UC9.AP1		Host
T51	Guest tries to turn exclusive on to self	UC8.AP3	Guest 1, 1591027200	Guest 1
T52	Unknown tries to turn exclusive on to Guest	UC8.AP3	Guest 1, 1591027200	Unknown
T53	Manager turns exclusive on to Guest	UC8.AP1	Guest 1, 1591027200	Manager 1
T54	Host tries to remove Guest that holds exclusive permission	UC3.AP3	Guest 1	Host
T55	Host removes Guest while exclusive is on to another Guest	UC3.AP2	Guest Past	Host
T56	Guest exclusive can open the lock	UC1.AP1	Guest 1	Guest 1

Table 6.2 Continue				
ID	**Short Description**	**Use Case**	**Call Arguments**	**Caller**
T57	Host tries to open the lock when exclusive is on to other	UC1.AP2	Host	Host
T58	Host tries to turn exclusive on to Guest when already on	UC8.AP4	Guest 1, 1591037220	Host
T59	Guest exclusive verifies its permission details	UC11.AP1		Guest 1
T60	Guest exclusive turns it off	UC9.MSS		Guest 1
T61	Host tries to turn exclusive on to Future Guest	UC8.AP5	Guest Future, 1591037220	Host
T62	Host tries to turn exclusive on to Unknown	UC8.AP5	Unknown, 1591037220	Host
T63	Guest tries to remove Guest	UC3.AP1	Guest Future	Guest 1
T64	Unknown tries to remove Guest	UC3.AP1	Guest Future	Unknown
T65	Manager removes Guest	UC3.MSS	Guest Future	Manager 2
T66	Host tries to remove a not existing Guest	UC3.AP4	Unknown	Host
T67	Guest tries to remove Manager	UC6.AP1	Manager 2	Guest 1
T68	Host removes Guest	UC3.MSS	Guest 1	Host
T69	Host retrieves Guests when none is registered	UC4.AP2		Host
T70	Manager tries to remove Manager	UC6.AP1	Manager 2	Manager 1
T71	Unknown tries to remove Manager	UC6.AP1	Manager 1	Unknown
T72	Host tries to remove a not existing Manager	UC6.AP4	Unknown	Host
T73	Host removes Manager	UC6.MSS	Manager 1	Host
T74	Host removes Manager	UC6.MSS	Manager 2	Host
T75	Host retrieves Managers when none is registered	UC7.AP2		Host

Table 6.3 – Ethereum accounts for the performance base case experiment

Actor	**Account address**
Host	0xcA18f8947783a38A61a710752673B3f5d0159F6F
Manager 1	0x470b57384Be9C15C9416958D8D35027a0b2a9f30
Manager 2	0x71330E718D52b82506c14A18bE625C585F194b01
Guest 1	0xc2596913A7283C727DF156f8F3a359c68559bBb2
Guest Past	0xDb0Dd99ffd184DcC1aD7C908a485E89A3a054935
Guest Future	0x59494603B8B16EaB741e9730cf612BF89cC2bC6D
Unknown	0x3d3bEcd44835ded70fB3820D3F9AA52aa3b308b3

6.1.2.2 The network dynamics experiment

Performance experiment two is set to evaluate how different Ethereum network conditions can impact the latency time. It is crucial to note that the proposed methodology assumes that a different day of the week and time of the day manifest distinct network dynamics. Ideally, those transactions happening on a later day should occur precisely at the same time as the previous day. However, since the transaction confirmation time varies unpredictably, and the test cases are sequential, the methodology sets only a start time. Then, the subsequent tests are continuously executed as early as the previous transactions are confirmed. The detailed methodology is:

1. Define a methodical and chronologically arranged subset of transactions from the base case experiment – see Table 6.4, which uses the same Ethereum accounts from Table 6.3;
2. Use the same Gas price offered in the base case experiment;
3. Execute the set of test cases:
   - Starting at nine AM on a Thursday;
   - Starting at six PM on the same Thursday;
   - Starting at nine AM on the following Sunday;
   - Starting at six PM on the same Sunday;
4. Execute the test cases in the mainnet, using the same Metamask wallet account and the same browser of the base case experiment;

   4.1 Execute the following test case as soon as the previous one is confirmed by the network;

   4.2 Record the transaction number;

   4.3 Record the smart contract address;

   4.4 Record both the submission and confirmation times obtained from the Metamask wallet log;

Table 6.4 – Set of tests for the network dynamics experiment

ID	Short Description	Use Case	Call Arguments	Caller
T1	Smart contract deploy	Deploy		Host
T3	Host adds new Manager	UC5.MSS	Manager 1	Host
T8	Host adds new Guest	UC2.MSS	Guest 1, 1583049600, 1591027200	Host

Table 6.4 Continue				
ID	**Short Description**	**Use Case**	**Call Arguments**	**Caller**
T9	Manager adds new Guest	UC2.MSS	Guest Future, 1591027200, 1593619200	Manager 1
T19	Host adds new Manager	UC5.MSS	Manager 2	Host
T30	Host turns exclusive on to self	UC8.MSS	Host, 1591027200	Host
T34	Host exclusive turns it off	UC9.MSS		Host
T37	Host turns exclusive on to Manager	UC8.AP2	Manager 1, 1591027200	Host
T49	Manager exclusive turns it off	UC9.MSS		Manager 1
T53	Manager turns exclusive on to Guest	UC8.AP1	Guest 1, 1591027200	Manager 1
T60	Guest exclusive turns it off	UC9.MSS		Guest 1
T65	Manager removes Guest	UC3.MSS	Guest Future	Manager 2
T68	Host removes Guest	UC3.MSS	Guest 1	Host
T73	Host removes Manager	UC6.MSS	Manager 1	Host
T74	Host removes Manager	UC6.MSS	Manager 2	Host

6.1.2.3 The Gas price experiment

Performance experiment three is set to investigate the impact of Gas price on the transaction confirmation time, where three distinct values are offered. Due to the networking dynamics, ideally, transactions having different Gas prices must be submitted simultaneously. It is important to highlight that Ethereum does not process multiple transactions from a single account in parallel. The blockchain follows the order that they were issued, which means that only after the first transaction sent is confirmed, the second is taken, and so on. Therefore, three different accounts are needed to examine the desired Gas price influence.

Having that said, a custom version of the management interface was developed to reduce the time necessary to submit those transactions. Essentially, this changed interface enables the creation and use of three contracts, each of which offers a different Gas price for its transactions. The value is selected accordingly to the account currently in use in Metamask. However, even with those changes, it is not possible to send all three transactions simultaneously. It is still required to manually change between accounts in Metamask and approve each transaction manually. Nevertheless, they are executed shortly after one another under the assumption that a few seconds is not enough to have a significant change in the network dynamics that could affect the experiment.

The detailed methodology is:

1. Define a methodical set of test cases chronologically arranged – see Table 6.5;
2. Define a fixed set of Ethereum accounts and the actors they correspond to – see Table 6.6;
3. Define three different Gas prices;
4. Execute the test cases in the mainnet, using the same Metamask wallet account and the same browser of the base case experiment;
   - 4.1 For each test case, select the appropriate account in Metamask and execute the test case with the corresponding Gas price offer – start from the lowest Gas price, then do the medium, and finally, do the highest price;
   - 4.2 Execute the next test case only after the network confirms all three previous transactions;
   - 4.3 Record the smart contract address;
   - 4.4 Record the transactions number;
   - 4.5 Record both the submission and confirmation times obtained from the Metamask wallet log.

Table 6.5 shows three possible callers for the first test case, but each Host must execute it exactly once. The following test cases refer to the "Respective Host" generally meaning the Actor that deployed the corresponding smart contract. For instance, GT3 is executed three times, once from each Host – Low, Medium, and High – where each of them interacts with the smart contract they deployed in GT1 using their address as the call argument.

To address the third step of the methodology, the definition of Gas prices, the values 3, 6, and 12 Gwei ($12x10^{-9}$ ETH) are used. The reason for that choice is to have a fixed multiplier factor difference between the prices to investigate the resulting impact they have in the latency.

Table 6.5 – Set of tests for the Gas price experiment

ID	**Short Description**	**Use Case**	**Call Arguments**	**Caller**
GT1	Smart contract deploy	Deploy		Host Low Host Medium Host High
GT2	Host adds new Manager	UC5.MSS	Manager PE3	Respective Host
GT3	Host adds new Guest	UC2.MSS	Guest PE3, 1583049600, 1591027200	Respective Host

Table 6.5 Continue				
ID	**Short Description**	**Use Case**	**Call Arguments**	**Caller**
GT4	Host turns exclusive on to self	UC8.MSS	Respective Host, 1591027200	Respective Host
GT5	Host exclusive turns it off	UC9.MSS		Respective Host
GT6	Host removes Guest	UC3.MSS	Guest PE3	Respective Host
GT7	Host removes Manager	UC6.MSS	Manager PE3	Respective Host

Table 6.6 – Ethereum accounts for the Gas price experiment

Actor	**Account address**
Host Low	0x71330E718D52b82506c14A18bE625C585F194b01
Host Medium	0x470b57384Be9C15C9416958D8D35027a0b2a9f30
Host High	0xcA18f8947783a38A61a710752673B3f5d0159F6F
Guest PE3	0xc2596913A7283C727DF156f8F3a359c68559bBb2
Manager PE3	0xDb0Dd99ffd184DcC1aD7C908a485E89A3a054935

6.1.3 Evaluating cost

A transaction cost in Ethereum is determined by its Gas consumption, measured in units of Gas. Each low-level Ethereum EVM operation has a determined Gas cost, and the transaction is charged according to the low-level steps it goes through. Two steps are necessary to arrive at the Canadian Dollar equivalent cost from the Gas consumption value. First, when submitting a transaction, users choose the Gas price in Ether (ETH), the Ethereum currency, they are willing to pay – see section 6.1.1. Therefore, the transaction cost in ETH is a simple multiplication of the Gas consumption and the price offered. Second, the ETH price in Canadian Dollars is set by the market, how much people are willing to pay to have them.

This section proposes three experiments to evaluate the cost to run the smart contract: (i) a base case; (ii) a test network case; (iii) a multiplicity case. They enable a discussion about what the running cost expected for using the smart contract is, how the number of actors registered impact that cost, and how accurate are the Gas consumption calculations of the Görli test network.

6.1.3.1 The base case experiment

The same test set used by the performance base case can be used to assess cost. Since the execution of that experiment already yields the Gas consumption and Gas cost of the transactions, without requiring any extra step, it is not necessary to perform a new experiment. All that is needed for this base case cost evaluation is to record the Gas consumption registered when running the experiment 6.1.2.1, obtaining the value from Etherscan. Observe that query call test cases – e.g., T2 “Host can open lock” - can be ignored since they are free of charge.

Note that the execution of performance experiments two and three, network and gas price respectively, also provide Gas consumption values without any further work. Those observations are useful to compare Gas consumption consistency when performing the same action, for instance. For that reason, the Gas consumption registered when running those experiments must also be compiled, obtaining the value from Etherscan.

In summary, the base case cost evaluation consists of gathering the Gas consumption data obtained through all three performance experiments.

In addition to gathering Gas consumption, the ETH cost in Canadian Dollars (CAD) must be defined. The Coinbase (2020) website is used as it tracks the market value of ETH. The methodology is:

1. Collect the Gas consumption and Gas price registered by all the transactions from the three performance experiments – see section 6.1.2 -, obtaining the values through the Etherscan website;
2. Define the ETH cost in Canadian Dollars

6.1.3.2 The test network experiment

Having the base case evaluation executed in production brings an exciting opportunity for the second experiment, which is to evaluate the Gas consumption calculations issued by Görli. Differently from the performance experiments discussion, network dynamics do not influence the Gas consumption of a transaction. In other words, Ethereum test networks as Görli can and do emulate that value. However, it is not clear how accurate those calculations are.

This experiment to investigate the accuracy, then, is to repeat the transactions from the performance base case experiment methodically but using the Görli test network and ignoring transaction latency measurements. Observe that query call test cases – e.g., T2 "Host can open lock" - can be ignored since they are free of charge. The methodology in detail is:

1. Methodically repeat all the transactions from the performance base case experiment from section 6.1.2.1 – see Table 6.7, which uses the same Ethereum accounts from Table 6.3;
2. Execute the test cases in the Görli test network;
   - 2.1 Record the transaction number;
   - 2.2 Record the smart contract address;
   - 2.3 Record if the behavior matches the expectation – otherwise, it might yield a different cost caused by a wrong behavior;
   - 2.4 Record the Gas consumption obtained from the Etherscan website

Table 6.7 – Set of tests for the network cost experiment

ID	Short Description	Use Case	Call Arguments	Caller
T1	Smart contract deploy	Deploy		Host
T3	Host adds new Manager	UC5.MSS	Manager 1	Host
T5	Host tries to add existing Manager	UC5.AP2	Manager 1	Host
T6	Manager tries to add a new Manager	UC5.AP1	Manager 2	Manager 1
T7	Unknown tries to add new Manager	UC5.AP1	Manager 2	Unknown
T8	Host adds new Guest	UC2.MSS	Guest 1, 1583049600, 1591027200	Host
T9	Manager adds new Guest	UC2.MSS	Guest Future, 1591027200, 1593619200	Manager 1
T15	Guest tries to add a new Manager	UC5.AP1	Manager 2	Guest 1
T16	Guest tries to add Guest	UC2.AP1	Guest 1, 1583049600, 1592027220	Guest 1
T17	Unknown tries to add Guest	UC2.AP1	Guest 1, 1583049600, 1592027220	Unknown
T18	Host adds existing Guest	UC2.AP2	Guest 1, 1583049600, 1593027240	Host
T19	Host adds new Manager	UC5.MSS	Manager 2	Host
T20	Manager adds new Guest	UC2.MSS	Guest Past, 1580554800, 1581554820	Manager 2
T30	Host turns exclusive on to self	UC8.MSS	Host, 1591027200	Host

Table 6.7 Continue				
ID	**Short Description**	**Use Case**	**Call Arguments**	**Caller**
T34	Host exclusive turns it off	UC9.MSS		Host
T36	Manager tries to turn exclusive on to self	UC8.AP2	Manager 1, 1591027200	Manager 1
T37	Host turns exclusive on to Manager	UC8.AP2	Manager 1, 1591027200	Host
T42	Host tries to remove Manager that holds exclusive permission	UC6.AP3	Manager 1	Host
T43	Host adds new Manager while exclusive is on	UC5.MSS	Manager 3	Host
T44	Host removes Manager while exclusive is on to other Manager	UC6.AP2	Manager 3	Host
T48	Host tries to turn exclusive off when on to other	UC9.AP2		Host
T49	Manager exclusive turns it off	UC9.MSS		Manager 1
T50	Host tries to turn exclusive off when off	UC9.AP1		Host
T51	Guest tries to turn exclusive on to self	UC8.AP3	Guest 1, 1591027200	Guest 1
T52	Unknown tries to turn exclusive on to Guest	UC8.AP3	Guest 1, 1591027200	Unknown
T53	Manager turns exclusive on to Guest	UC8.AP1	Guest 1, 1591027200	Manager 1
T54	Host tries to remove Guest that holds exclusive permission	UC3.AP3	Guest 1	Host
T55	Host removes Guest while exclusive is on to other Guest	UC3.AP2	Guest Past	Host
T58	Host tries to turn exclusive on to Guest when already on	UC8.AP4	Guest 1, 1591037220	Host
T60	Guest exclusive turns it off	UC9.MSS		Guest 1
T61	Host tries to turn exclusive on to Future Guest	UC8.AP5	Guest Future, 1591037220	Host
T62	Host tries to turn exclusive on to Unknown	UC8.AP5	Unknown, 1591037220	Host
T63	Guest tries to remove Guest	UC3.AP1	Guest Future	Guest 1
T64	Unknown tries to remove Guest	UC3.AP1	Guest Future	Unknown
T65	Manager removes Guest	UC3.MSS	Guest Future	Manager 2
T66	Host tries to remove a not existing Guest	UC3.AP4	Unknown	Host
T67	Guest tries to remove Manager	UC6.AP1	Manager 2	Guest 1
T68	Host removes Guest	UC3.MSS	Guest 1	Host
T70	Manager tries to remove Manager	UC6.AP1	Manager 2	Manager 1
T71	Unknown tries to remove Manager	UC6.AP1	Manager 1	Unknown

Table 6.7 Continue				
ID	**Short Description**	**Use Case**	**Call Arguments**	**Caller**
T72	Host tries to remove a not existing Manager	UC6.AP4	Unknown	Host
T73	Host removes Manager	UC6.MSS	Manager 1	Host
T74	Host removes Manager	UC6.MSS	Manager 2	Host

6.1.3.3 The multiplicity experiment

Finally, the last experiment is set to evaluate if the quantity of Guests and Managers registered in the system modifies Gas consumption values. Note that, if the experiment from section 6.1.3.2 shows that Görli correctly emulates the Gas consumption, the test network can be used to run the tests instead of using Ethereum *mainnet*. The methodology for this experiment is:

1. Define a methodical set of test cases chronologically arranged – see Table 6.8;
2. Define a fixed set of Ethereum accounts and the actors they correspond to – see Table 6.9;
3. Execute the test cases using the Görli test network instead of Ethereum *mainnet*:
   3.1 If experiment 6.1.3.2 shows that Görli's Gas calculations match those yielded by the *mainnet*;
4. Execution instructions:
   4.1 Record the transaction number;
   4.2 Record the smart contract address;
   4.3 Record the Gas consumption obtained from the Etherscan website.

Table 6.8 – Set of tests for the multiplicity cost experiment

ID	**Short Description**	**Use Case**	**Call Arguments**	**Caller**
CT1	Smart contract deploy	Deploy		Host
CT2	Host adds new Manager	UC5.MSS	Manager 1	Host
CT3	Host adds new Manager	UC5.MSS	Manager 2	Host
CT4	Host adds new Manager	UC5.MSS	Manager 3	Host
CT5	Host adds new Manager	UC5.MSS	Manager 4	Host
CT6	Host adds new Manager	UC5.MSS	Manager 5	Host
CT7	Host adds new Manager	UC5.MSS	Manager 6	Host
CT8	Host adds new Manager	UC5.MSS	Manager 7	Host
CT9	Host adds new Manager	UC5.MSS	Manager 8	Host
CT10	Host adds new Manager	UC5.MSS	Manager 9	Host

Table 6.8 Continue				
ID	**Short Description**	**Use Case**	**Call Arguments**	**Caller**
CT11	Host adds new Manager	UC5.MSS	Manager 10	Host
CT12	Host adds new Guest	UC2.MSS	Guest 1, 1583049600, 1591027200	Host
CT13	Host adds new Guest	UC2.MSS	Guest 2, 1583049600, 1591027200	Host
CT14	Host adds new Guest	UC2.MSS	Guest 3, 1583049600, 1591027200	Host
CT15	Host adds new Guest	UC2.MSS	Guest 4, 1583049600, 1591027200	Host
CT16	Host adds new Guest	UC2.MSS	Guest 5, 1583049600, 1591027200	Host
CT17	Manager adds new Guest	UC2.MSS	Guest 6, 1583049600, 1591027200	Manager 1
CT18	Manager adds new Guest	UC2.MSS	Guest 7, 1583049600, 1591027200	Manager 1
CT19	Manager adds new Guest	UC2.MSS	Guest 8, 1583049600, 1591027200	Manager 1
CT20	Manager adds new Guest	UC2.MSS	Guest 9, 1583049600, 1591027200	Manager 1
CT21	Manager adds new Guest	UC2.MSS	Guest 10, 1583049600, 1591027200	Manager 1
CT22	Host turns exclusive on to the first Manager	UC8.AP2	Manager 1, 1591027200	Host
CT23	Manager exclusive turns it off	UC9.MSS		Manager 1
CT24	Host turns exclusive on to the fifth Manager	UC8.AP2	Manager 5, 1591027200	Host
CT25	Manager exclusive turns it off	UC9.MSS		Manager 5
CT26	Host turns exclusive on to the tenth Manager	UC8.AP2	Manager 10, 1591027200	Host
CT27	Manager exclusive turns it off	UC9.MSS		Manager 10
CT28	Manager turns exclusive on to the first Guest	UC8.AP1	Guest 1, 1591027200	Manager 1
CT29	Guest exclusive turns it off	UC9.MSS		Guest 1
CT30	Manager turns exclusive on to the fifth Guest	UC8.AP1	Guest 5, 1591027200	Manager 1
CT31	Guest exclusive turns it off	UC9.MSS		Guest 5
CT32	Manager turns exclusive on to the tenth Guest	UC8.AP1	Guest 10, 1591027200	Manager 1
CT33	Guest exclusive turns it off	UC9.MSS		Guest 10
CT34	Host removes fourth added Guest	UC3.MSS	Guest 4	Host
CT35	Host removes eighth added Guest	UC3.MSS	Guest 8	Host

Table 6.8 Continue				
ID	**Short Description**	**Use Case**	**Call Arguments**	**Caller**
CT36	Host removes sixth added Guest	UC3.MSS	Guest 6	Host
CT37	Host removes second added Guest	UC3.MSS	Guest 2	Host
CT38	Host removes tenth added Guest	UC3.MSS	Guest 10	Host
CT39	Manager removes third added Guest	UC3.MSS	Guest 3	Manager 5
CT40	Manager removes fifth added Guest	UC3.MSS	Guest 5	Manager 5
CT41	Manager removes ninth added Guest	UC3.MSS	Guest 9	Manager 5
CT42	Manager removes first added Guest	UC3.MSS	Guest 1	Manager 5
CT43	Manager removes seventh added Guest	UC3.MSS	Guest 7	Manager 5
CT44	Host removes sixth added Manager	UC6.MSS	Manager 6	Host
CT45	Host removes fourth added Manager	UC6.MSS	Manager 4	Host
CT46	Host removes ninth added Manager	UC6.MSS	Manager 9	Host
CT47	Host removes second added Manager	UC6.MSS	Manager 2	Host
CT48	Host removes tenth added Manager	UC6.MSS	Manager 10	Host
CT49	Host removes first added Manager	UC6.MSS	Manager 1	Host
CT50	Host removes fifth added Manager	UC6.MSS	Manager 5	Host
CT51	Host removes eighth added Manager	UC6.MSS	Manager 8	Host
CT52	Host removes third added Manager	UC6.MSS	Manager 3	Host
CT53	Host removes seventh added Manager	UC6.MSS	Manager 7	Host

Table 6.9 – Ethereum accounts for the multiplicity case cost experiment

Actor	**Account address**
Host	0xcA18f8947783a38A61a710752673B3f5d0159F6F
Manager 1	0x470b57384Be9C15C9416958D8D35027a0b2a9f30
Manager 2	0x3d3bEcd44835ded70fB3820D3F9AA52aa3b308b3
Manager 3	0xFd2a96AEA8E2C886915a031b80FEb3f099b719eb
Manager 4	0x9847f30610f6F76363Ed5978d40841d89F4DE687
Manager 5	0x71330E718D52b82506c14A18bE625C585F194b01
Manager 6	0x3d1b95E8Dff394bfD428ae1aCA99C3681B2d5263
Manager 7	0x4D71139A5DbC2d7d03e004f26B7B01AaB9EE18c7
Manager 8	0x6CA44286bB7e7841E861BC9CeF912d09C24b9c46
Manager 9	0x9c171798f4794F024e6bF93Bc5EC8766bda4F192
Manager 10	0xB5274295A8820D6E6Bb735D939a776437310C16b
Guest 1	0xc2596913A7283C727DF156f8F3a359c68559bBb2
Guest 2	0xDCdb60D1A30a1335ae0Ed349F8f7A5C62dF70DfF
Guest 3	0x6272803c8D64053ec2e2dF4730e179a47faF3d50
Guest 4	0x38f30f7414287951167a62cab37bB7Fd133FE897
Guest 5	0xDb0Dd99ffd184DcC1aD7C908a485E89A3a054935

Table 6.9 Continue	
Actor	**Account address**
Guest 6	0x67E551Cc0684c761Ab3f16f77C6d069792135B8D
Guest 7	0x133a06D3408518C1c932f29Cf934fcb8568481A2
Guest 8	0x30FBa3d2c2Ec41DBdeEE8506bdaDA71b6241d9dD
Guest 9	0x6443992197f258A7c6f0696E3DaF2Dd522E0cA19
Guest 10	0x59494603B8B16EaB741e9730cf612BF89cC2bC6D

6.2 Results

This section presents and discusses the findings from the performance and cost evaluations defined in section 6.1, following the same order in which the experiments were introduced. Note that all the transactions were recorded in the public network they were executed – either Görli or Ethereum *mainnet* – and they can be viewed using the Etherscan website.

6.2.1 Performance evaluation: The base case experiment

The experiment's procedure defined a set of seventy-five (75) sequential test cases comprised of both transactions and queries to the smart contract. The smart contract address is 0xe4D483036750d386D1f9eee9F401c45b13a2cFcC. The experiment was performed on April 15, 2020, approximately between 9:00 AM to 11:10 AM EDT.

The use case behavior was verified along with each test case execution and all of them met their specification. Therefore, the smart contract functionalities are verified, and it works as expected.

To measure query latency, the submission and response timestamps were provided by the management interface tool. The results are shown in Figure 6.22 – the y-axis is the latency measured in milliseconds, and the x-axis is the test case ID. First, note that all queries get an answer quickly, within a few hundred milliseconds generally. More precisely, approximately eighty (80) percent of the queries replied within one hundred (100) milliseconds. Those results mean that the architecture can deliver reasonable response times to retrieve data from the blockchain, including the unlock function that the door lock hardware relies on.

The two slowest queries – T22 and T23 - are related to retrieving the Guest list. As explained before and illustrated by Figure 6.6, this operation, to be fully completed, needs to query the smart contract G+1 times, where G is the number of Guests registered. Therefore, the higher the number of Guests, the longer it will take to retrieve them along with their permission information as described by the use case UC4. The wait, however, seems insufficient to affect the user experience or compromise the use case success.

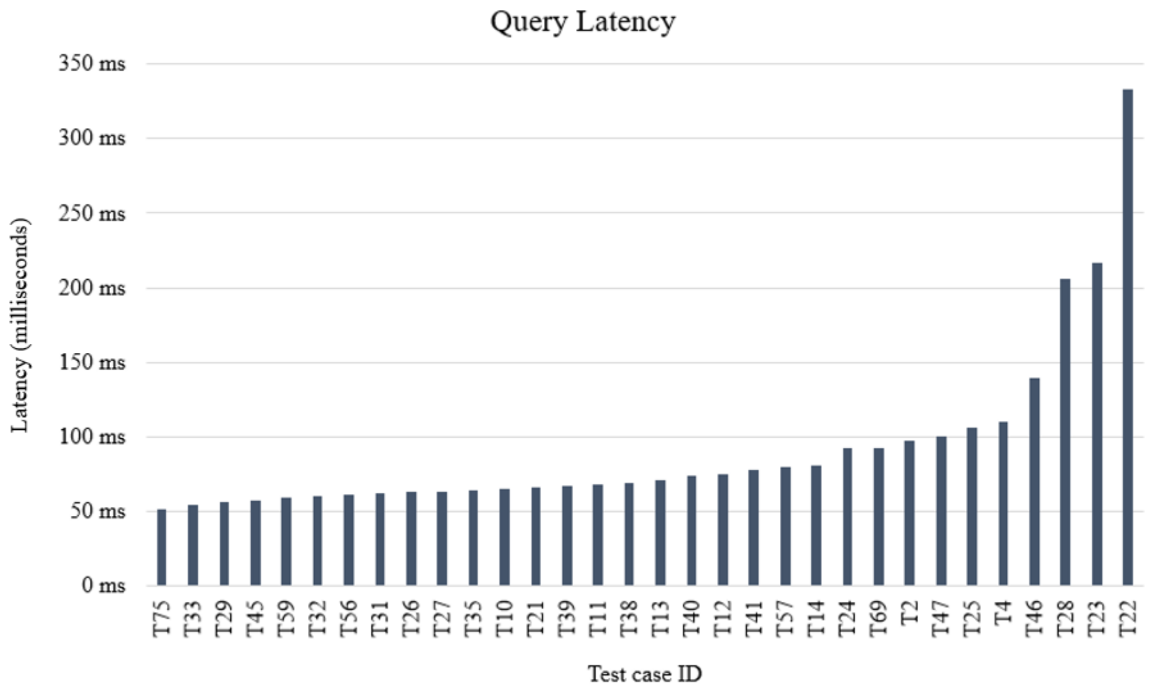


Figure 6.22 – Query latency

To measure transaction (Tx) latency, the submission and confirmation times were acquired through the Metamask wallet log file, downloaded at the end of the experiment. At that moment, however, the log was missing data for the first three transactions of the experiment – i.e., test cases T1, T3, and T5. The wallet erased the records for those transactions apparently due to a capability limit on the account's contract interaction history. Nevertheless, it is possible to verify that all transactions happened, and their respective information, using Etherscan – Table 6.10 brings their hashes. The problem is that Etherscan does not track submission time, it provides only the confirmation time of a transaction. For that reason, the latency for the first three test cases is not included in the results discussed next. It is crucial to note that the test cases T1, T3, and T5 were executed multiple times by other performance experiments and are discussed later in this work. Therefore, their absence here does not harm the proposed evaluation.

Table 6.10 – Excluded transactions from the performance base case analysis

Item	Tx Hash
Tx T1	0x5dcf06331e862bed48de969d2a63a711f311540d84c223ee99a3d4695613b3cf
Tx T3	0x0040715edb8ff93f7b4ffec72ecc65d082bcd9eb72e1c6b1467ba2d3a51ac8b0
Tx T5	0xfd3e8e9a1cf5c435e270f92a15542aac5b2429b51911925bcd9c4d453bcac016

As described by the methodology, the recommendation for the "Fast" Gas price was taken from the Ethereum Gas Station website at the beginning of the experiment, and the suggested value was eight Gwei (8x10^{-9} ETH). The transaction latency results are shown in Figure 6.23 – the y-axis is the latency measured in seconds, and the x-axis is the test case ID. First, observe that except for a single observation, all transactions were confirmed within one minute. As argued before, besides unlocking the door, none of the use cases are considered time-sensitive. Therefore, having a confirmation time under a couple of minutes seems realistic.

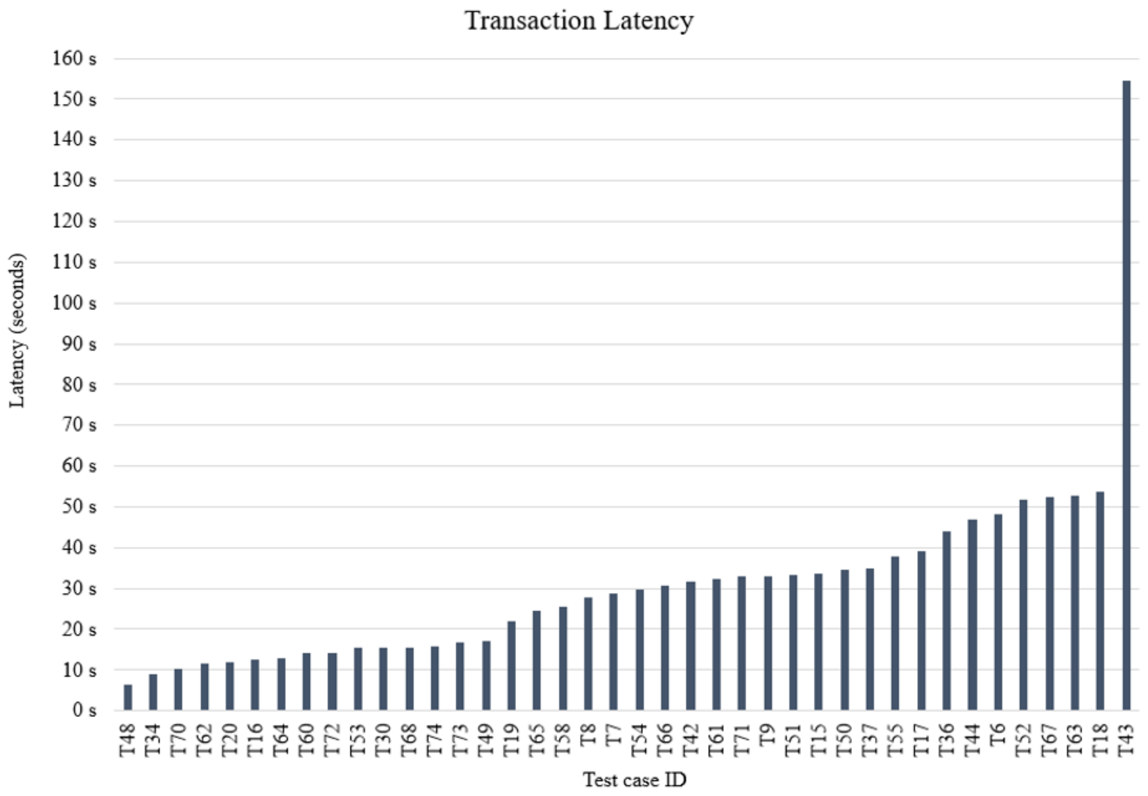


Figure 6.23 – Performance evaluation base case: Transaction latency

Note that Figure 6.23 sort the observations according to the latency values, causing the test case ID's to be completely out of order – remember that those IDs also represent the execution sequence. Considering that the Gas price offered was the same for all the transactions, the chart shows how the network behavior changes dynamically. Looking at this figure alone, however, might lead to an incomplete analysis. Each test case belongs to a different use case scenario and is performed by a different Actor. To draw a complete picture, each of those variables must be investigated for the latency results. Hence, Figure 6.24 shows the same latency chart but including three transaction properties to axis x, namely the use case, the scenario type – i.e., either the main success scenario or an alternative path -, and the Actor that executed it.

The use case and Actor properties do not display any relationship with the transaction latency outcome. However, from the use case scenario type property, it is possible to infer that the main success scenarios (MSS) are typically confirmed faster by the network than the alternative paths (AP). That is a positive result, keeping in mind that many of those alternative paths would not happen often, as mentioned before, once the blockchain wallet would warn the user about the transaction failure in advance.

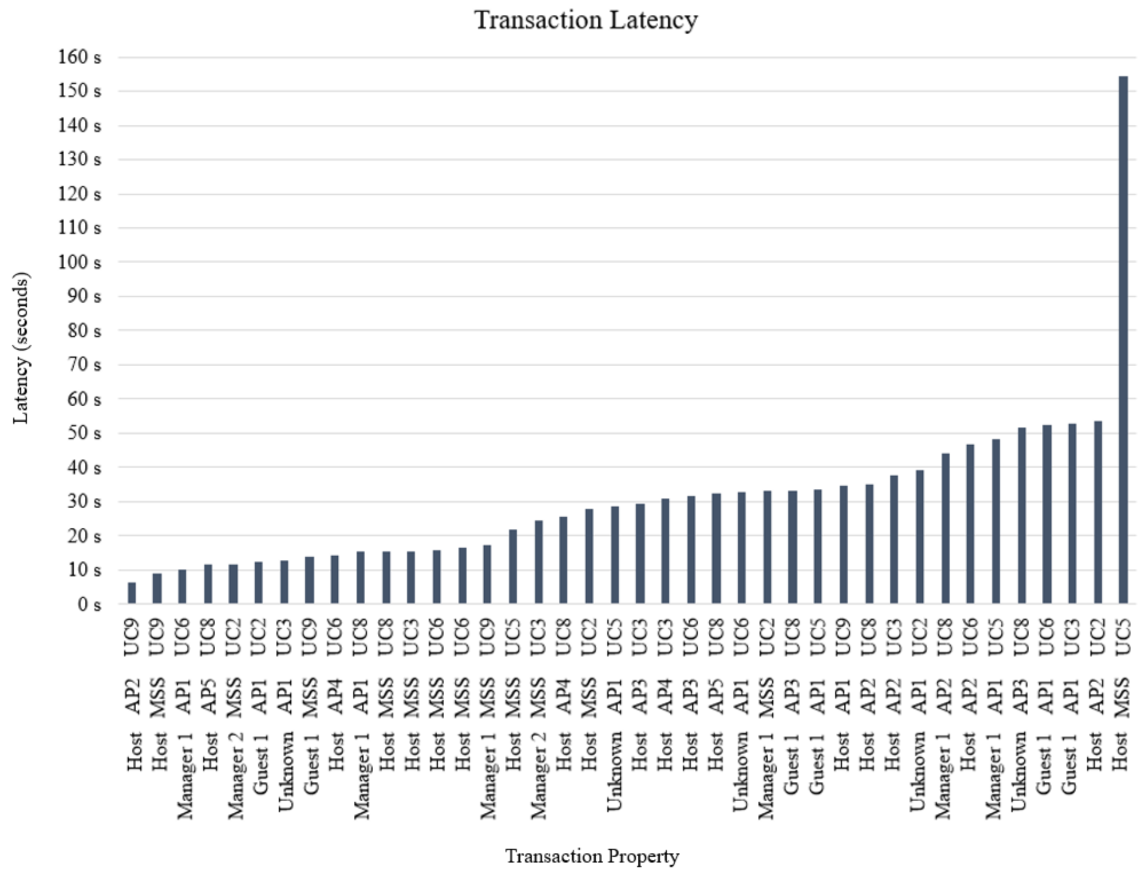


Figure 6.24 – Performance evaluation base case: Transaction latency by its properties

Finally, Figure 6.24 indicates that the Ethereum network behavior changes dynamically and that it might be the primary factor to determine a transaction latency outcome – given a fixed Gas price offer.

Test case T43 shows the longest latency measured, which more than doubles the second-highest value. There are many possible explanations for that behavior. To begin with, the network might have been too busy at that moment, with a high transaction volume, which could cause a higher latency for a transaction offering a Gas price of eight Gwei. It is also possible that the network slowed down the mining process for some consensus reason. Beyond the blockchain behavior, there is a chance that Metamask registered the wrong timestamp in the log file, which could be caused by a bug, by a problem in their backend service, or even by a local network issue at the machine running the experiment.

For the transaction in question, Metamask registered a submission timestamp of 1586960396, and confirmation 1586960551, hence the 155 seconds latency. Inspecting the transaction details in Etherscan – Tx hash 0x922ec2822cf76178deffc9a1fc4d74a07df50248dc2e36fbdaa31a0ee7aa8741 -, the confirmation timestamp registered is 1586960513, 38 seconds earlier than Metamask. Although this value could support the Metamask delay issue hypothesis, it requires finding the confirmation difference between them for all other transactions – see Figure 6.25, where the y-axis is the confirmation timestamp difference in seconds (Metamask minus Etherscan), and the x-axis displays the test case ID. The chart shows that the timestamp confirmation discrepancy can go from six seconds up to forty-nine (49) seconds, and therefore it does not enable drawing a conclusion on the outlier value found for the test case T43 transaction.

The timestamp shown by Etherscan is equal to the block creation timestamp in which the transaction was confirmed, and it shows the value that is recorded in the Ethereum network, not a value they measured locally at their website server. In fact, various Etherscan timestamp values are even lower than the submission timestamp recorded by Metamask, a fact that indicates that they were operating under a distinct clock. On the other hand, this evidence does not explain the high variance found in the timestamp comparison. The best hypothesis is that Metamask relies on a signal to trigger the confirmation event at the local machine and that for some reason – e.g., internet issues - the wallet had some delays to get or process it.

It is crucial to remark that once Etherscan does not track submission timestamps, the blockchain wallet is still the best option to measure transaction latency. In addition to that, the experiments seek to investigate the system's performance under the user's point of view, and the blockchain wallet is the user's gateway to send transactions to the blockchain. Consequently, it strengths the Metamask timestamps as the best choice.

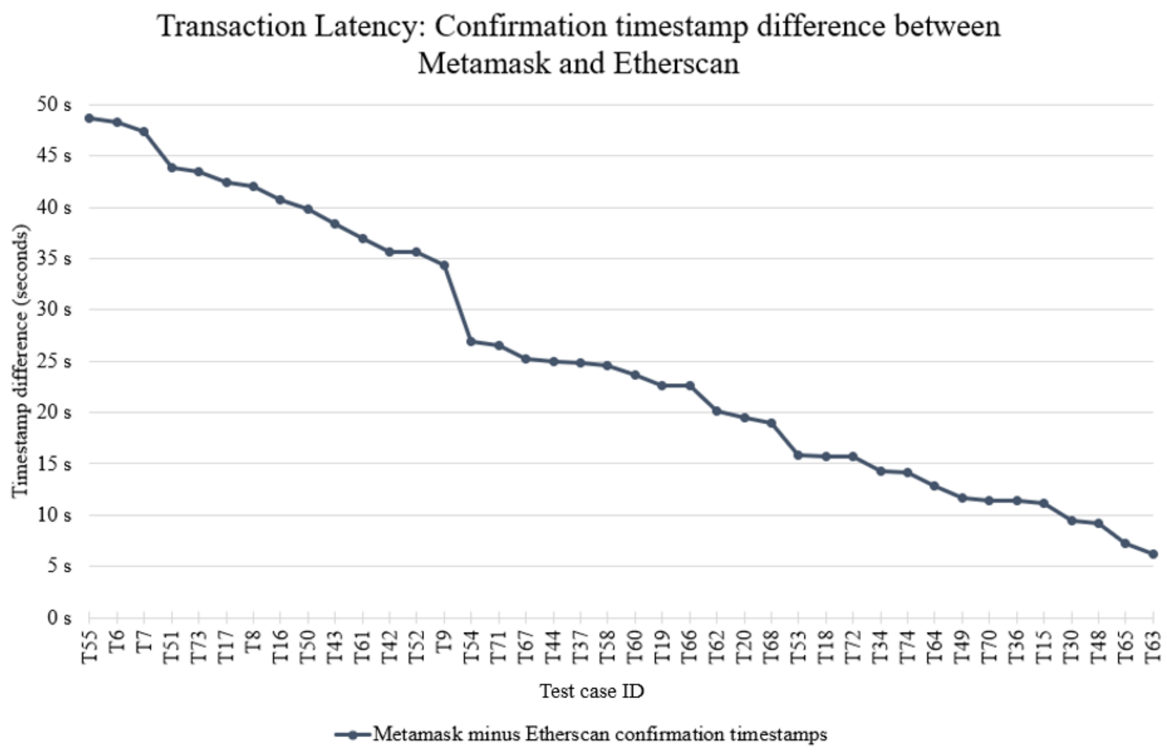


Figure 6.25 – Performance evaluation base case: Confirmation timestamp comparison between Metamask and Etherscan

In summary, the performance experiment base case shows that:

- The smart contract behavior matches the use case requirements;
- All queries to the blockchain have reasonable response time;
- All transactions have a feasible confirmation latency – considering the Gas price set at the "Fast" price recommendation at the beginning of the experiment;
- The Ethereum network performance appears to change constantly;
- The Metamask account log must be downloaded after every transaction performed to avoid data loss;
- Metamask and Etherscan timestamps are neither equivalent nor comparable;
- Although Metamask displays some inconsistencies when measuring latency, it is still the best representation of latency concerning the user experience.

6.2.2 Performance evaluation: The network dynamics experiment

The experiment's procedure defined a set of fifteen (15) sequential test cases, being a subset of those transactions from the base case evaluation. It also established a fixed Gas price offer of eight Gwei (8×10^{-9} ETH) for all transactions, matching the previous experiment price. The test set was executed four times as following:

- On April 16, 2020, from 9:00 AM to approximately 9:55 AM EDT – smart contract address 0xCa0b9f3Cd393D4768Fe060ADAAb2cB8067bC5BD7;
- On April 16, 2020, from 6:00 PM to approximately 6:30 PM EDT – smart contract address 0xc4C068404dA742b169a88dd2E2d30d4b243B29DD;
- On April 19, 2020, from 9:00 AM to approximately 9:30 AM EDT – smart contract address 0x5E349E3D973a876BeAcF6B3EF1A3D9723b595f5C;
- On April 19, 2020, from 6:01 PM to approximately 6:25 PM EDT – smart contract address 0x1B938BA5ac432046Cd9e6277424F68dA75cBACD9.

The latency results are shown in Figure 6.26 – the $\log_2$ scale y-axis is the latency measured in seconds, the x-axis is the test case ID, and each bar represents one of the four executions. To begin with, from a total of sixty (60) transactions, eighty-five (85) percent had a latency time under sixty (60) seconds. Moreover, only a couple of transactions took more than two minutes to get a confirmation, and they both happened on the 9:00 AM EDT procedure from April 16. As mentioned before, those values are realistic to the use cases and support the proposed architecture.

In addition to that, Figure 6.26 displays the dynamic changes that happen in the Ethereum network. The same transaction, offering the same Gas price, performed in a different day, or time of the day, yielded various latency values, without any consistency. Figure 6.27 brings an alternative view to that chart that compares transaction latency between the four executions – the y-axis is the latency share, i.e., how much the given latency amount represents of the test case's total, and the x-axis is the test case ID. Observe that although the first test set run needed almost twice the time to complete than the following three trials, only one-third of their transactions took longer to confirm than their peers' – namely T1, T8, T30, T65, and T73.

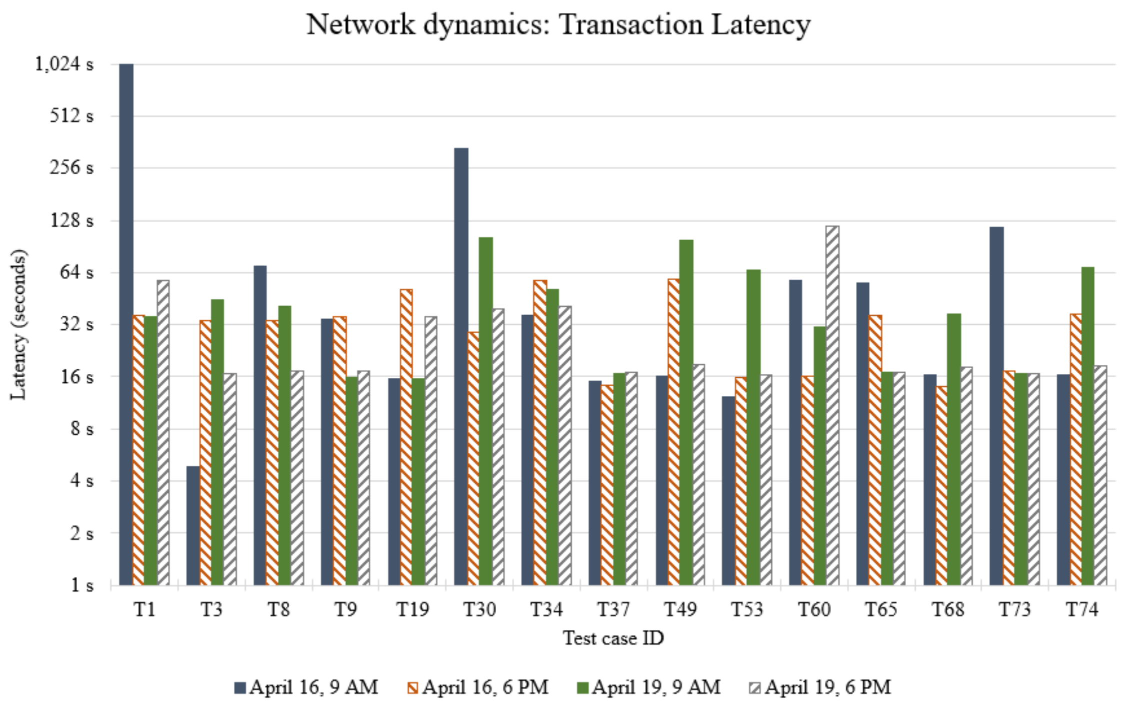


Figure 6.26 – Network dynamics performance evaluation: Latency

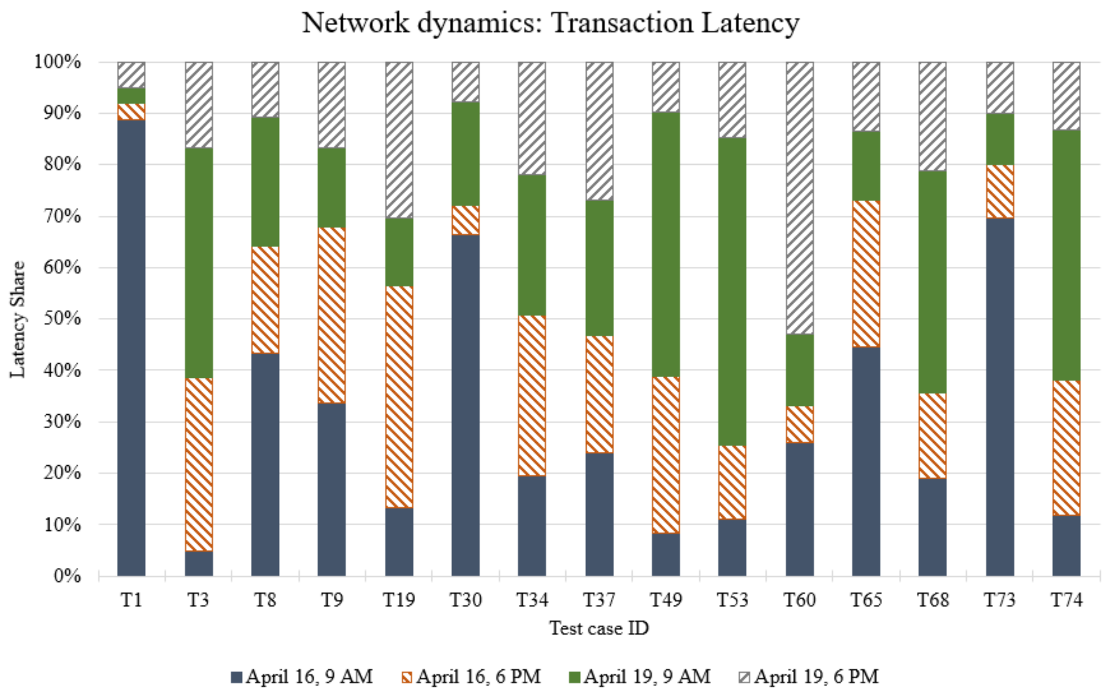


Figure 6.27 – Network dynamics performance evaluation: Latency share

The transaction with the overall highest latency is from test case T1, the smart contract deploy on April 16 at 9:00 AM EDT – Tx hash 0xd208aaec4c29065ad2f9eb3557125092aa1519b6e0132e790bd822d65c01d61e. Although

Metamask and Etherscan timestamps cannot be compared as explained earlier, it can provide at least an evidence that indicates if either Metamask had any issue to log the confirmation, or the network took more time to process it. Furthermore, since that is the transaction that started the evaluation, its submission time is established exactly at 9:00 AM EDT, which takes the Metamask submission time out of the equation. According to Etherscan, that transaction was confirmed at 9:16:50 AM EDT – i.e., 1010 seconds -, which is close to the 1014 seconds yielded by Metamask. Therefore, it is possible to affirm that the network indeed took more time to confirm that transaction. However, the reason does not seem to be related to the test case action of deploying the smart contract once its three other executions took Ethereum under sixty (60) seconds to confirm.

The transaction with the second-highest latency is from test case T30, also on the April 16 at 9:00 AM EDT execution – Tx hash 0x3cc78d5045ef86a3d9d885f8e08d9c2e2c16904e8dc16e69f97235cfd68c284a. Etherscan shows the confirmation at 9:34:26 AM EDT, while Metamask logged it at 9:34:32 AM EDT, extremely close once again. Even though there is no submission time to rely upon besides the Metamask log, it is feasible to infer that the network actually took more time to process that transaction. And, similarly as before, it does not seem to be related to the test case action when the three other observations for T30 are considered - twenty-nine (29), one hundred and one (101), and thirty-nine (39) seconds respectively.

In summary, the network dynamics performance experiment shows that:

- The Ethereum network performance changes constantly;
- Following the "Fast" Gas price from the base case experiment, eighty-five (85) percent of the transactions were confirmed in less than one minute, whereas ninety-seven (97) percent were processed in less than two minutes.

6.2.3 *Performance evaluation: The Gas price experiment*

The experiment's procedure defined a set of seven sequential test cases. Each of them was executed three times, offering three different prices of Gas – three, six, and twelve (12) Gwei. The smart contracts created by the deploy test case GT1 for each Gas price category were:

- Low price – contract address 0x1e9769e79a656ef48093273c65ca229670a4bb62;

- Medium price – contract address 0x6c625f3bf4cfa4c75b91a6d7135ab8888c03e3fb;
- High price – contract address 0x3173ae8AE0fDc8AE1C9aD662e24E0BC62cD01Bb8.

To compare latency results between the Gas price offers, the transactions for a given test case were sent in the order defined – from the lowest to the biggest offer -, and as soon as possible, to minimize the network dynamics effect.

However, the Metamask wallet could not handle quick interactions and slowed down to show the next transaction for approval and to change between the Actors' account afterward, causing an undesired delay between submissions. Figure 6.28 shows the amount of time elapsed between each test case transaction and the corresponding low gas price submission – the y-axis is the test case Tx submission time for the respective Gas price minus the low Gas price Tx submission time – i.e., the delay -, and the x-axis is the test case ID. Unfortunately, the delays between the steps were too long, generally over forty (40) seconds, compromising the proposed analysis. The reason is that, due to those long intervals, it is not reasonable to isolate the network dynamics impact on the results.

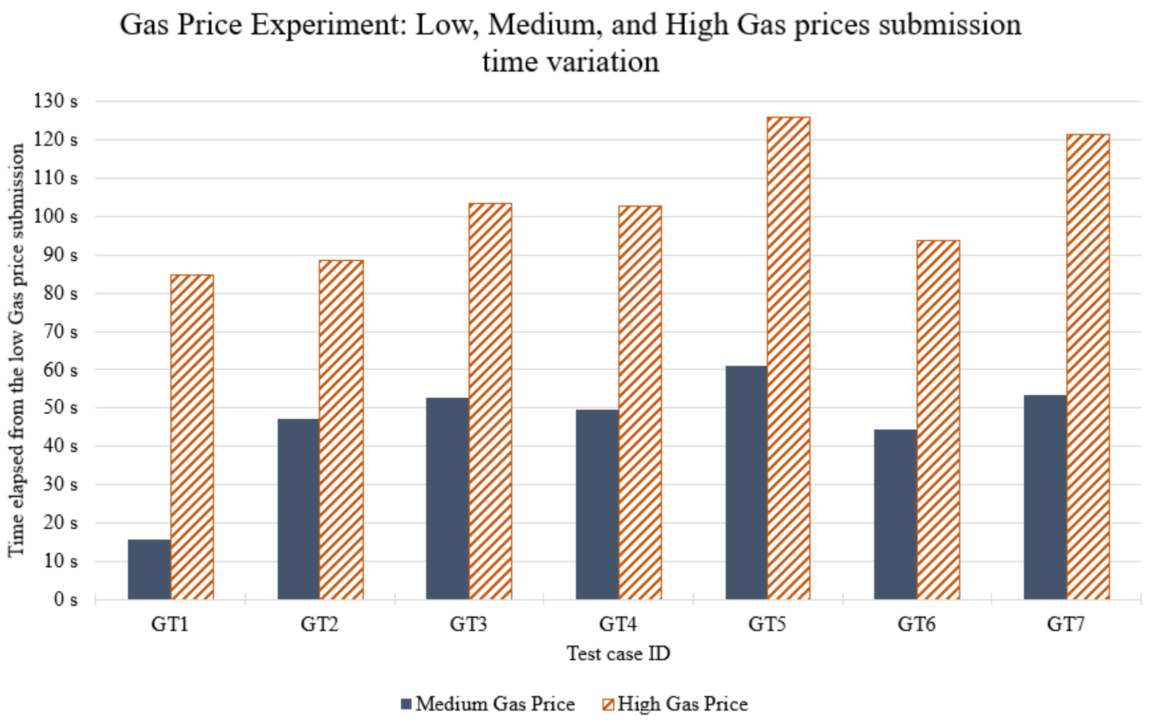


Figure 6.28 – Gas price performance evaluation: Time elapsed between Tx submissions

Although it is not feasible to draw exact conclusions on the Gas price and latency relationship, the data still offers some insights about the network behavior in general. It is crucial to highlight that, contrary to the base case and network dynamics evaluations, the test cases of this experiment did not happen on the same day. Figure 6.29 shows the latency results – the $\log_2$ y-axis is the latency measured in seconds, the x-axis is both the test case ID and the respective low Gas Tx submission date (EDT). Each bar in the chart represents a Gas price offer.

First, note that all transactions offering the high Gas price were confirmed under one minute. Second, note that both test cases that started in the morning – GT1 and GT4 – display extremely high latency for both medium and low Gas prices. Considering that those values are significantly higher than the time elapsed between their submissions, it is possible to infer that the network was busy at that moment, which led to meaningful latency discrepancies based on the Gas offer. Moreover, observe that for most of the other test cases, the latency did not exhibit an expressive disparity.

Finally, those results suggest that high Gas price offers can keep the latency in a reasonable value regardless of the network condition. Whereas lower offers might yield equally low latencies but are more vulnerable to network changes.

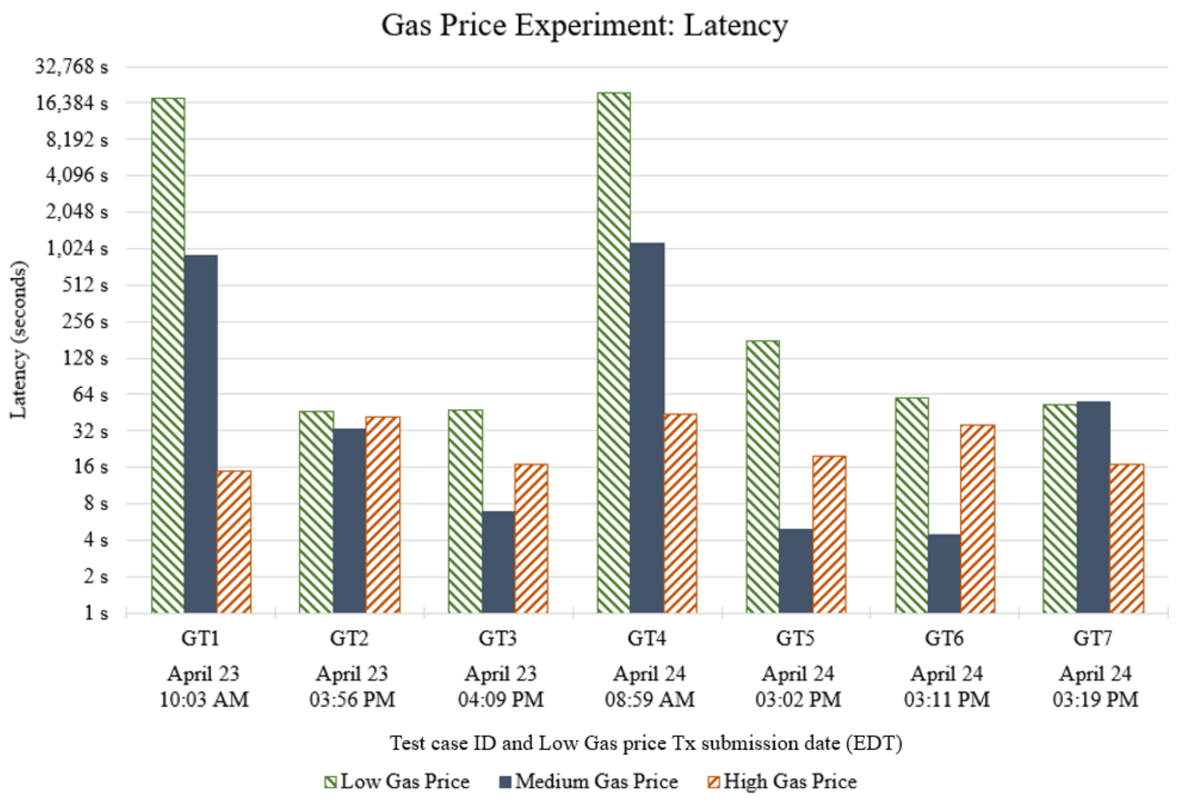


Figure 6.29 – Gas price performance evaluation: Latency

6.2.4 *Cost evaluation: The base case experiment*

The experiment procedure did not require any additional testing, because all the transactions from previous experiments yielded the amount of Gas required to execute them. Although some of those transactions offered different Gas prices for the same test case, the Gas consumption is the same, only the final cost in ETH – Gas used multiplied by the Gas price – is affected.

Note that the Gas price choice directly impacts the transaction cost and, as shown before, its resulting latency. As a consequence, the Gas price used for this analysis is twelve (12) Gwei, because it was consistently able to get transactions confirmed under one minute.

As defined by the procedure, the ETH cost in CAD must be obtained from the Coinbase website, and, as of April 29, 2020, at 02:08 PM, the value of 1 ETH was CAD 296.75.

Figure 6.30 shows the Gas consumption of the performance experiment base case transactions – the $\log_2$ left y-axis is the unit of Gas used by the transaction, the $\log_2$ right y-axis is the respective cost in CAD, and the x-axis is the test case ID with the corresponding use case ID.

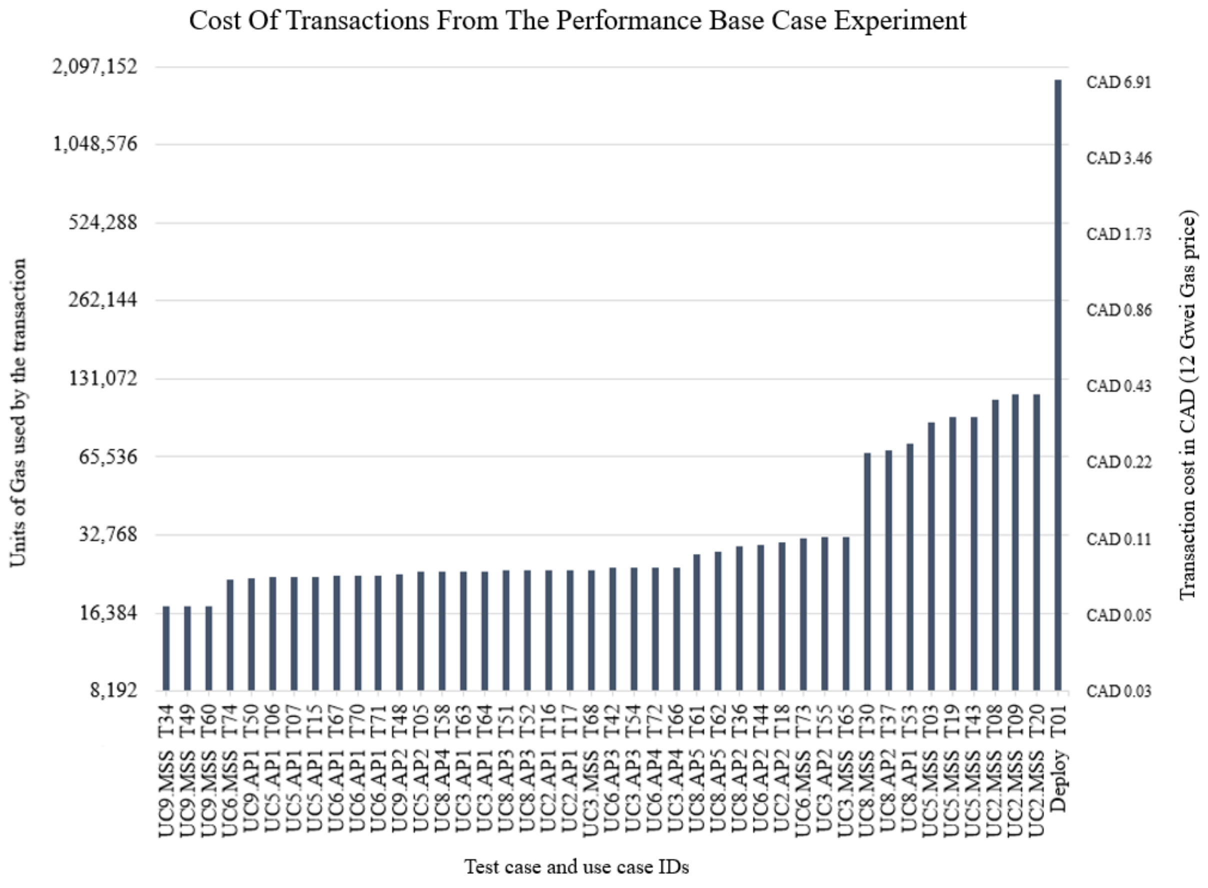


Figure 6.30 – Cost evaluation: Performance base case Gas consumption

First, observe that the most expensive operation is the smart contract deploy – test case T01 – costing CAD 6.63 (1,862,258 Gas units). Although a relevant sum of money, the deploy is a one-time-only action. Besides that, all other transactions yielded costs below forty (40) cents, where the second most expensive use case is register Guest (UC2.MSS), followed by register Manager (UC5.MSS), and turn on the exclusive permission (UC8.MSS, AP1, and AP2).

In general, those costs are believed to be too low to be prohibitive, especially considering the application context. For example, when someone needs to constantly alter the smart contract, it could mean that multiple rents are happening, which can dilute the costs to manage the door lock. On the other hand, a domestic setting would hardly demand constant changes, keeping those expenses low and sparse over time.

The following sections expand the cost analysis for each use case, including results from the two other performance evaluations.

6.2.4.1 Deploy

The smart contract deploy was executed a total of eight times, from different Actor accounts, and consistently yielded a Gas consumption of 1,862,258 Gas units, which amounts to CAD 6.63 - considering the Gas price at 12 Gwei. For a one time only operation, the value seems reasonable considering the application context. Take into consideration that if low latency is not required, that price can drop significantly – i.e., CAD 1.66 at three Gwei.

6.2.4.2 UC2 – Register Guest

Figure 6.31 shows all seventeen (17) transactions related to registering a Guest – the y-axis is the units of Gas used, the x-axis is the test and use cases IDs, and above each column is the respective cost in Canadian Dollars.

There are five different Gas consumption values, though one of them is not visible in the chart. At the same time, note that among the same test case, the Gas consumption was consistently the same. To support the discussion, Figure 6.32 reveals the implementation for the *registerGuest* function – see sequence diagrams 6.1.3.2 and 6.1.3.3 for details.

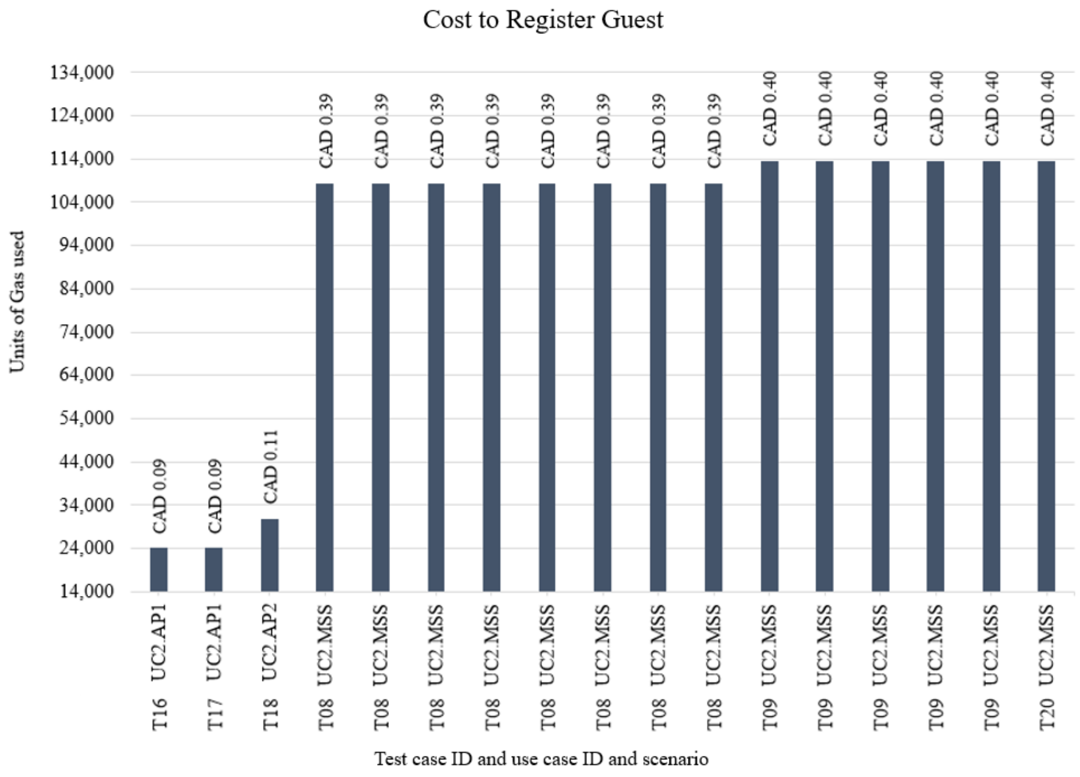
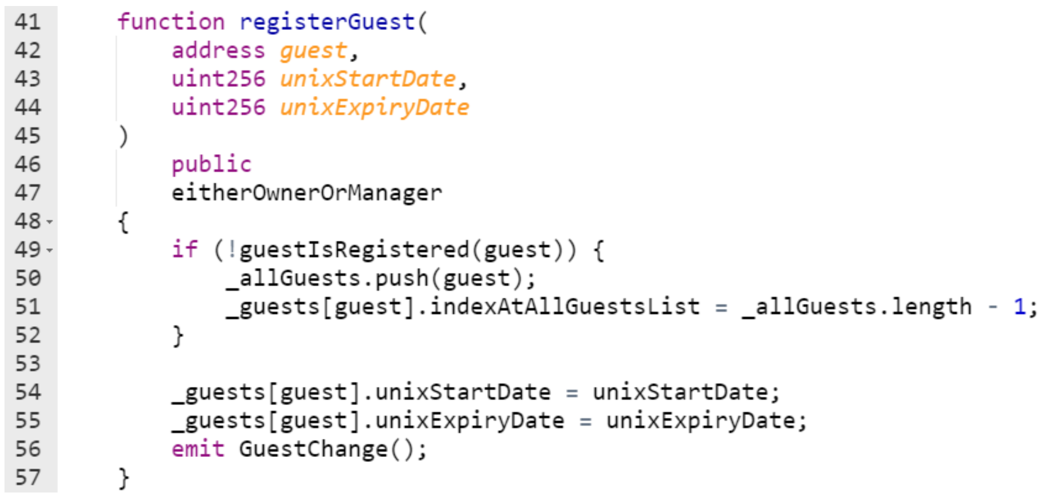


Figure 6.31 – Cost evaluation: Register Guest



```
function registerGuest(
    address guest,
    uint256 unixStartDate,
    uint256 unixExpiryDate
)
    public
    eitherOwnerOrManager
{
    if (!guestIsRegistered(guest)) {
        _allGuests.push(guest);
        _guests[guest].indexAtAllGuestsList = _allGuests.length - 1;
    }

    _guests[guest].unixStartDate = unixStartDate;
    _guests[guest].unixExpiryDate = unixExpiryDate;
    emit GuestChange();
}
```

Figure 6.32 – Function *registerGuest* implementation

Starting from UC2.AP1, this alternative path represents a failed transaction – e.g., someone other than the Host or a Manager trying to add a Guest. Hence, only the identity check at the *eitherOwnerOrManager* modifier happens, and nothing is stored in the smart contract, keeping those costs low.

Use case UC2.AP2 is an alternative path where the Host registers a Guest that is already registered, in which case only updates the start and expiry dates – it skips lines 50 and 51. Therefore, it requires more computation than UC2.AP1, but less than to register a new Guest.

Next, the other three values are 108,424 (T08), 113,591 (T09), and 113,615 (T20) Gas units, where the second and third amounts represent approximately a five percent increase from T08. Two main characteristics distinguish between those test cases: (i) the Host registers the Guest in T08, while the Manager does it in both other cases; (ii) the Host always register the first Guest (T08), followed by the Manager adding the second (T09) and third Guests (T20).

To allow the registration, the function initially verifies the caller's identity using the *eitherOwnerOrManager* modifier. The process to verify if the address sending the transaction is a Manager involves more computational steps than what is required to check the Host identity, consequently rising the cost – see Figure 6.33. This explains the extra charge from T08 to T09.

On the other hand, the cost to add the third Guest (T20) had a subtle increase from T09, which can be explained by either the push method slightly raising the cost according to the array size, or by the different Guest address, start date, and expiry date used.

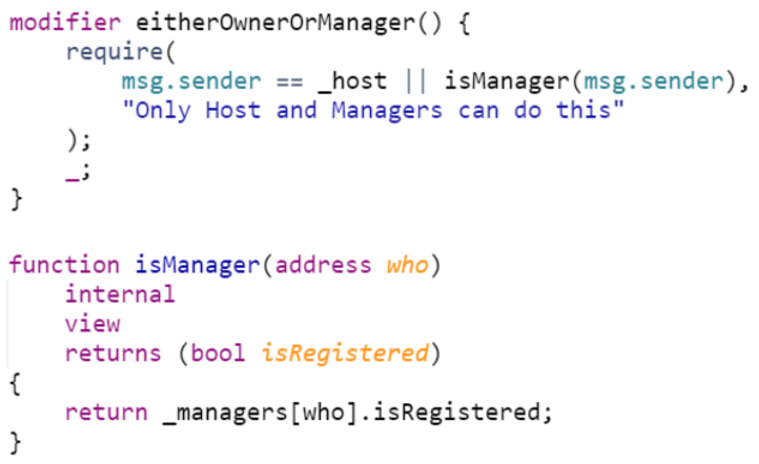

```
modifier eitherOwnerOrManager() {
    require(
        msg.sender == _host || isManager(msg.sender),
        "Only Host and Managers can do this"
    );
    _;
}

function isManager(address who)
    internal
    view
    returns (bool isRegistered)
{
    return _managers[who].isRegistered;
}
```

Figure 6.33 – Modifier *eitherOwnerOrManager* and function *isManager* implementations

6.2.4.3 UC3 – Remove Guest

Figure 6.34 shows all eighteen (18) transactions related to remove a Guest – the y-axis is the units of Gas used, the x-axis is both the test and use cases IDs, and above each column is the respective Dollar amount.

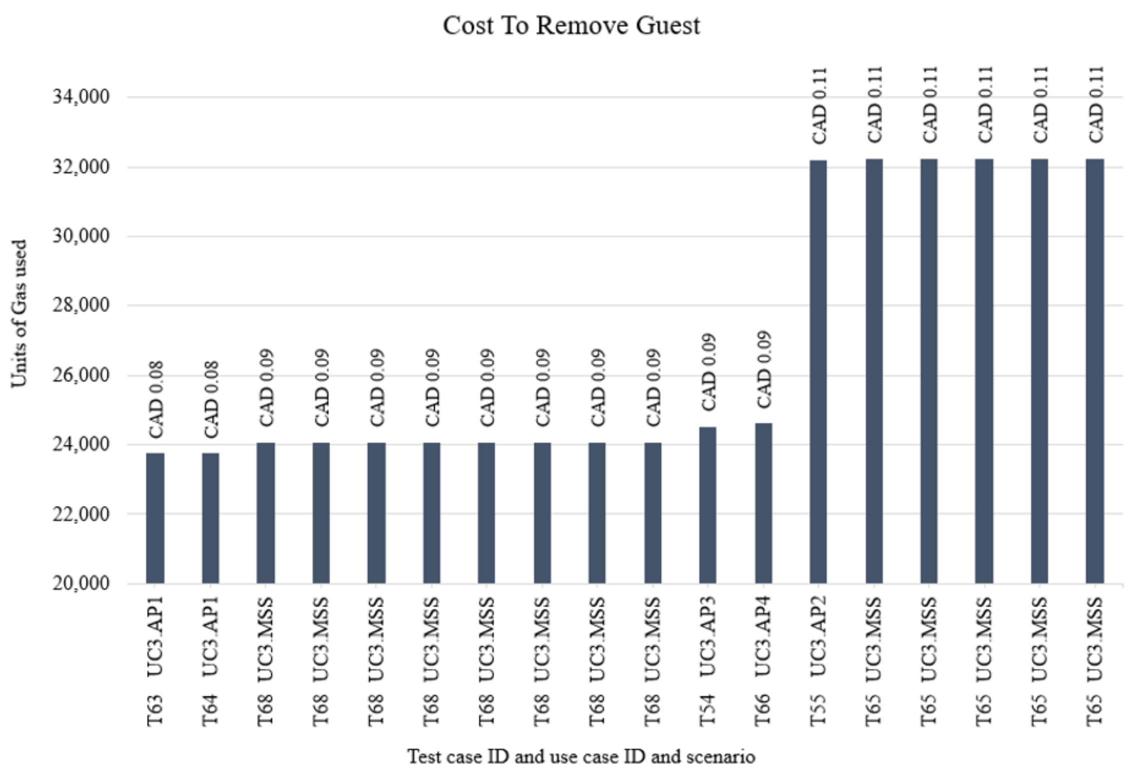


Figure 6.34 – Cost evaluation: Remove Guest

There are six different Gas consumption values, though two of them are not distinguishable looking at the chart – T66 and T65. On the other hand, among the same test case, Gas consumption was consistently equal. To support the discussion, Figure 6.35 reveals the implementation for the *removeGuest* function – see sequence diagrams 6.1.3.4 and 6.1.3.5 for details.

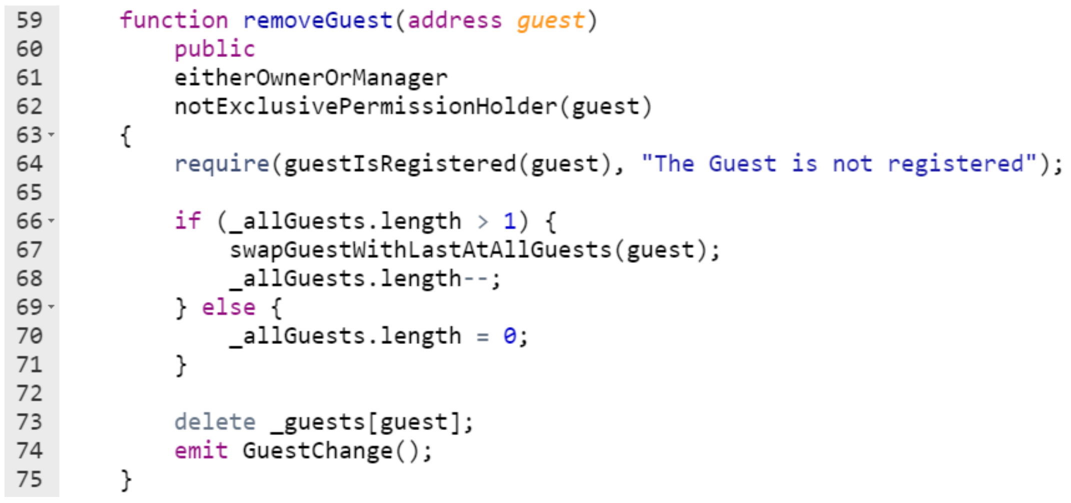

```
function removeGuest(address guest)
    public
    eitherOwnerOrManager
    notExclusivePermissionHolder(guest)
{
    require(guestIsRegistered(guest), "The Guest is not registered");

    if (_allGuests.length > 1) {
        swapGuestWithLastAtAllGuests(guest);
        _allGuests.length--;
    } else {
        _allGuests.length = 0;
    }

    delete _guests[guest];
    emit GuestChange();
}
```

Figure 6.35 – Function *removeGuest* implementation

Similarly to UC2.AP1, UC3.AP1 represents a failed transaction – e.g., someone other than the Host or a Manager trying to remove a Guest, which stops the execution at the identity check. UC3.AP3 also fails because the Host is trying to remove a Guest that holds the exclusive permission, stopping at the second function modifier. The last failing use case is UC3.AP4, where the Host tries to remove someone not registered as a Guest, error identified at line 64, aborting the execution. Therefore, those three failed transactions use distinct amounts of Gas as they quit the function at different points.

Observe that UC3.MSS is executed by two test cases, T65 and T68. Two main characteristics distinguish between those test cases: (i) The Host removes the Guest in T68, while the Manager does it in T65; (ii) T65 removes one of two Guests registered, and T68 removes the last one.

As explained before, the identity check for a Manager requires more work than verifying a Host, consequently using more units of Gas. Furthermore, the function *removeGuest* implementation shows that, when the array of Guests has only one address stored, lines sixty-seven (67) and sixty-eight (68) are not executed. Thus, both aspects have an impact on the resulting cost.

Finally, test case T55 costs slightly less than T65, and both remove a Guest with the respective array size larger than one. In this case, however, the Host executes the action instead of the Manager, and it happens while the exclusive permission is turned on to another Guest. Therefore, there is less work to check the identity, but more effort to verify that the Guest is not the exclusive permission holder.

6.2.4.4 UC5 – Register Manager

Figure 6.36 shows all eighteen (18) transactions related to registering a Manager – the y-axis is the units of Gas used, the x-axis is both the test and use cases IDs, and above each column is the respective cost in Canadian Dollars.

There are four different Gas consumption values but among the same test case, the Gas consumption was consistently the same. To support the discussion, Figure 6.37 reveals the implementation for the *registerManager* function – see sequence diagrams 6.1.3.8 and 6.1.3.9 for details.

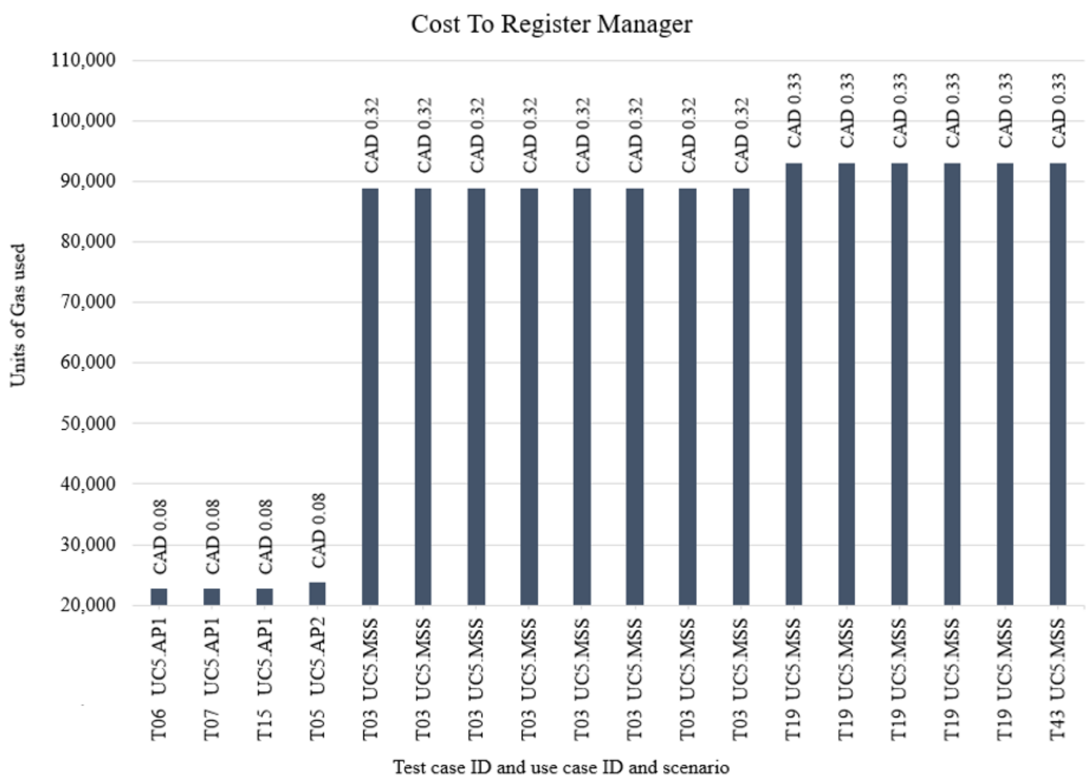


Figure 6.36 – Cost evaluation: Register Manager

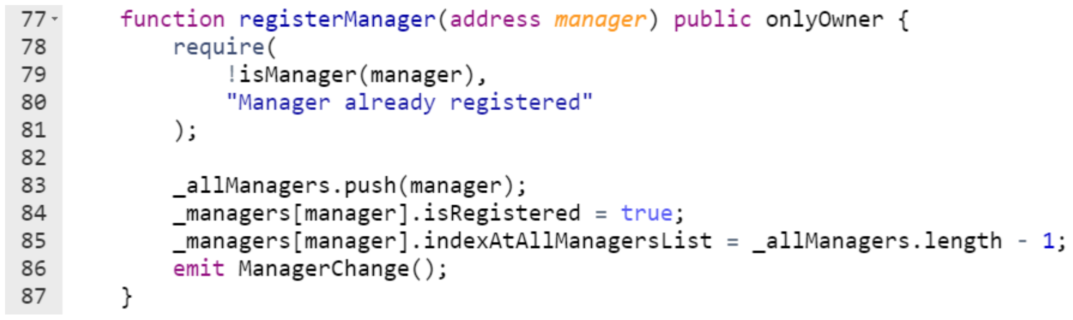

```
77    function registerManager(address manager) public onlyOwner {
78        require(
79            !isManager(manager),
80            "Manager already registered"
81        );
82
83        _allManagers.push(manager);
84        _managers[manager].isRegistered = true;
85        _managers[manager].indexAtAllManagersList = _allManagers.length - 1;
86        emit ManagerChange();
87    }
```

Figure 6.37 – Function *registerManager* implementation

Similarly to previous discussions, UC5.AP1 and UC5.AP2 are prohibited actions that result in failed transactions. The former is someone other than the Host trying to register a Manager, and the latter is the Host trying to add an existing Manager. Their subtle Gas consumption discrepancy relates to where the code execution stops, one at the *onlyOwner* modifier and the other at line seventy-nine (79).

Test cases T03, T19, and T43 executes the use case UC5.MSS, and all of them have the Host adding a new Manager to the smart contract. Test case T43 adds a third Manager while the exclusive permission feature is turned on to someone, but this is not relevant to this function – i.e., it does not check anything related to that as shown by the implementation.

Finally, the first Manager registration consumes slightly fewer Gas units than the following additions, suggesting that the array size indeed alters the push method cost.

6.2.4.5 UC6 – Remove Manager

Figure 6.38 shows all nineteen (19) transactions related to remove a Manager – the y-axis is the units of Gas used, the x-axis is both the test and use cases IDs, and above each column is the respective Dollar amount.

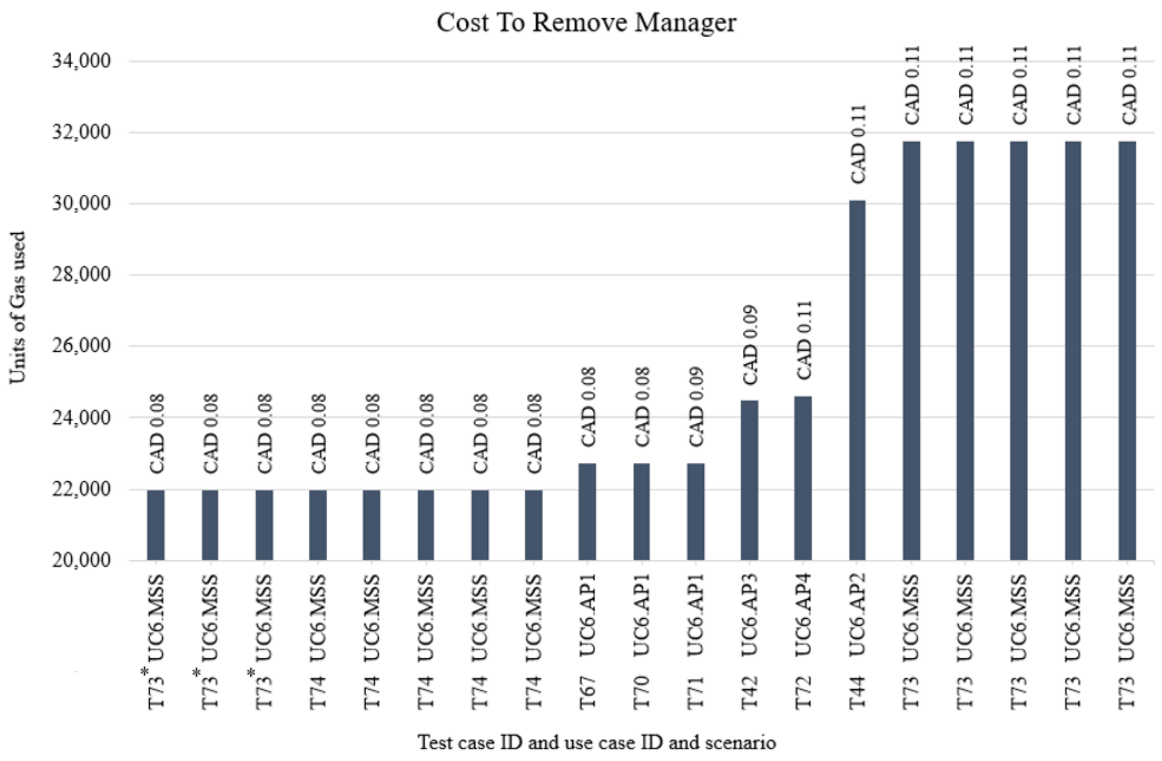
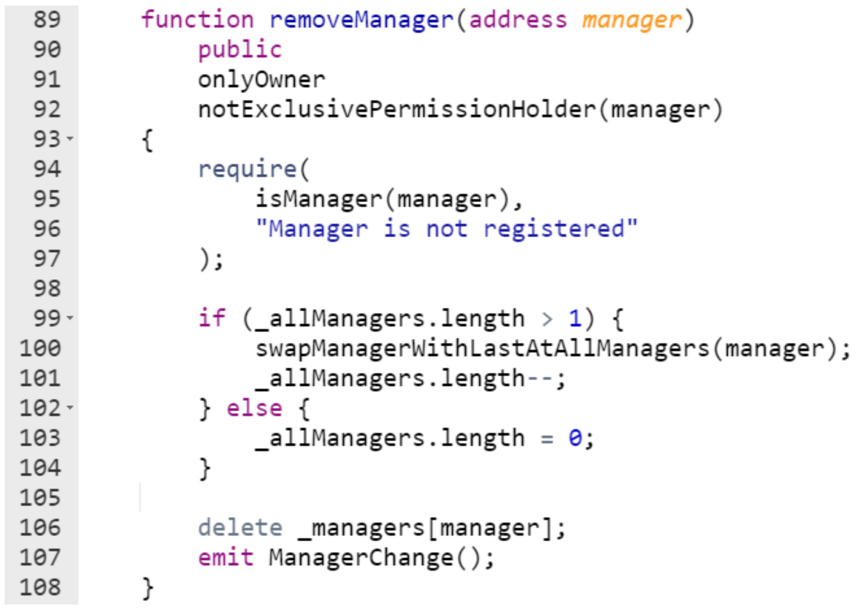


Figure 6.38 – Cost evaluation: Remove Manager

There are six different Gas consumption values, though one of them is not recognizable looking at the chart – T72. For the first time, the same test case yielded two distinguished costs, namely T73. However, the cheapest T73 transactions, identified with a * symbol, were executed in the Gas price performance experiment, and, in that case, only a single manager was registered in the smart contract. Those expensive T73 transactions were removing the second to last Manager in both other experiments. Therefore, the test cases T73* are, in fact, equivalent to T74 of the remaining experiments. Consequently, Gas consumption is consistent among the same test cases as expected.

Figure 6.39 reveals the implementation for the *removeManager* function – see sequence diagrams 6.1.3.10 and 6.1.3.11 for details. Since the *removeManager* implementation, use cases, and cost behavior are highly similar to *removeGuest*, their cost analyses are also equivalent. For that reason, no further discussion is necessary here.



```
function removeManager(address manager)
    public
    onlyOwner
    notExclusivePermissionHolder(manager)
{
    require(
        isManager(manager),
        "Manager is not registered"
    );

    if (_allManagers.length > 1) {
        swapManagerWithLastAtAllManagers(manager);
        _allManagers.length--;
    } else {
        _allManagers.length = 0;
    }

    delete _managers[manager];
    emit ManagerChange();
}
```

Figure 6.39 – Function *removeManager* implementation

6.2.4.6 UC8 – Turn on exclusive permission

Figure 6.40 shows all twenty-four (24) transactions related to turning on the exclusive feature – the y-axis is the units of Gas used, the x-axis is both the test and use cases IDs, and above each column is the respective cost in Canadian Dollars.

There are eight different Gas consumption values, though one of them is not distinguishable looking at the chart – T51. On the other hand, among the same test case, Gas consumption was consistently the same. To support the discussion, Figure 6.41 reveals the implementation for the *turnExclusiveFeatureOn* function – see sequence diagrams 6.1.3.14 and 6.1.3.15 for details.

Similar to previous discussions, some alternative paths represent prohibited actions that result in failed transactions, therefore consuming small amounts of Gas. In this case, those scenarios are: (i) UC8.AP4, where the Host tries to turn the feature on when it is already on; (ii) UC8.AP3, where someone other than the Host or a Manager tries to turn the feature on; (iii)

UC8.AP5, when the Host tries to turn the feature on to someone without permission to unlock the door; (iv) UC8.AP2, where a Manager tries to turn the feature on to itself.

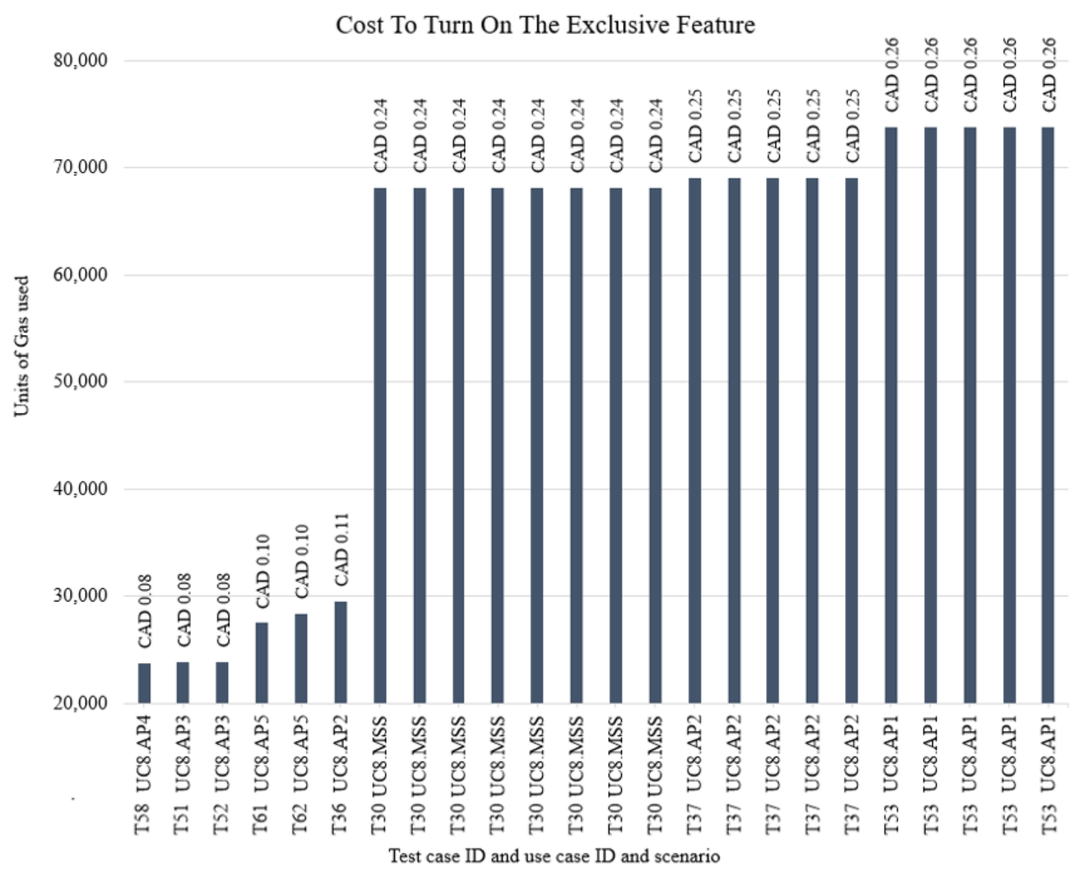


Figure 6.40 – Cost evaluation: Turn on exclusive permission

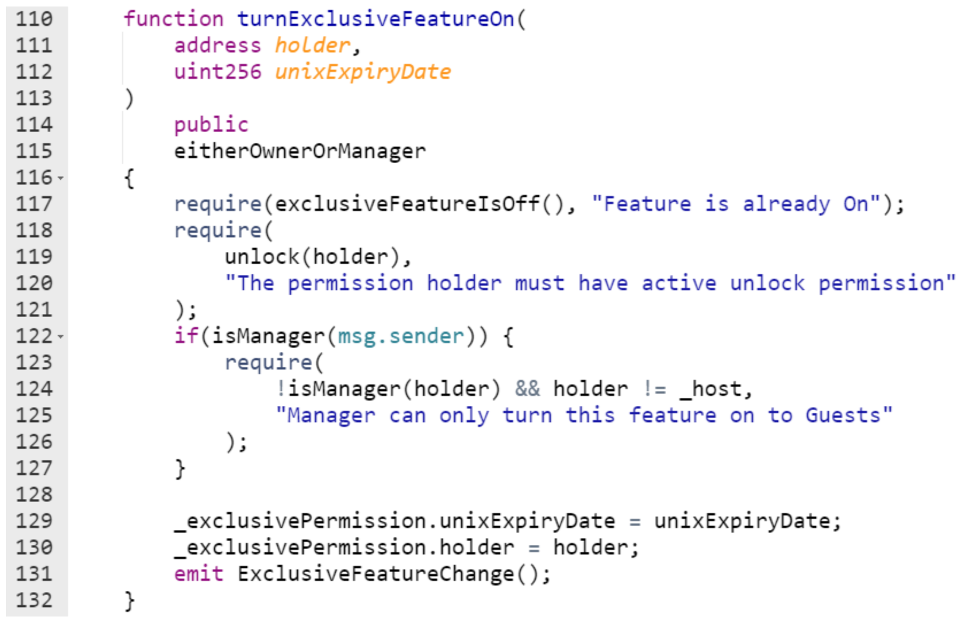

```
function turnExclusiveFeatureOn(
    address holder,
    uint256 unixExpiryDate
)
    public
    eitherOwnerOrManager
{
    require(exclusiveFeatureIsOff(), "Feature is already On");
    require(
        unlock(holder),
        "The permission holder must have active unlock permission"
    );
    if(isManager(msg.sender)) {
        require(
            !isManager(holder) && holder != _host,
            "Manager can only turn this feature on to Guests"
        );
    }

    _exclusivePermission.unixExpiryDate = unixExpiryDate;
    _exclusivePermission.holder = holder;
    emit ExclusiveFeatureChange();
}
```

Figure 6.41 – Function *turnExclusiveFeatureOn* implementation

Use case UC8.MSS is the Host turning the feature on to itself, whereas in UC8.AP2 the Host turns it on to a Manager, both having the same expiry date. Their resulting cost differs by only 995 units of Gas can be attributed to the *unlock* method call to verify if the desired exclusive permission holder can open the lock. Once again, a Manager requires more steps than the Host.

Test case T53 is performed by a Manager, turning the feature on to a Guest. Besides the usual extra work to check the identity and the *unlock* method call to a Guest – the most complex stakeholder to validate the ability to unlock the device -, it requires an extra validation of conditions – refer to line one hundred and twenty-four (124). Hence the highest cost.

6.2.4.7 UC9 – Turn off exclusive permission

Figure 6.42 shows all twenty (20) transactions related to turning off the exclusive permission feature – the y-axis is the units of Gas used, the x-axis is the test and use cases IDs, and above each column is the respective Dollar amount.

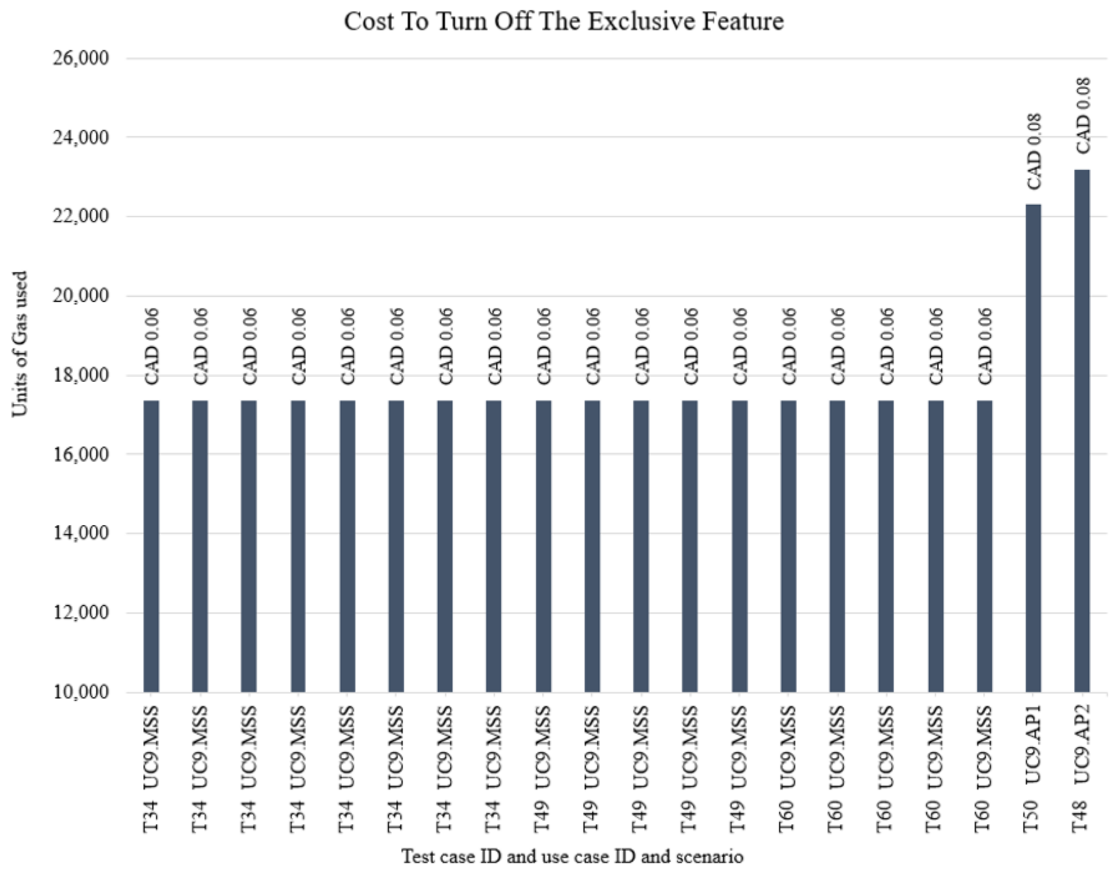
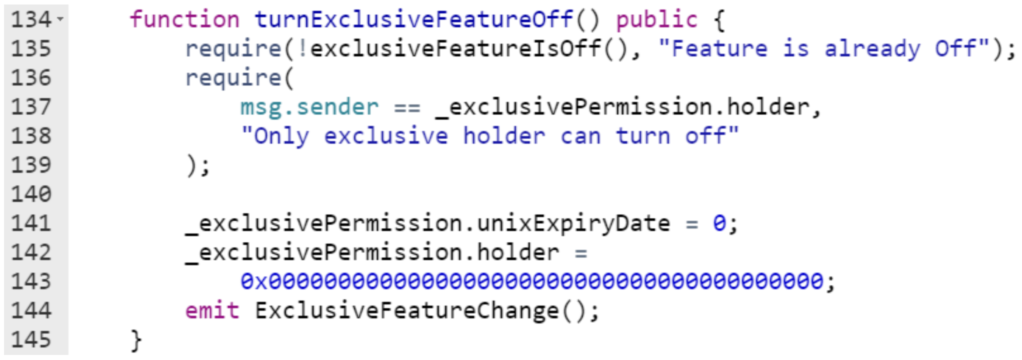


Figure 6.42 – Cost evaluation: Turn off exclusive permission

There are three different Gas consumption values but among the same test case, the Gas consumption was consistently equal. To support the discussion, Figure 6.43 reveals the implementation for the *turnExclusiveFeatureOff* function – see sequence diagrams 6.1.3.16 and 6.1.3.17 for details.



```
function turnExclusiveFeatureOff() public {
    require(!exclusiveFeatureIsOff(), "Feature is already Off");
    require(
        msg.sender == _exclusivePermission.holder,
        "Only exclusive holder can turn off"
    );

    _exclusivePermission.unixExpiryDate = 0;
    _exclusivePermission.holder =
        0x0000000000000000000000000000000000000000;
    emit ExclusiveFeatureChange();
}
```

Figure 6.43 – Function *turnExclusiveFeatureOff* implementation

Use case UC9.MSS, where the permission holder successfully turns the feature off, is executed by three test cases, T34, T49, and T60. The distinction between them is who the permission holder is and, consequently, who is calling the function, respectively the Host, a Manager, and a Guest.

However, note that this function is the simplest in terms of access control among all others discussed. As long as the person calling holds the exclusive permission, the function runs, and it does not care if that someone is the Host, a Manager, or a Guest. Furthermore, it does not insert any additional data to the smart contract, it only changes the value for a couple of variables already created. That is the reason why the use case UC9.MSS is the cheapest one, as shown in Figure 6.30.

Finally, UC9.AP1 and UC9.AP2 are alternative paths that result in transaction failure. The former is the Host trying to turn the feature off when it is already off, and the latter is the Host trying to turn it off when someone else holds the permission. Ethereum charges unsuccessful transactions a base Gas amount plus the computational steps executed to reach that conclusion.

6.2.5 Cost evaluation: The test network experiment

The experiment's procedure established a set of forty-three (43) sequential test cases, more specifically formed by all the transactions from the performance experiment base case, to be executed using the Görli test network. The smart contract address is 0x21Be6d84605607989934Cf5294e789dD681c8297 and can also be viewed using Etherscan when Görli is selected.

The test case behaviors were verified along with each execution as defined in the methodology, and all of them met the specification. Furthermore, all transactions resulted in the same Gas charge yielded by Ethereum *mainnet*.

In conclusion, the Görli *testnet* perfectly emulates the Gas calculations of the Ethereum main network.

6.2.6 Cost evaluation: The multiplicity experiment

The experiment's procedure defined a set of fifty-three (53) sequential transactions, which were executed using the Görli test network due to the results reported in the previous section. The smart contract address is 0xEf007eE1d68A16AF5D005caBF543CDA61F20Ea8c. The following paragraphs present and discuss the findings for each use case.

Figure 6.44 shows the Gas consumption of ten Guest registrations, ordered by execution – the y-axis is the units of Gas used, and the x-axis is both the Actor doing the transaction and the test case ID. As explained before based on the implementation, the first Guest is cheaper to register due to less work required to complete the action. After that, the only Gas charge change happens from CT16 to CT17, exactly when the Manager starts adding Guests, surcharge also discussed before. In conclusion, the number of Guests registered in the smart contract only impacted the cost when none was registered.

Figure 6.45 shows the Gas consumption of ten Manager registrations, ordered by execution – the y-axis is the units of Gas used, and the x-axis is the test case ID. Remember that only the Host can successfully register Managers. Similarly to registering Guests, the number of Managers only altered the Gas consumption when none was registered. After that, the cost was the same.

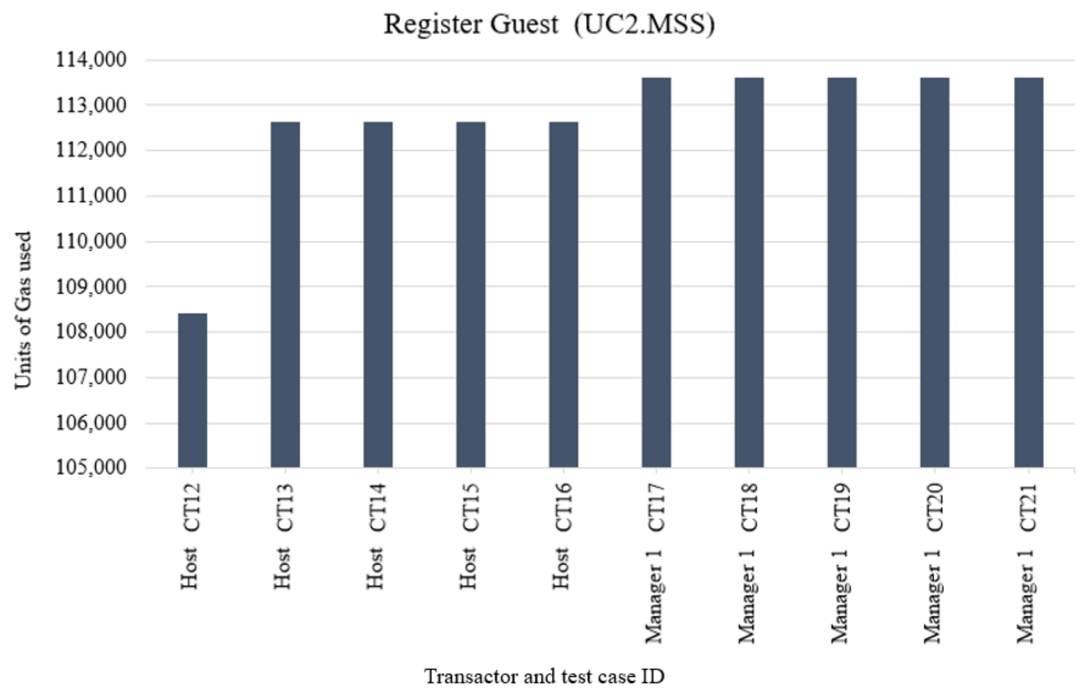


Figure 6.44 – Multiplicity cost evaluation: Register Guest (UC2.MSS)

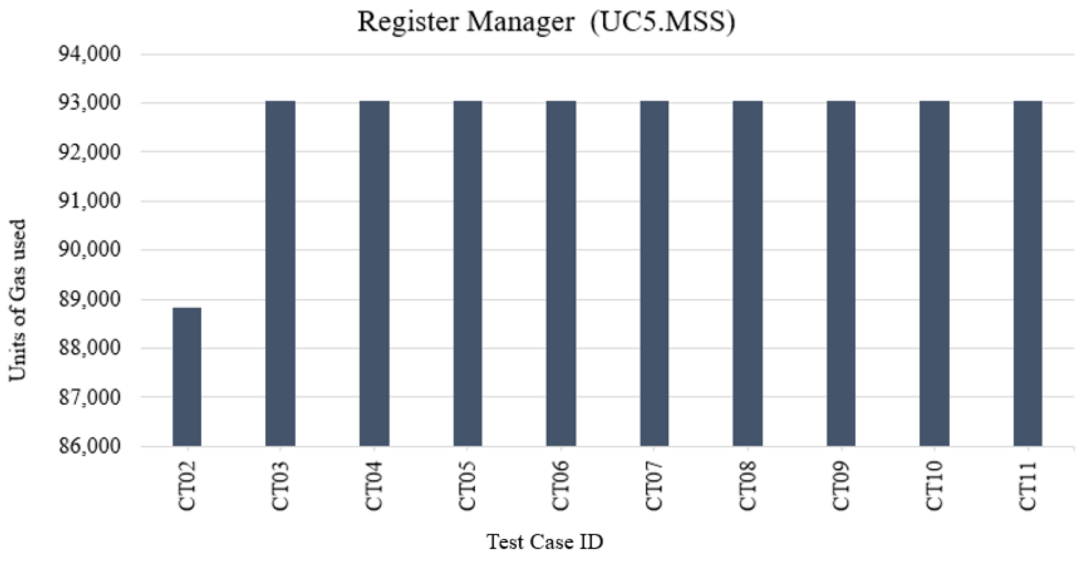


Figure 6.45 – Multiplicity cost evaluation: Register Manager (UC5.MSS)

Figure 6.46 shows the Gas consumption of ten Guest removals, ordered by execution – the y-axis is the units of Gas used, and the x-axis is both the Actor doing the transaction and the test case ID. As expected, the chart shows the price increase when changing the stakeholder from the Host to a Manager. Then, the last three transactions, therefore the last three Guests removed from the smart contract, required lower Gas than the others.

The fact that the last removal used the least amount of Gas was expected following the results from section 6.2.4.3. For CT41, the reason is the position that the Guest occupies in the array. To remove a Guest from the smart contract, it must be in the last position of the Guest array – see Figure 6.35. CT41 is the only test case in which the Guest being removed is already in the last position. CT42 requires a position swap, but between the first and last elements, which consumed slightly less Gas than CT39 and CT40. Therefore, to remove Guests, the Gas consumption might change according to the Guest quantity and the respective array position occupied by the desired address.

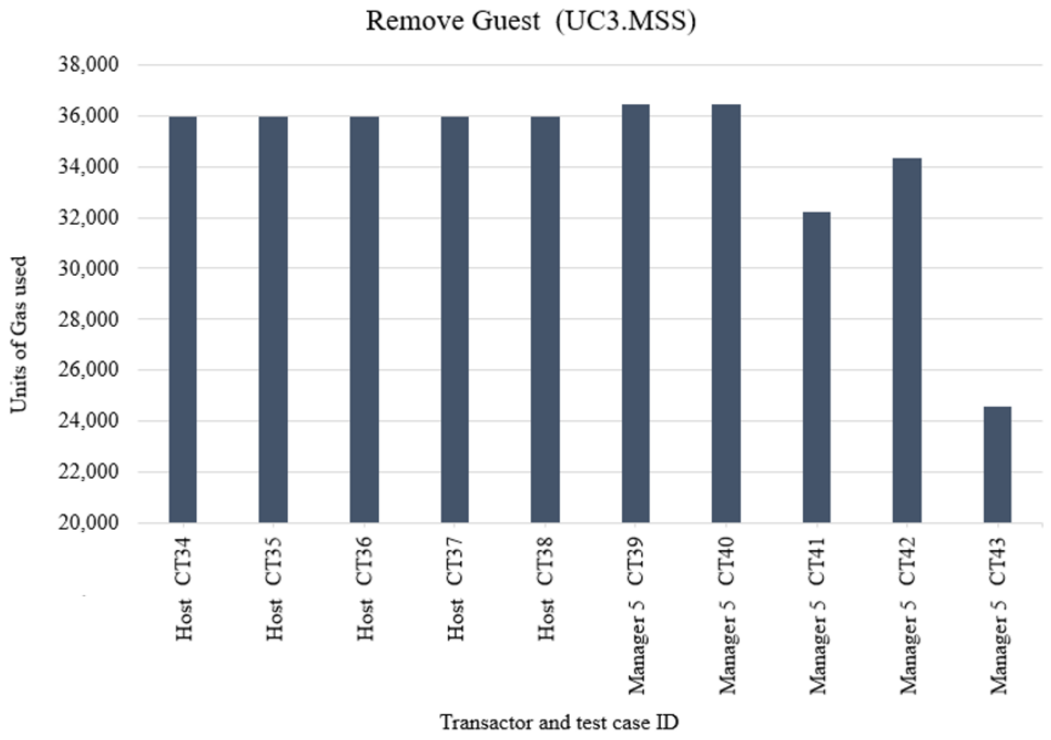


Figure 6.46 – Multiplicity cost evaluation: Remove Guest (UC3.MSS)

Figure 6.47 shows the Gas consumption of ten Manager removals, ordered by execution – the y-axis is the units of Gas used, and the x-axis is the test case ID. Remember that only the Host can successfully remove a Manager. Once again, this discussion is similar to *removeGuest* since they share most of the implementation strategy, including removal by position swapping.

The first four Managers are positioned in the middle of the array, hence the higher cost. CT48, on the other hand, was already positioned at the last position. The next four Managers – CT49 to CT52 - were always located at the first position of the array, requiring a swap between the first and last elements. Lastly, as expected, the lowest cost was to remove the last Manager.

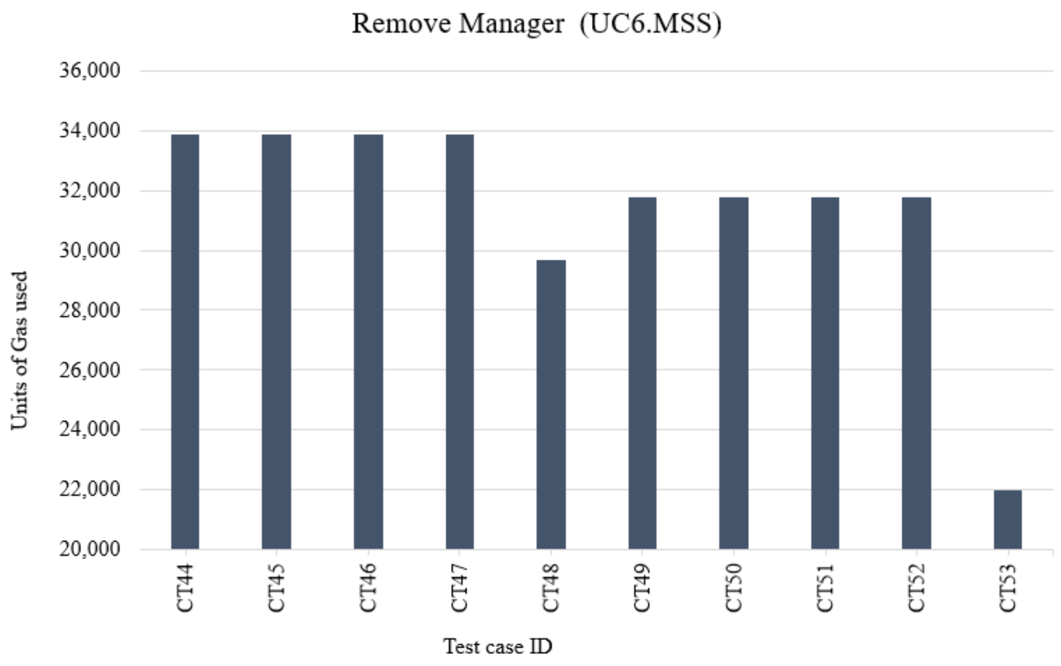


Figure 6.47 – Multiplicity cost evaluation: Remove Manager (UC6.MSS)

Figure 6.48 shows the Gas amount required to turn on the exclusive permission, ordered by execution – the y-axis is the units of Gas used, and the x-axis is the use case ID. Use case UC8.AP2 is the Host turning the feature on to three Managers, each located either at the beginning, the middle, or at the end of the Manager's array. Therefore, the Manager position does not impact the cost of that use case.

Use case UC8.AP1 is the Manager turning the feature on to three Guests, each located either at the beginning, the middle, or at the end of the Guest's array. Once again, the Gas consumption does not change according to the Guest position.

The surcharge of UC8.AP1 in comparison to UC8.AP2 was explained earlier in section 6.2.4.6.

Finally, Figure 6.49 shows the cost to turn off the exclusive permission feature (UC9.MSS), ordered by execution – the y-axis is the units of Gas used, and the x-axis is who is the Actor performing the transaction.

In accordance with the findings and analysis of section 6.2.4.7, the cost to turn off the feature does not change regardless of how many Managers or Guests are registered in the smart contract.

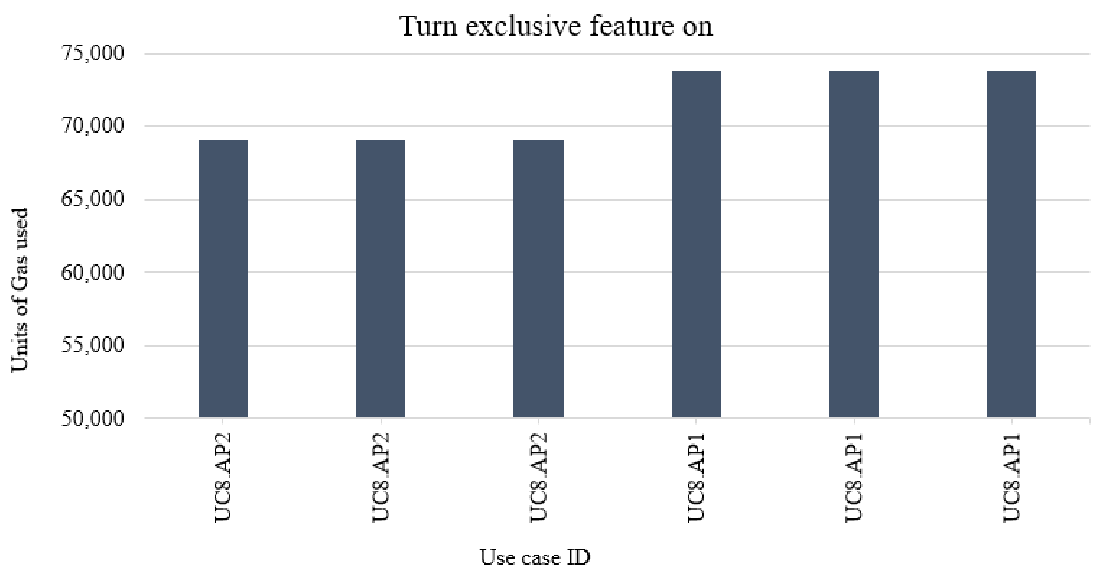


Figure 6.48 – Multiplicity cost evaluation: Turn the exclusive feature on (UC8.AP1 and UC8.AP2)

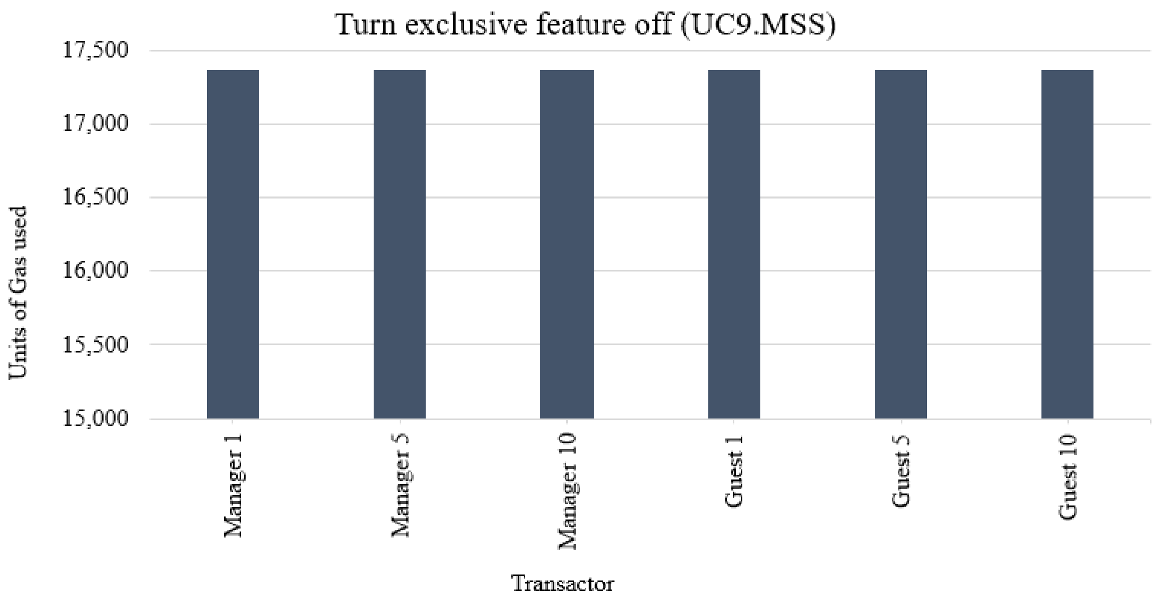


Figure 6.49 – Multiplicity cost evaluation: Turn exclusive feature off (UC9.MSS)

CHAPTER 7

CONCLUSIONS, CONTRIBUTION, AND FUTURE WORK

This work proposes, implements, and evaluates a smart lock system built using the Ethereum blockchain. The system allows a person to have full control of their device and manage it remotely without requiring a central authority – e.g., the lock's manufacturer - to host and administrate the solution.

The architecture discussion in chapter four addresses the first research question and objective – how a smart lock solution can be implemented using a permissionless blockchain. It creates a detailed application scenario in which various functionalities are required from the smart lock and proposes an architecture capable of meeting those needs. The discussion demonstrates how complex it is to build an application that properly embraces decentralization and the many aspects and components that should be considered when doing so. Moreover, it shows that poorly designed solutions can use the blockchain but still face some of the same issues encountered when using centralized applications.

From a different perspective and at a lower level of abstraction, chapter five also addresses the first research question and objective. The discussion surrounding the smart contract implementation shows that data privacy and security must be accounted for when writing every function. If they are not designed properly, the smart contract can expose sensitive data or allow unauthorized users to perform actions that they are not supposed to. Furthermore, it debates on the tradeoffs associated with some design decisions as, for example, when to use a transaction or a query-based function and the corresponding access control security and delay to process the call.

In summary, chapters four and five show that it is possible to design the desired smart lock solution, how it looks like, how it is implemented, and how it works. In addition to that, it highlights the impact that some architecture and implementation decisions can have on the system behavior.

The experiments and evaluations from chapter six address the remaining research questions and objectives, related to the system behavior, performance, and cost. First, it shows that the system delivers the expected behavior to all the test cases. Therefore, it demonstrates that the architecture and implementation are indeed capable of meeting the needs and use cases described throughout this work.

Regarding the performance, the evaluation shows that the delay to retrieve data from the blockchain – i.e., from the smart contract – is in the milliseconds' range. On the other hand, it also reveals that transaction delays – i.e., sending data to the smart contract – are affected by many factors but that there is a mechanism that users can leverage to keep them low, offering higher gas prices. Regardless of the network dynamics, a Gas price of twelve Gwei was able to keep the transaction delays under one minute. Besides unlocking the lock, which is a data retrieval operation that happens in milliseconds, all the other actions are not time-sensitive, meaning that delays of a few minutes are acceptable and do not compromise the applicability of the proposed solution.

The cost evaluation adopted the high Gas price mentioned above to calculate the cost of each use case scenario representing a worst-case scenario – users less sensitive to longer waiting times could offer lower Gas prices, consequently paying less money. The evaluation shows that, even with that consideration, the most expensive action is to deploy the smart contract, which costs approximately seven CAD. This deployment, however, is a one-time-only operation performed when setting up the device. Besides that, all other transactions yielded costs between five and forty cents. Therefore, the results suggest that the costs are reasonable considering the application and the benefits of the proposed system.

In conclusion, this work shows that it is possible and feasible to leverage the public Ethereum blockchain to build a smart lock solution.

7.1 Limitations

The limitations of this work are:

- Lock's hardware development: This device plays a crucial role in the success of the proposed solution and must be carefully designed to support the use cases securely for all stakeholders. It was out of the scope of this work due to its high complexity, which is believed to require a thesis of its own. For instance, additional use cases and considerations for the application might be necessary – e.g., how to set up and keep the smart contract address of the device, how the stakeholders' identities are verified, how to allow the smart contract address to be changed without compromising the stakeholders' rights over the device, and how to deal with an unavailable internet connection.

- Use cases: (i) Only a single Guest can hold the exclusive permission feature, but it could be necessary to enable others to have it simultaneously – e.g., a couple or a group of friends staying at a place might not want to rely on single permission; (ii) The smart contract only allows a single Host to exist, but it could be necessary to have multiple people with the same rights over the device.
- User experience: (i) Once the lock's management interface does not keep any state saved, users must store they smart contract address somewhere and inform it every time they open the tool; (ii) Users are required to exchange their Ethereum accounts and smart contract addresses to manage the device; (iii) The architecture would allow a malicious Host to trick a Guest by creating a fake smart contract that is not the one being used by the lock device.
- Privacy: As discussed before, information as the start and end dates of stays, and the address of the Guests and Managers, for instance, can be obtained through the records of the blockchain transactions.
- Cost: Although the Gas amount charged by the transactions are fixed, the corresponding value of ETH in CAD changes frequently, which might impact users.

7.2 Contributions

The contributions of this work are:

- The early version of this work was published (De Camargo Silva et al., 2019).
- Architecture and implementation: The in-depth discussion surrounding the required components, issues, and factors that must be taken into consideration when designing and developing an application that leverages the blockchain;
- Evaluations: It shows that it is possible and feasible to build and use the proposed smart lock solution using the Ethereum blockchain while demonstrating its tradeoffs.
- Architecture, implementation, and evaluations: Sets a foundation that can be extended to build many other applications as gym and school lockers, self-storage facilities,

access control to workplaces, hospitality services, other physical properties, among others. It provides an initial idea about expected costs and performance for that kind of application.

7.3 Future Work

Some possibilities for future works associated with this thesis are:

- Develop the lock device hardware discussed in this architecture;
- Address the limitations presented in section 7.1;
- Build other access control applications based on this architecture as suggested in section 7.2;
- Substitute Ethereum with other public blockchains, or even with private or consortium blockchains, and contrast the architecture, implementation, tradeoffs, behavior, performance, and cost with the system proposed in this thesis.

REFERENCES

[1] Alchemy. (2020). Retrieved January 13, 2020, from https://alchemyapi.io/

[2] August (2020). August Lock – How it Works. Retrieved June 12, 2020, from https://august.com/pages/how-it-works

[3] Aung, Y. N., & Tantidham, T. (2018). Review of Ethereum: Smart home case study. Proceeding of 2017 2nd International Conference on Information Technology, INCIT 2017, 2018-Janua, 1–4. https://doi.org/10.1109/INCIT.2017.8257877

[4] Aung, Y. N., & Tantidham, T. (2019). Ethereum-based Emergency Service for Smart Home System: Smart Contract Implementation. International Conference on Advanced Communication Technology, ICACT, 2019-Febru, 147–152. https://doi.org/10.23919/ICACT.2019.8701987

[5] Brandão, A., Mamede, H. S., & Gonçalves, R. (2018). Systematic Review of the Literature, Research on Blockchain Technology as Support to the Trust Model Proposed Applied to Smart Places. In 10th European Conference on Information Systems Management. Academic Conferences and publishing limited (Vol. 1, pp. 1163–1174). https://doi.org/10.1007/978-3-319-77703-0_113

[6] Buterin, V. (2016). Ethereum: Platform Review - Opportunities and Challenges for Private and Consortium Blockchains. Retrieved October 25, 2019, from https://pt.scribd.com/doc/314477721/Ethereum-Platform-Review-Opportunities-and-Challenges-for-Private-and-Consortium-Blockchains

[7] Buterin, V. (2015). On Public and Private Blockchains. Retrieved October 2, 2019, from https://blog.ethereum.org/2015/08/07/on-public-and-private-blockchains/

[8] Buterin, V. (2014a). Ethereum: A Next-Generation Generalized Smart Contract and Decentralized Application Platform. Retrieved October 25, 2019, from https://web.archive.org/web/20140111180823/http:/ethereum.org/ethereum.html

[9] Buterin, V. (2014b). Ethereum: Now Going Public. Retrieved October 25, 2019, from Ethereum blog website: https://blog.ethereum.org/2014/01/23/ethereum-now-going-public/

[10] Casino, F., Dasaklis, T. K., & Patsakis, C. (2019). A systematic literature review of blockchain-based applications: Current status, classification and open issues. Telematics and Informatics, 36(November 2018), 55–81. https://doi.org/10.1016/j.tele.2018.11.006

[11] Christidis, K., & Devetsikiotis, M. (2016). Blockchains and Smart Contracts for the Internet of Things. IEEE Access, 4, 2292–2303. https://doi.org/10.1109/ACCESS.2016.2566339

[12] Coinbase. (2020). Retrieved March 13, 2020, from https://www.coinbase.com/price/ethereum/cad

[13] De Camargo Silva, L., Samaniego, M., & Deters, R. (2019). IoT and Blockchain for Smart Locks. IEEE 10th Annual Information Technology, Electronics and Mobile Communication Conference (IEMCON), Vancouver, BC, Canada, 2019, pp. 0262-0269, doi: 10.1109/IEMCON.2019.8936140.

[14] Ethereum Developer Resources. (2019). Retrieved October 25, 2019, from Ethereum website: https://www.ethereum.org/developers/#getting-started

[15] Ethereum Gas Station. (2020). Retrieved March 13, 2020, from https://ethgasstation.info/

[16] Ethereum White Paper. (2019). Retrieved October 25, 2019, from Github website: https://github.com/ethereum/wiki/wiki/White-Paper

[17] Etherscan. (2020a). Ethereum Developer APIs. Retrieved January 13, 2020, from https://etherscan.io/apis

[18] Etherscan. (2020b). Retrieved January 21, 2020, from https://goerli.etherscan.io/

[19] Feng, Q., He, D., Zeadally, S., Khan, M. K., & Kumar, N. (2019). A survey on privacy protection in blockchain system. Journal of Network and Computer Applications, 126, 45–58. https://doi.org/10.1016/j.jnca.2018.10.020

[20] Friday (2020). Friday Smart Lock. Retrieved June 12, 2020, from https://www.fridayhome.net/views/home-page.html

[21] Galal, H. S., & Youssef, A. M. (2019a). Verifiable Sealed-Bid Auction on the Ethereum Blockchain. https://doi.org/10.1007/978-3-662-58820-8_18

[22] Galal, H. S., & Youssef, A. M. (2019b). Trustee: Full Privacy Preserving Vickrey Auction on top of Ethereum. Retrieved from http://arxiv.org/abs/1905.06280

[23] Google (2020). Nest X Yale Lock. Retrieved June 12, 2020, from https://store.google.com/ca/product/nest_x_yale_lock

[24] Görli. (2020). Görli Testnet. Retrieved January 21, 2020, from https://goerli.net/

[25] Halpin, H., & Piekarska, M. (2017). Introduction to Security and Privacy on the Blockchain. 2017 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW), 1–3. https://doi.org/10.1109/EuroSPW.2017.43

[26] Han, D., Kim, H., & Jang, J. (2017). Blockchain based smart door lock system. International Conference on Information and Communication Technology Convergence: ICT Convergence Technologies Leading the Fourth Industrial Revolution, ICTC 2017, 2017-Decem, 1165–1167. https://doi.org/10.1109/ICTC.2017.8190886

[27] Infura. (2020). Retrieved January 13, 2020, from https://infura.io/

[28] Joseph, J., & Navaie, K. (2019). Blockchain Enabled Rooms Implementation For Internet of Things. (IoTSMS).

[29] Kwikset (2020). Kwikset Premis Lock. Retrieved June 12, 2020, from https://www.kwikset.com/premis

[30] Li, Y. (2019). A privacy preserving ethereum-based E-voting system (University of Stuttgart). Retrieved from http://dx.doi.org/10.18419/opus-10409

[31] Lu, Y. (2018). Blockchain and the related issues: a review of current research topics. Journal of Management Analytics, 5(4), 231–255. https://doi.org/10.1080/23270012.2018.1516523

[32] McCorry, P., Shahandashti, S. F., & Hao, F. (2017). A Smart Contract for Boardroom Voting with Maximum Voter Privacy. https://doi.org/10.1007/978-3-319-70972-7_20

[33] Memon, M., Hussain, S. S., Bajwa, U. A., & Ikhlas, A. (2018). Blockchain Beyond Bitcoin: Blockchain Technology Challenges and Real-World Applications. 2018 International Conference on Computing, Electronics & Communications Engineering (ICCECE), 29–34. https://doi.org/10.1109/iCCECOME.2018.8658518

[34] Mercer, R. (2016). Privacy on the Blockchain: Unique Ring Signatures. Retrieved from http://arxiv.org/abs/1612.01188

[35] Metamask. (2020). Metamask. Retrieved January 21, 2020, from https://metamask.io/

[36] Nakamoto, S. (2008). Bitcoin: A peer-to-peer electronic cash system. Retrieved from https://bitcoin.org/bitcoin.pdf

[37] Nodesmith. (2020). Retrieved January 13, 2020, from https://nodesmith.io/

[38] Pocket Network. (2020). Retrieved January 13, 2020, from https://www.pokt.network/

[39] Postman. (2020). Retrieved March 13, 2020, from https://www.postman.com/

[40] Rahman, M. A., Loukas, G., Abdullah, S. M., Abdu, A., Rahman, S. S., Hassanain, E., & Arafa, Y. (2019). Blockchain and IoT-based secure multimedia retrieval system for a massive crowd: Sharing economy perspective. ICMR 2019 - Proceedings of the 2019 ACM

International Conference on Multimedia Retrieval, 404–407. https://doi.org/10.1145/3323873.3326924

[41] Remix. (2020). Retrieved January 21, 2020, from https://remix.ethereum.org/

[42] Schiefer, M. (2015). Smart Home Definition and Security Threats. 2015 Ninth International Conference on IT Security Incident Management & IT Forensics, 114–118. https://doi.org/10.1109/IMF.2015.17

[43] Seebacher, S., & Schüritz, R. (2017). Blockchain Technology as an Enabler of Service Systems: A Structured Literature Review. https://doi.org/10.1007/978-3-319-56925-3_2

[44] Slock.it Incubed Client. (2020). Retrieved January 13, 2020, from https://slock.it/incubed/#products

[45] Szabo, N. (1997). Formalizing and Securing Relationships on Public Networks. First Monday, 2(9). https://doi.org/10.5210/fm.v2i9.548

[46] Szabo, N. (1994). Smart Contracts. Retrieved October 2, 2019, from http://www.fon.hum.uva.nl/rob/Courses/InformationInSpeech/CDROM/Literature/LOTwinterschool2006/szabo.best.vwh.net/smart.contracts.html

[47] Tama, B. A., Kweka, B. J., Park, Y., & Rhee, K.-H. (2017). A Critical Review of Blockchain and Its Current Applications. International Conference on Electrical Engineering and Computer Science (ICECOS) 2017, 109–113.

[48] Tikhomirov, S. (2017). Ethereum: state of knowledge and research perspectives. International Symposium on Foundations and Practice of Security, 206–221. Springer.

[49] Unterweger, A., Knirsch, F., Leixnering, C., & Engel, D. (2018). Lessons Learned from Implementing a Privacy-Preserving Smart Contract in Ethereum. 2018 9th IFIP International Conference on New Technologies, Mobility and Security (NTMS), 1–5. https://doi.org/10.1109/NTMS.2018.8328739

[50] Wahab, J. (2018). Privacy in Blockchain Systems. Retrieved from http://arxiv.org/abs/1809.10642

[51] Wang, H., Zheng, Z., Xie, S., Dai, H. N., & Chen, X. (2018). Blockchain challenges and opportunities: a survey. International Journal of Web and Grid Services, 14(4), 352. https://doi.org/10.1504/ijwgs.2018.10016848

[52] Web3.js. (2020). Web3.js - Ethereum JavaScript API. Retrieved January 13, 2020, from https://web3js.readthedocs.io/

[53] Web3j. (2020). Web3j. Retrieved January 13, 2020, from http://web3j.io/

[54] Wikipedia. (2020). Unix time. Retrieved January 30, 2020, from https://en.wikipedia.org/wiki/Unix_time

[55] Wood, G. (2014). ETHEREUM: A SECURE DECENTRALISED GENERALISED TRANSACTION LEDGER. Retrieved from https://www.semanticscholar.org/paper/ETHEREUM%3A-A-SECURE-DECENTRALISED-GENERALISED-LEDGER-Wood/da082d8dcb56ade3c632428bfccb88ded0493214?citationIntent=methodology#citing-papers

[56] Xu, Q., He, Z., Li, Z., & Xiao, M. (2019). Building an Ethereum-Based Decentralized Smart Home System. Proceedings of the International Conference on Parallel and Distributed Systems - ICPADS, 2018-Decem, 1004–1009. https://doi.org/10.1109/PADSW.2018.8644880

[57] Yavuz, E., Koc, A. K., Cabuk, U. C., & Dalkilic, G. (2018). Towards secure e-voting using ethereum blockchain. 2018 6th International Symposium on Digital Forensic and Security (ISDFS), 1–7. https://doi.org/10.1109/ISDFS.2018.8355340

[58] Zaparoli, M. X., de Souza, A. D., & de Oliveira Monteiro, A. H. (2019). SmartLock: Access Control Through Smart Contracts and Smart Property. (Itng), 105–109. https://doi.org/10.1007/978-3-030-14070-0_16

[59] Zheng, Z., Xie, S., Dai, H. N., Chen, X., & Wang, H. (2018). Blockchain challenges and opportunities: a survey. International Journal of Web and Grid Services, 14(4), 352. https://doi.org/10.1504/IJWGS.2018.10016848