A Design of a System developed on the Blockchain-based EV Sharing Management Model, its Simulation and Use in Evaluation of Different Consensus Mechanisms

Ian Sinegubko

4th Year Project Report
Computer Science and Mathematics
School of Informatics
University of Edinburgh

2023

Abstract

In combination, car sharing systems and the use of Electric Vehicles (EVs) provide a sustainable solution to transportation. However, there is a lack of trust, security, privacy and transparency in suggested EV sharing platforms. In this project, a blockchain-based EV sharing management model was introduced in order to address that challenge by using the advantages of the blockchain technology. Based on the proposed model, a full system was then designed.

In order to simulate and demonstrate the functionality of the designed system, a simplified tool was implemented and tested under the three Byzantine Fault Tolerant consensus mechanisms, namely BFT-SMaRt, Linear BFT-SMaRt and Hotstuff-inspired BFT-SMaRt. Then, different experiments were designed in order to analyse and compare the performance of the algorithms. Various observations were made and interesting results were obtained.

Research Ethics Approval

This project was planned in accordance with the Informatics Research Ethics policy. It did not involve any aspects that required approval from the Informatics Research Ethics committee.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

(*Ian Sinegubko*)

Acknowledgements

I would like to thank my supervisor, professor Xiao Chen, for his guidance throughout the project, and Mr Meng Kun for his technical support with the setup of the consensus libraries. I would also like to thank my family and my friends for all the love and support during these difficult times.

Table of Contents

Chapter 1

Introduction

1.1 Background of EV Sharing

The need for an improvement in resource utilization, as well as the social demand for making various aspects of human life easier and more convenient, have lead to an emergence of sharing economy [14]. In recent decades, a lot of sub-economies have formed within the sharing economy, each of which correspond to a certain aspect of human life - from housing to dog kennels. Car sharing industry have emerged as one of such sub-economies.

Car sharing was proposed in the 1980s for the first time in Berlin, Germany [20]. It was presented as a solution to sustainable transportation development, a topic that has drawn a lot of attention due to a growth in public awareness of environmental issues. Another advantage of a car sharing solution that has drawn attention of the experts is its convenience. It could become a solution that would be a balance between private and public transports, since on the one hand it is more attractive in terms of user experience, yet on the other hand less emissions are produced. Therefore, car sharing could reduce both costs on new public transport infrastructure and a negative impact on the environment.

Electric vehicle sharing has emerged as an extension to the car sharing industry with an aim to improve fuel economy [14]. It was proposed as a more sustainable solution as opposed to vehicle sharing, since EVs produce no tailpipe emissions.

Electric vehicle sharing platforms are designed to allow users to book electric vehicles by using software such as a mobile app or website. The vehicles are placed at various locations throughout the area, such as parking lots or designated charging stations. The users pay for the time they use the vehicle, typically on an hourly basis, and then return it to the required location when finished.

1.2 Key Challenges in the Current Use Case

As with any emerging technology, even though EV sharing gains popularity and is a promising model, there are challenges to address.

One of the major challenges is the lack of infrastructure. This includes the lack of charging stations in many areas, as well as their accessibility. The standards for manufacturing charging equipment vary with different countries [10]. For instance, USA, Europe and Japan use separate charging connector standards. Also, charging stations that work on renewable energy are expensive to design and implement, and they require a lot of space. It is also worth noting that there is a lack of charging stations on the highways.

A lot of challenges are related to electric vehicle batteries. Even though the charge of some electric vehicles can last for up to 500 km, the driving range of most of the EVs is typically limited from 250 km to 350 km [24]. Also, it normally takes quite long to charge an EV. To fully charge an EV's battery, it typically takes from four to eight hours. It is also worth noting that the batteries for electric vehicles are expensive, heavy and take up considerable vehicle space.

There are problems that EV sharing industry have inherited from a car sharing industry. This includes challenges related to user behavior, such as the potential for users to damage the vehicles or not properly charge them. If a lot of customers act immature and do not take the proper care of the vehicles while using them, this could result in significant costs and make a bad impression on future customers. Even though big damages can be detected with more ease, small damages can stay unnoticed for a while, which, considering the cumulative effect, can still result in significant problems.

Another inherited challenge are the regulations. The regulatory environment around EV sharing is still evolving, and companies may face challenges related to obtaining permits, complying with regulations, and ensuring the safety of their vehicles and users.

Finally, a significant challenge is a lack of trust, privacy and security. Users do not trust vehicle sharing companies, as they are afraid that their personal details can be stolen and used for malicious purposes. Users also fear unfair scenarios in case of a dispute, a possibility of being disadvantaged by EV sharing companies. If the system is not transparent, it is easy to take advantage of the user.

1.3 Project objective

The aim of this project was to propose a solution to the problem of lack of trust, privacy, security and transparency in EV sharing systems by using the advantages of blockchain technology. Namely, to introduce a blockchain-based EV sharing management model and describe a design of a potential system that can be based on that model. Then, implement a tool that simulates the functionality of the system within the three different testbeds, which are the original BFT SMaRt library, the Linear BFT SMaRt with the use of threshold signatures, and the updated version of the BFT SMaRt that simulates the behaviour of the Hotstuff consensus algorithm. Lastly, run different tests, experiments

and measurements in order to observe how the tool behaves under different consensus mechanisms and present the results.

The rest of the report is organised as follows. Chapter 2 provides the background knowledge required to understand the further parts of the report. Chapter 3 describes the current challenges and blockchain-based solutions to EV sharing management. Chapter 4 is focused on describing the proposed EV sharing model. The focus of Chapter 5 is to specify the design of the whole system, as well as the blockchain and the consensus designs. Chapter 6 explains how the simulation tool was implemented and how the testbeds were tested. Chapter 7 is about the performance evaluation, namely how testbeds were deployed and configured, how experiments were designed and what measurements and analysis were carried out. Finally, Chapter 8 concludes the report with discussing the achievements and limitations of this project, as well as what could be done in order to extend this work.

Chapter 2

Background

2.1 Blockchain

2.1.1 Blockchain Essentials

Blockchain technology is a decentralized system that allows for secure, transparent, and tamper-proof transactions. It enables parties to transact with each other directly without the involvement from such intermediaries as banks, government institutions, or other third parties. At its core, a blockchain can be described a digital ledger that is represented as a sequence of blocks and records transactions in a chronological order. Each block contains a cryptographic hash of the previous block, creating a chain of blocks that cannot be changed without altering every subsequent block. This makes it virtually impossible to forge the ledger without being detected. Figure 2.1 shows how the blockchain system can be represented as a generic chain of blocks.

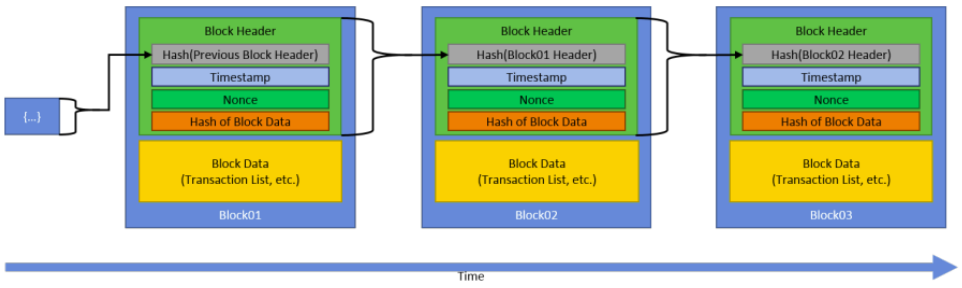

Figure 2.1: Generic chain of blocks [Figure 3, [30]].

Every blockchain runs under a consensus mechanism, which is often referred to as a *blockchain protocol*. It ensures that all parties in a blockchain network agree on the validity of the transactions being recorded and hence a consensus is reached. A blockchain protocol is a crucial notion for this project that is further explained in section 2.2.

The first blockchain was created in 2008 by an individual or group of individuals under the pseudonym Satoshi Nakamoto in the famous Bitcoin Whitepaper [21]. Nakamoto's

creation was the Bitcoin blockchain, which was designed to provide a decentralised solution to a long-lasting financial problem called a *double spending* problem. The blockchain system proposed by Nakamoto run under the consensus mechanism that was called *Proof of Work* (PoW) algorithm. In PoW, blocks are appended to the blockchain by the actors in a network that are called *miners*, who compete to create new blocks by solving a hard mathematical problem that requires big computational power. A miner that solves the problem first and whose block gets validated receives a reward paid in *Bitcoins*, which is a cryptocurrency of the Bitcoin blockchain.

The rise of Bitcoin blockchain has lead to an emergence of other blockchain platforms, which have extended the capabilities of blockchain beyond just financial transactions, as well as have proposed new rules for their blockchain systems and come up with new consensus mechanisms. Some of the most known examples of such platforms are Ethereum [5], Ripple [25], Hyperledger Fabric [2], R3 Corda [15] and Tendermint [18].

2.1.2 Classification

There exists a blockchain classification that consists of 4 categories: public, private, permissionless and permissioned.

- A *public* blockchain is a blockchain that anyone can get an access to. Public blockchains are best suited for use cases that require transparency and accountability, such as digital currencies or voting systems. Bitcoin and Ethereum are examples of public blockchains.
- A *private* blockchain is a blockchain that is operated by a single organisation or consortium of organisations. These blockchains have restricted access, which means that an organisation that operates the blockchain determines if someone is allowed to join. Private blockchains are best suited for use cases that require privacy and confidentiality, such as enterprise supply chain management or medical records.
- A *permissionless* blockchain is a blockchain in which everyone who has joined a network is alloed to participate, meaning that anyone can validate transactions and create new blocks. Permissionless blockchains are best suited for use cases that require censorship resistance and decentralization, such as digital currencies or decentralized applications (dapps). Besides being public, Bitcoin and Ethereum are also examples of permissionless blockchains.
- A *permissioned* blockchain is a blockchain in which only a limited group of participants have been granted a permission to act in the network, which means that only approved nodes are allowed to validate transactions and create new blocks. Permissioned blockchains are best suited for use cases that require a balance between transparency and privacy, such as government records or financial transactions. Examples of such are R3 Corda and Ripple.

A blockchain is always either public or private and either permissioned or permissionless. For example, Bitcoin blockchain is public and permissionless, while Hyperledger Fabric is private and permissioned. In some papers, terms *public* and *permissionless* are used

interchangeably, as well as terms *private* and *permissioned*, since public-permissioned or private-permissionless are a very untypical setups for a blockchain. However, it is important to note that in this report they are being distinguished, as in some cases there is a need to draw a line between the authentication and authorisation, for instance, in applications in the internet of things [12].

At first, all blockchains were permissionless, as Bitcoin blockchain. However, as the number of users has grown, so has the amount of data that needs to be processed. This has led to delays in transaction processing and higher transaction fees, as users compete to have their transactions validated first. In other words, the *scalability problem* has occured. Permissioned blockchains were created in part as a response to the scalability problem of permissionless blockchains. By limiting the number of nodes that are allowed to participate in the validation of transactions and the creation of new blocks, permissioned blockchains are able to process transactions more quickly and efficiently. In addition, permissioned blockchains are often designed with specific use cases in mind, which allows them to optimize their architecture for those use cases. This can lead to better performance and scalability than permissionless blockchains, which need to be more generalized in order to support a wide range of use cases. In this report, the focus is on permissioned blockchains.

2.1.3 Hyperledger Fabric

Fabric is a modular and extensible open-source system for deploying and operating permissioned blockchains and one of the Hyperledger projects hosted by the Linux Foundation [2]. The platform was designed for use in enterprise settings and is suitable for various applications such as supply chain management, asset tracking and identity management. Its modular architecture makes Fabric flexible and customisable. This means that a company or an organisation which uses the platform is able to adjust the components, such as consensus mechanism or chaincodes, based on the user's needs.

An interesting property of Fabric that comes from its modular architecture is that it supports pluggable consensus mechanisms, which means that it allows the users to choose the most suitable consensus algorithm for their needs and purposes. Because of that property, Hyperledger Fabric blockchain was used for the purposes of this project.

2.2 BFT Protocols

A *Byzantine Fault Tolerant* (BFT) protocol is a protocol for which is able to solve *Byzantine generals problem*. This section starts with an explanation of the Byzantine generals problem, and then focuses on the description of several BFT protocols that are relevant for this project.

2.2.1 Byzantine Generals Problem

Byzantine generals problem is a famous problem in distributive computing introduced by Robert Shostak, Marshall Pease and Leslie Lamport [19]. The overview of the problem is as follows. Let there be n armies, each commanded by a general, that have

surrounded the enemy castle. Each army has two options: attack or retreat. The generals have to coordinate their armies' actions in order to succeed, as the success is achieved if all the armies perform the same action. One of the generals (let's call them commander) is the head of all the other generals which sends order to all other generals, namely which action other generals have to perform. The question is how can the generals coordinate their actions considering that some of them (including the commander) may be traitors?

The first solution, proposed in the same paper, was called *Oral Message* algorithm, for which it was considered that the sent messages are oral messages, which means that they are fully controlled by the sender. For this type of messages, if there are m traitors, there must be at least $3m+1$ honest generals. As it can be seen in Figure 2.2, on the left the winning scenario is displayed, as there are no traitors. However, on the right, two of the six generals are traitors and, since $3*2+1=7\neq 6$, this scenario is a losing one.

Later in the paper, another algorithm was proposed as a more effective solution to the problem, which was called *Signed Message* algorithm, in which messages were considered to be unforgeable signed messages. Since the messages could not be forged, the validity of the exchanged messages was ensured.

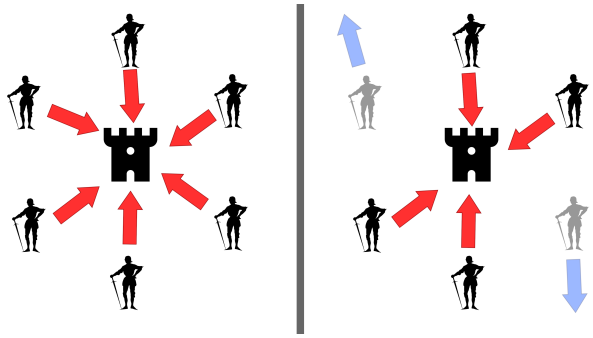

Figure 2.2: The winning scenario (on the left) and the losing scenario (on the right) [8].

Substituting generals with nodes in a network, a crucial problem in the field of distributive computing is obtained. The notion of Byzantine Fault Tolerance was later introduced to describe a system that tolerates Byzantine faults, which are the most difficult faults to deal with as there are no restrictions or assumptions to be made about the behaviour of such a node. Being Byzantine Fault Tolerant has become an essential condition for any distributive system.

2.2.2 PBFT

Practical Byzantine Fault Tolerance (PBFT) consensus algorithm was published in 1999 by Miguel Castro and Barbara Liskov [6]. It has provided high-performance Byzantine State Machine Replication (SMR), processing thousands of requests per second with sub-millisecond increases in latency. It is assumed that the algorithm acts in an *asynchronous* distributive system, which is a system in which there are no fixed bounds on the time it takes for messages to be transmitted between nodes or the time it takes for a node to process a message. Nodes in a system are assumed to be connected

by a network. In this subsection, some of the key characteristics of the algorithm are described.

2.2.2.1 BFT

The algorithm provides both *safety* and *liveness* if at most $\frac{n-1}{3}$ nodes are faulty, where n is the total number of nodes in the network. This means that in terms of BFT, PBFT performs equivalently to Oral Message algorithm explained in 2.2.1 earlier.

2.2.2.2 SMR

PBFT follows a *state machine replication* approach, where each node maintains a deterministic state machine and is called a *replica*. In other words, the algorithm implements a *deterministic replicated service* with a *state* and *operations*. The state of every replica contains the state of the service, the *message log* of messages that were accepted by the replica and an integer value which indicates the number of the current *view*, where view is a period during which the consensus is run under the same leader node. All replicas process requests from clients and produce a result, ensuring that the system remains consistent despite faulty nodes.

2.2.2.3 The Normal Case Operation and its Phases

Normal case operation is a state of the algorithm during which it is run under the same leader node that is called *primary* (other nodes are called *backup* nodes). In short, the algorithm can be described as follows.

1. A client sends a request to the primary node to start a service operation.
2. The primary node then sends the request to all the backup nodes.
3. All the replicas process the request and send their reply to the client.
4. Once the client receives replies from $f + 1$ different replicas, where f is the number of faulty replicas, the consensus is reached.

In PBFT, this algorithm is split on three phases, which are *pre-prepare*, *prepare* and *commit*.

Pre-prepare phase: The primary node sends the *pre-prepare* message to the backups. Then each backup replica checks the received data and if it is correct, it accepts the message and hence enters the prepare phase.

Prepare phase: Each replica that has received a pre-prepare message, sends a *prepare* message to all other replicas. If that prepare message is valid, replicas accept it and enter the commit phase.

Commit phase: Each replica that has received a prepare message, sends a *commit* message. If that commit message is valid, replicas execute the operation mentioned in message m and send the response to the client.

The Figure 2.3 displays the whole normal case operation of PBFT. Note that in this example, replica 0 acts as the primary node, while replica 3 is being a faulty node.

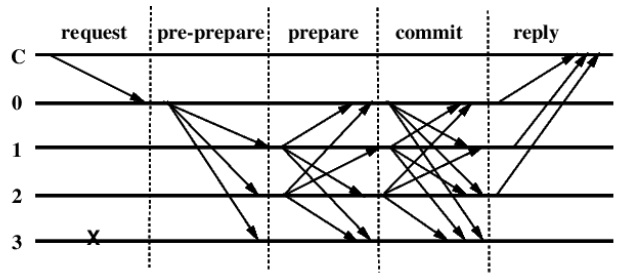

Figure 2.3: Normal Case Operation of PBFT [Figure 1, [6]]

2.2.2.4 View-Change Phase

View-change phase is the state of the algorithm during which the role of the primary node is assigned to a former backup node. This mechanism is executed in order to guarantee liveness in case the previous primary node fails.

The view-change process is triggered by the timeouts. If a timer run by a backup replica detects a timeout, this replica broadcasts the view-change message to all other replicas, which initiates the view-change process. Then the new primary node is determined.

2.2.2.5 Communication Complexity

PBFT has a communication complexity of $O(n^2)$, where n is the number of nodes in the network. This is because each node has to communicate with every other node during the prepare and commit phases. This makes the algorithm not highly scalable, since its performance can degrade with an increase in the number of nodes.

2.2.2.6 Heritage

The hype around the blockchain technology has increased the demand for consensus algorithms, as they could be used as blockchain protocols. Hence, a lot of consensus algorithms have emerged under PBFT's influence in the recent years, such as BFT-SMaRt[3], HotStuff[31], SBFT[13], Tendermint[4] and others. In this project, the focus is on BFT-SMaRt and HotStuff.

2.2.3 BFT-SMaRt

BFT-SMaRt is an open-source Java-based library implementing robust BFT SMR that was introduced by Alysson Bessani, João Sousa and Eduardo E. P. Alchieri in 2014 [3]. The library was introduced to provide an improved SMR-based solution with the aim to outperform the existing solutions such as PBFT algorithm described in 2.2.2. Similarly to PBFT, the algorithm implemented by the library follows a SMR approach and provides both safety and liveness if at most $\frac{n-1}{3}$ nodes are faulty, where n is the total number of nodes in the network.

The algorithm assumes a *partially synchronous* network model, which means that the system transitions from an asynchronous phase, where there are no bounds on message delays or processing times, to a synchronous phase, where messages are delivered

within known time bounds. The transition happens at an unknown point in time called *Global Stabilization Time* (GST). In this subsection, some of the key characteristics of the algorithm are described.

2.2.3.1 Modularity

While PBFT implements a *monolithic* protocol by embedding the consensus algorithm inside the SMR, BFT-SMaRt implements a *modular* SMR protocol. Modular protocols are easier to implement and reason about and this approach does not affect the performance in any way. Figure 2.4 shows how the functionality of the library can be divided on modules, where modules that are relevant to each other communicate.

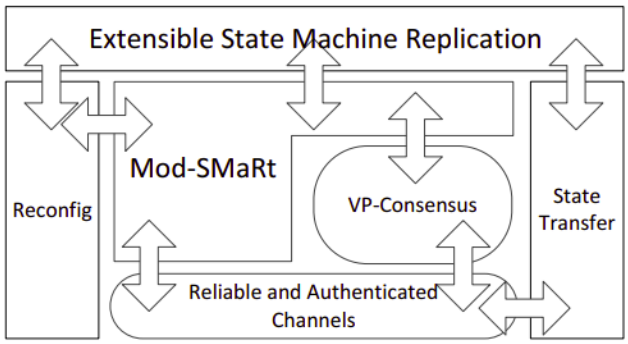

Figure 2.4: The modularity of BFT-SMaRt [Figure 1, [3]]

2.2.3.2 Mod-SMaRt

Mod-SMaRt is a modular protocol that implements BFT SMR using underlying consensus primitive in order to achieve *total order multicast*, which means it is ensured that messages are delivered to all nodes in the same order. Total order is achieved through a sequence of consensus instances, which are implemented in a way similar to PBFT. The normal case operation consists of three communication steps that are called *propose*, *write* and *accept*, which correspond to the three phases in PBFT: *pre-prepare*, *prepare* and *commit* respectively. Figure 2.5 displays the normal case operation of Mod-SMaRt.

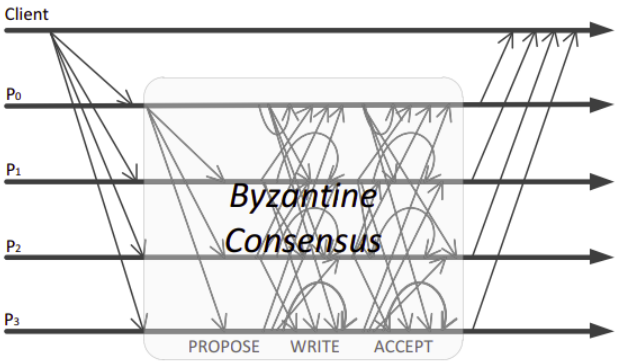

Figure 2.5: Normal Case Operation of BFT-SMaRt [Figure 2, [3]]

The view-change phase is implemented within Mod-SMaRt and is called a *synchronisation phase*. During this phase, besides the change of the leader node, the replicas are forced to switch to the same consensus instance, which can make some replicas trigger the state transfer protocol.

2.2.3.3 The State Transfer Protocol

The *state transfer* protocol makes sure that the replicas get repaired and reintegrated into the system without restarting the whole replicated service, in order to implement a practical state machine replication. It also recovers the whole system in case more than f replicas became faulty due to a sequence of correlated failures. The protocol is implemented in a separate module between the replication protocol and the application, without making any impact on the consensus protocol.

2.2.3.4 Reconfiguration

Unlike PBFT, BFT SMaRt provides an additional protocol that that enables replicas to be added or removed from the system during the execution. The protocol is implemented in a separate module that is denoted as Reconfig in Figure 2.4. The process can only be started by system administrators running a *View Manager* client. The protocol improves the flexibility and adaptability of the system to Byzantine faults.

2.2.3.5 Evaluation

The communication complexity of BFT-SMaRt is the same as of PBFT. Namely, it is $O(n^2)$, where n is the number of nodes in the network. This is because, in terms of communication, BFT-SMaRt follows the same steps as PBFT. However, it is worth noting that BFT-SMaRt uses such concept as *Multi-core awareness*, which means that a multi-core architecture of servers is used to scale the throughput of the system. This optimisation, combined with the use of the reconfiguration protocol, improves the overall performance of BFT-SMaRt in comparison to PBFT.

2.2.4 Basic HotStuff

Basic HotStuff is a consensus algorithm proposed by Maofan Yin, Dahlia Malkhi, Michael K. Reiter, Guy Golan-Gueta, and Ittai Abraham in 2018 [31]. The introduced algorithm was the first consensus algorithm to provide linear view change and optimistic responsiveness. Similarly to PBFT and BFT-SMaRt, the algorithm follows a SMR approach and provides both safety and liveness if at most $\frac{n-1}{3}$ nodes are faulty, where n is the total number of nodes in the network. Similarly to BFT-SMaRt, Hotstuff assumes a partially synchronous network model. In this subsection, some of the key characteristics of the algorithm are provided and described.

2.2.4.1 Cryptographic Primitives

Basic Hotstuff makes use of such concept as *threshold signatures* [26]. A threshold signature is a type of digital signature scheme that allows a group of participants, or

nodes, to generate a single signature for a given message. A (k,n)-threshold signature scheme is used by the algorithm, where $k = 2f + 1$ for f faulty replicas and n is the total number of replicas in the network. Each replica in the network holds the same public key and a private key, that is unique for each replica. If there are at least k replicas, each of which contribute a partial signature, then these k partial signatures can generate a complete signature that can be verified by using a public key.

2.2.4.2 Communication Network

Unlike in PBFT or BFT-SMaRt, where each replica has to communicate to every other replica in the network, in Basic Hotstuff every node communicates with other replicas only through the leader node. In other words, the algorithm uses a *star* network setup instead of the *mesh* network setup, used by previously mentioned algorithms.

2.2.4.3 Linear view change

In Basic Hotstuff, the view change is merged into the normal case operation in order to reduce the communication complexity. Instead of changing the leader node in the case of its failure, as it happens in PBFT and BFT-SMaRt, the algorithm uses a rotating leader paradigm, which means that the leader replica is changed every three rounds. This allows to achieve a linear view change.

2.2.4.4 Optimistic Responsiveness

Another innovation introduced by Basic Hotstuff is the *optimistic responsiveness*. Once the new leader replica is determined, it has to wait for only $n - f$ responses to be able to guarantee that its proposal will progress after GST, where n is the total number of nodes and f is the number of faulty nodes.

2.2.4.5 Communication Operation

The communication of Basic Hotstuff consists of four phases, which are *prepare*, *pre-commit*, *commit* and *decide*.

Prepare phase. First, each replica sends a new-view message with the highest *prepare* Quarum Certificate (QC) that it has received. A *Quarum Certificate* is a collection of $(n - f)$ votes over the leader proposal. The leader receives the messages and chooses *prepare* QC with the biggest view number. Then, the leader node attaches the data received from the chosen QC to the *prepare* message and send it to other replicas. Replicas then check received messages and if the safety requirements are met, they send their *prepare* votes with partial signatures to the leader.

Pre-commit phase. Once the leader replica receives the *prepare votes*, it combines them into a *prepare* QC and then broadcasts it to all other replicas in *pre-commit* messages. Replicas then check received messages and if the safety requirements are met, they send their *pre-commit* votes with partial signatures to the leader.

Commit phase. Once the leader replica receives the *pre-commit votes*, it combines them into a *pre-commit* QC and then broadcasts it to all other replicas in *commit* messages.

Replicas then check received messages and if the safety requirements are met, they send their *commit* votes with partial signatures to the leader.

Decide phase. Once the leader replica receives the *commit votes*, it combines them into a *commit* QC and then broadcasts it to all other replicas in *decide* messages. This starts the next view.

It is worth noting that there is also a *Next view interrupt*, which forces the system to start the new view in case of a timeout.

2.2.4.6 Evaluation

The use of threshold signature scheme and rotating leader paradigm allows Basic Hotstuff to reduce its complexity to $O(n)$, which is a better performance in comarison to PBFT and BFT-SMaRt. However, it is important to note that in order to simplify the leader replacement process, another phase had to be added to each view. Therefore, the best-case latency, which is the number of round trips needed to commit a transaction after GST assuming an honest leader node, has been increased to three round trips. In comparison, the best-case latency for both PBFT and BFT-SMaRt is two round trips. However, as claimed in [31], Basic Hotstuff is able to achieve comparable latency and much higher throughput than BFT-SMaRt.

2.3 Internet of Things

The term “Internet of Things” (IoT) was first used in 1999 by British technology pioneer Kevin Ashton to describe a system in which objects in the physical world could be connected to the Internet by sensors[22]. Since then, the term generally refers to scenarios where network connectivity and computing capability extends to objects, sensors and everyday items not normally considered computers, allowing these devices to generate, exchange and consume data with minimal human intervention [IoT Definitions, [22]].

Nowadays, most of the IoT solutions rely on a server-client paradigm[12]. However, new challenges that are arising with the IoT growth increase the demand for the new, alternative paradigms. Some experts claim that the most promising paradigm would be the one where decentralized architectures are involved. For instance, there was a proposal to use decentralized architectures in order to create large Peer-to-Peer Wireless Sensor Networks [27, 1, 17]. Figure 2.6 shows the evolution of IoT from centralized mainframe architectures to open-access cloud-centered architectures, also proposing the model for the future - decentralized architectures, where the cloud functionality is distributed among multiple peers.

An IoT solution must be reliable in terms of privacy and security, which was not the case with the Peer-to-Peer Wireless Sensor Networks. It was then realized that the blockchain technology could be used to improve those factors and, hence, perfect the potential future model for IoT. Then, the Blockchain-based IoT (BIoT) architectures were introduced.

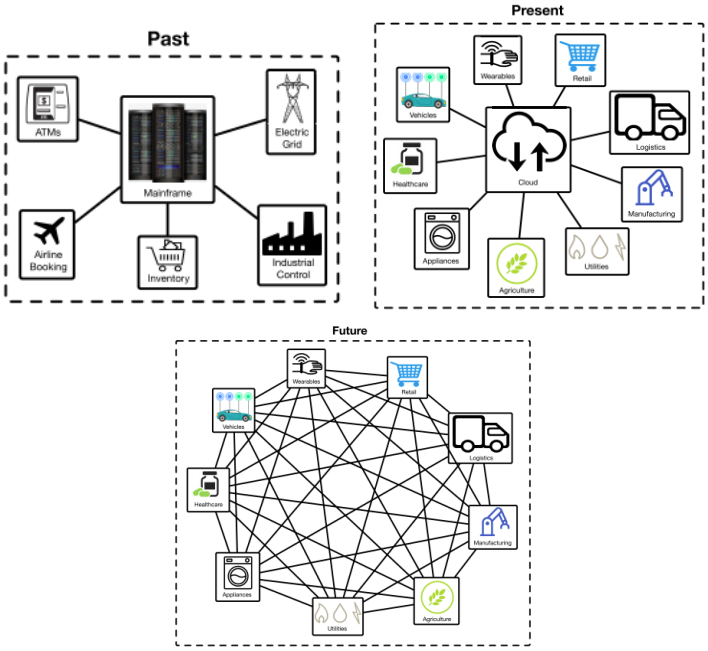

Figure 2.6: Past, present and future IoT architectures. [Figure 1, [12]]

Blockchain-based IoT can be applied in many areas [12]. Figure 2.7 shows some of the most relevant fields. In this project, the focus is on BIoT application in management of EV sharing systems.

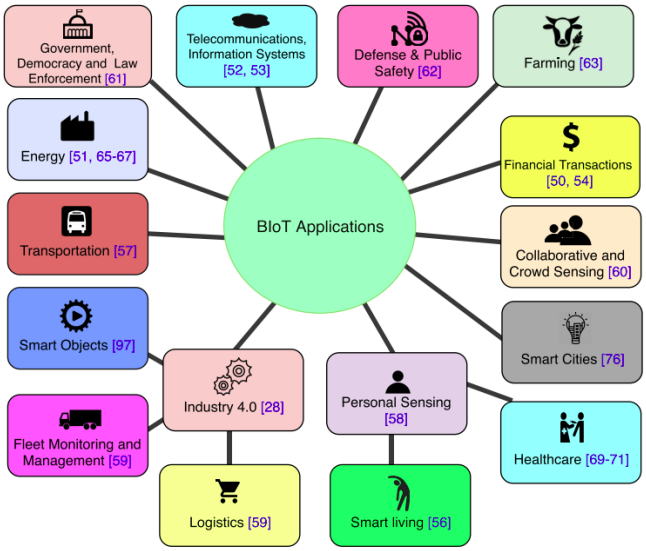

Figure 2.7: BIoT applications. [Figure 4, [12]]

Chapter 3

Related Work

Currently, there exists a number of blockchain-based solutions for sharing management of vehicles of different kinds. Some of the solutions specialise on EV sharing in particular. In this chapter, a collection of a few projects is described.

Helbiz is an Italian-American company that provides an EV-sharing platform. The platform mostly provides electric bicyles and scooters and works very similar to such EV-sharing platforms as Lime, Voi or Bird, except that it uses blockchain technology. The idea behind the platform was to create a sustainable transportation ecosystem, in which users pay using a platform's own ERC-20 [29] called HBZ. It is also worth mentioning that users get rewarded with HBZ tokens if they provide the data about their driving, which is then passed to insurance companies.

Another interesting project introduced in 2017 is *DAV* (Decentralized Autonomous Vehicles) [9]. DAV is a blockchain-based platform that connects autonomous vehicles, potential users and service providers, such as owners of charging stations. The platform operates on Ethereum blockchain, which uses Proof of Stake (PoS) consensus mechanism [23]. Transactions in the system are made with DAV token, however some services in the network can be purchased with virtual Ethereum-based tokens. When users that own a vehicle or a charging station get rewarded with DAV tokens if they provide it to other users in the network.

Also it is worth mentioning a project that is called *HireGo* [16]. HireGo provides a decentralised, peer-to-peer marketplace that allows its users to lease their vehicles. The application is built on Ethereum and uses its own HGO tokens, which are ERC-20 standard tokens. The application consists of three smart contracts, the first of which is used to get HGO tokens, the second one is used to get a ERC-721 token [11] which is a token needed to unlock the vehicle, and the last one is a rental contract. The rental contract acts as a middle man between users and vehicle owners. The HGO tokens sent by a user and the unlocking token sent by the vehicle owner get locked until for a certain period of time. This is done to ensure that the exchange is made at the same time and hence makes the process more secure.

A good example of a blockchain-based car-sharing platform is proposed in a paper published by Viktor Valaštín, Kristián Košt'ál, Rastislav Bencel and Ivan Kotuliak

in 2019 [28]. The application is built on top of Ethereum blockchain and uses two different tokens, a fungible ERC-20 token and a non-fungible ERC-721 token, where the first token represents car asset and second is the Unlock token that will be needed to unlock the car. The platform provides client-to-client, business-to-client and business-to-business solutions. Figure 3.1 shows the user interaction with the system. First, the user chooses the car, rental period and the preferred currency. The application exchanges the money of the user to ETH and then the user gets an Unlock token through an interaction with the smart contract. When user requests an access to the car by providing the Unlock token, the IoT device in the car validates the token by communicating with the blockchain network.

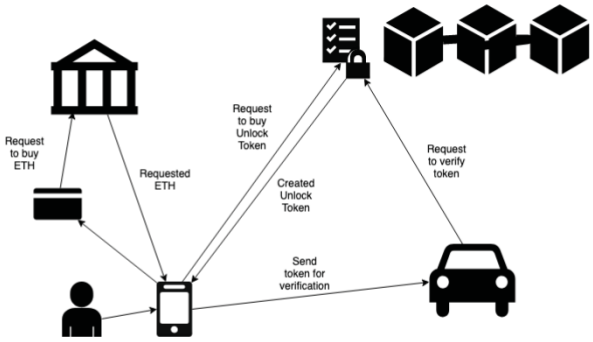

Figure 3.1: Graphic representation of our solution [Figure 4, [28]].

In 2020, Qihao Zhou, Zhe Yang, Kuan Zhang, Kan Zheng and Jie Liu proposed a decentralized car-sharing control scheme [32] that is based on blockchain and smart contracts. The proposed decentralised architecture consists of three components, which are terminals, base stations and a cloud server. Terminals consist of three types of entities, such as owner, tenant and vehicle. Base stations represent an Ethereum private chain that replaces a centralized third-party server. The cloud server is responsible for application operations, mainly user registration and management. Figure 3.2 displays the architecture of the system.

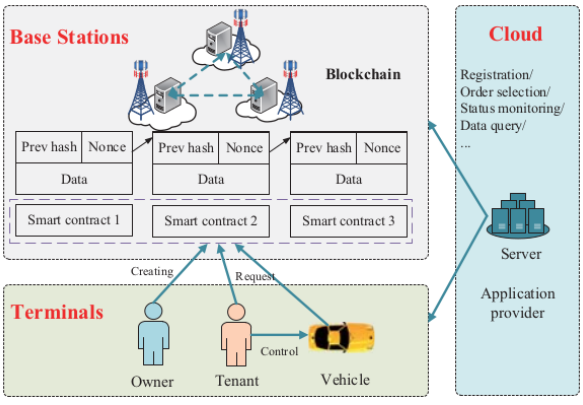

Figure 3.2: System architecture with the proposed control scheme for car sharing [Figure 2, [32]].

Chapter 4

System Model

4.1 Scenario and Requirements

In this chapter, a blockchain-based electric vehicle (EV) sharing model designed to address the challenges of sustainable urban transportation, is described. The model was designed to provide a Business-to-Consumer (B2C) solution. Therefore, the entities involved are EV sharing companies and its customers. The software platform connects companies that own electric vehicles with individuals who need access to a vehicle. A user can book a vehicle through the platform, then get an access to the vehicle and then leave it at one of the offices of the company.

It is important that the designed model meets certain requirements. Firstly, since the platform is blockchain-based, it is expected to be secure, transparent and decentralised. Using the advantages of a blockchain system, the platform must ensure secure and transparent transactions, protecting user data and preventing fraud or malicious activities. Secondly, the model must show good scalability. The platform must be able to handle an increasing number of users and transactions, as the system grows.

Thirdly, the platform must be accessible and easy to use. Anyone should be able to get an access to it through the internet or by downloading a certain software. The GUI of the system must be intuitive and clear to any potential user.

Also it is important that the interoperability requirement is met. Since it is assumed that the platform operates in an areas with all the necessary infrastructure, it must be ensured that the platform is aware of and is compatible with various types of electric vehicles, as well as it keeps track of the charging stations in the areas and can communicate with them.

4.2 Design Principles

In order to make the system secure, transparent and decentralised, it was connected to a Fabric blockchain described in Section 2.1.3. The fact that Fabric is a permissioned blockchain makes the system more scalable. Since the number of nodes required to

maintain consensus is reduced, transactions can be executed faster and the throughput increases. Also, the system was developed to run on three different consensus mechanisms, which are BFT-SMaRt, Linear BFT-SMaRt and the Hotstuff-inspired version of BFT SMaRt, which also improves the scalability and efficiency of the system since those mechanisms can be optimised for specific needs of the platform. The use of a permissioned blockchain also allows to optimise the performance of the system by tuning such blockchain parameters as block size and transaction fees. Also it is worth noting that in a permissioned blockchain it is possible to control data visibility and implements privacy features. This makes the data processing more efficient, as well as it allows a better managing of sensitive data, which results in a better user experience.

4.3 Use Case of the Model

The workflow of the model can be described in the following steps.

1. **Vehicle registration and identity verification.** Vehicle owners register their EVs on the Fabric-based platform, providing vehicle information and proof of ownership. The platform verifies the vehicle identity and ownership using smart contracts and stores the vehicle information on the distributed ledger.
2. **User registration and verification.** Users register on the platform and undergo identity verification to ensure they meet the minimum age and driving license requirements. The platform also verifies user identity using smart contracts and stores the user information on the distributed ledger.
3. **Vehicle booking and payment.** Users can browse available vehicles on the platform, select a vehicle, and make a deposit payment using fiat currency or cryptocurrency. The payment is processed through a payment gateway integrated with the Fabric platform, and the payment information is stored on the distributed ledger using smart contracts.
4. **Vehicle access and usage.** Vehicle access is managed through a secure mobile app provided by the platform. The app grants the user access to the vehicle and provides information about status, location, and battery level of the vehicle. The vehicle usage information is stored on the distributed ledger using smart contracts, which also record the time and location of vehicle usage.
5. **Payment and settlement.** At the end of the rental period, the platform automatically calculates the rental fee based on the usage time and distance. Payment is executed using fiat currency or cryptocurrency, and the payment information is stored on the distributed ledger using smart contracts. A portion of the fee is automatically deducted for maintenance and repair services.
6. **Maintenance and repair.** The platform manages maintenance and repair services for the shared vehicles, including battery replacement, tire replacement, and general maintenance. Funds for maintenance and repair are managed through a common pool, with vehicle owners and users contributing based on usage time and vehicle performance. The maintenance and repair information is stored on the distributed ledger using smart contracts.

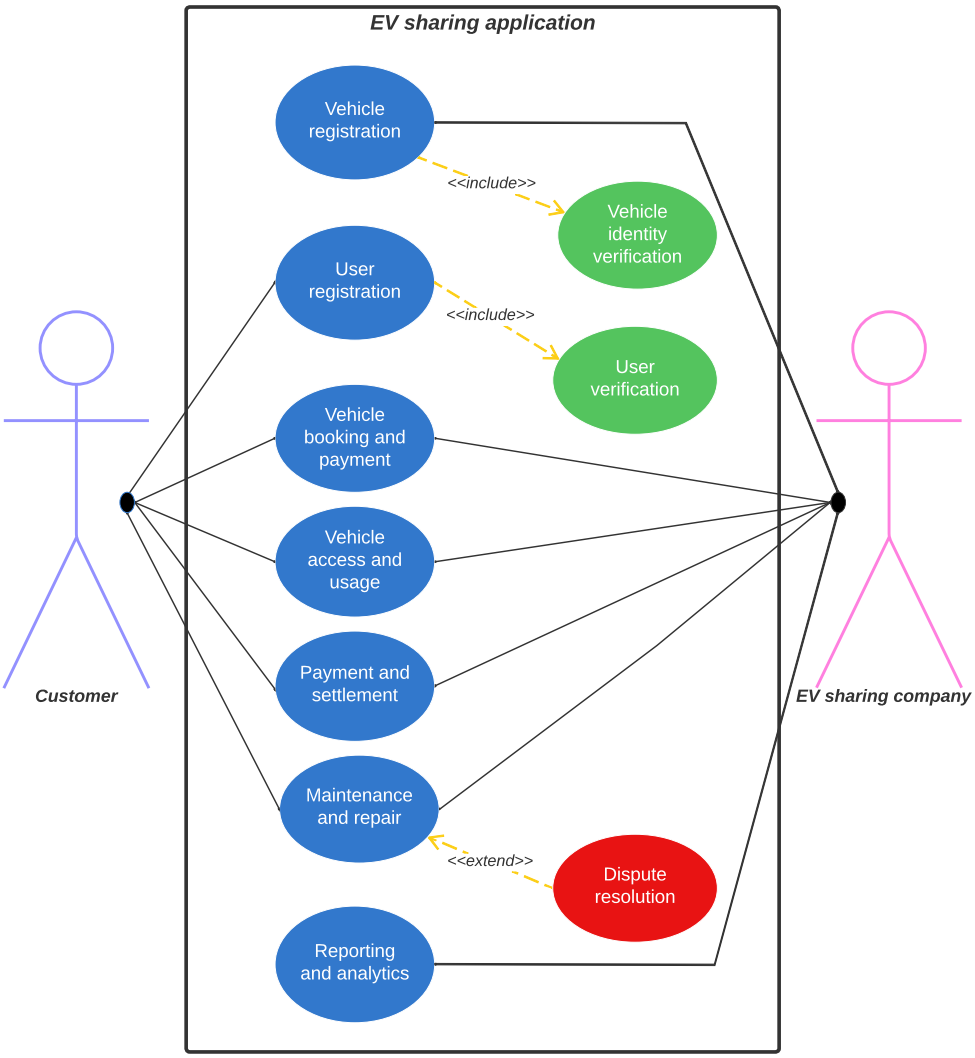

Figure 4.1: The model's use case diagram

7. **Dispute resolution.** In case of disputes, the platform provides a transparent and decentralized mechanism for dispute resolution, using smart contracts to enforce dispute resolution rules. The dispute resolution information is stored on the distributed ledger for transparency and auditability.

8. **Reporting and analytics.** The platform generates reports and analytics on vehicle usage, maintenance and repair costs, and revenue generated from rental fees. These reports and analytics are available to all stakeholders on the platform for transparency and decision-making.

Figure 4.1 shows the use case diagram for the model.

Chapter 5

System Design

5.1 System Architecture Overview

The designed system can be divided on four components - communicators, which are users, vehicles and an EV sharing company, an application which provides a User Interface (UI) such that communicators can send requests, a blockchain network which stores the logged data, and a consensus mechanism under which the blockchain network runs. Note that an EV is also a communicator in the system, since it has a built-in IoT device which sends data to an application backend.

After an application backend receives and processes a request sent from communicators through the UI (or directly in case it is EV), it creates a transaction and sends it to the blockchain network. The blockchain then validates the transaction and executes a smart contract that corresponds to the request. Then, the transaction is added to the candidate block which is then validated with the use of a consensus mechanism, which makes sure that the nodes in the network agree on accepting the block. Eventually, if the block is valid, it is appended to the blockchain. The blockchain network also sends updates to the application backend, which is then passed to communicators through the UI or directly to the EV. Figure 5.1 illustrates the operation flow within the system.

5.2 Application Design

The application for the system consists of two parts - backend/API and UI. Through UI users and vehicle owners send their requests, that are later processed by the backend. The backend processes requests from UI or from IoT devices in the EVs and sends them to smart contracts. It also provides feedback from the blockchain to the EVs and to UI, so that users could see it.

5.3 Blockchain Design

It was chosen to use the Hyperledger Fabric blockchain network, since it is a private permissioned blockchain, which means only the users who has a special permission

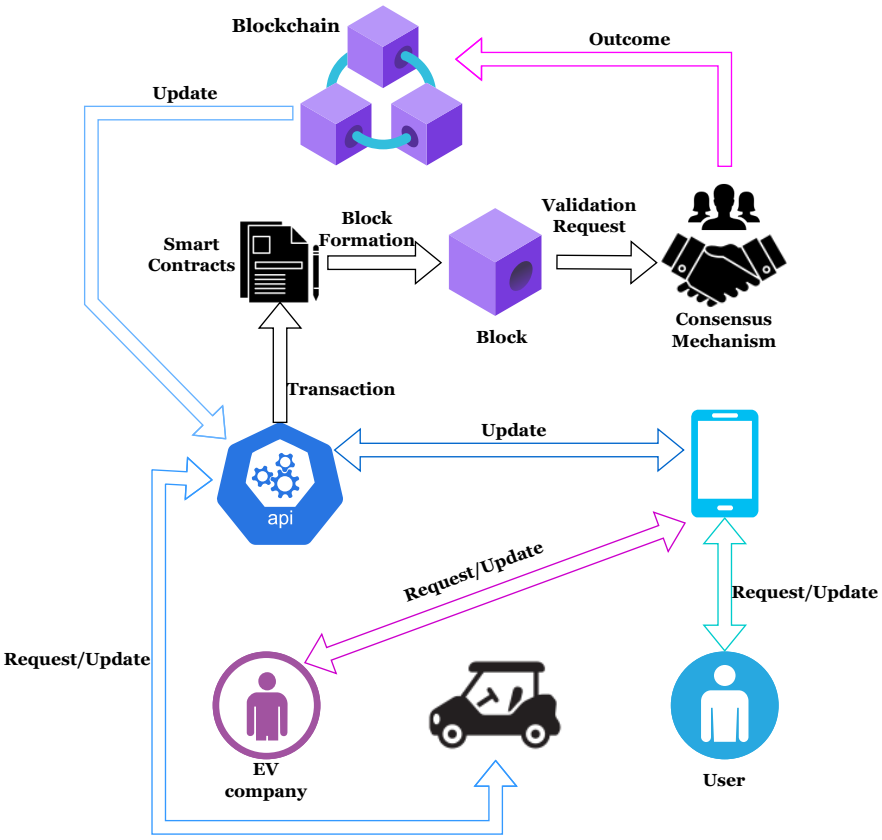

Figure 5.1: Operation flow in system architecture

would be able to join and act in it. A potential user would first need to register and buy a custom token to be able to interact with the system further.

Also, Fabric is a suitable choice since it allows any consensus mechanism to be plugged into it (Figure 5.2), and three consensus mechanisms are considered for the use. A certain algorithm might be chosen to be plugged based on the current needs for the system. Generally, modular architecture of Fabric allows to configure various components such as smart contracts and consensus mechanisms, which is a useful feature.

5.4 Consensus Design

As mentioned earlier, the system is meant to function under the three different consensus mechanisms, which are BFT-SMaRt, Linear BFT-SMaRt and the Hotstuff-inspired version of the BFT-SMaRt. An overview of the BFT-SMaRt has already been given in Subsection 2.2.3. In this section, the other two consensus mechanisms are described. The implementations of both Linear BFT-SMaRt and the Hotstuff-inspired version of the BFT-SMaRt were provided as testbeds for this project.

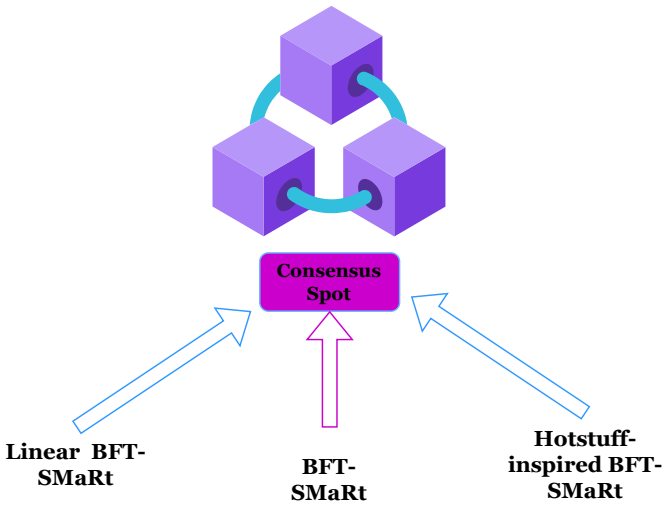

Figure 5.2: Pluggable Consensus in Fabric Blockchain

5.4.1 Linear BFT-SMaRt

Linear BFT-SMaRt was implemented by Meng Kun in his Master's Thesis. The goal was to modify the original BFT-SMaRt library by mainly integrating the Practical Threshold Signatures scheme [26] into the consensus mechanism in order to achieve linearity in normal case operation.

5.4.2 Hotstuff-inspired Linear BFT-SMaRt

A consensus mechanism that in this report is referred to as Hotstuff-inspired BFT-SMaRt is a modification of BFT-SMaRt described in [7]. The key difference is that in Hotstuff-inspired BFT-SMaRt the view-change operation is merged with the normal case operation in the same way it is done in Basic Hotstuff, a consensus mechanism described in Subsection 2.2.4.

It is important to note that Hotstuff-inspired BFT-SMaRt is not completely equivalent to the Basic Hotstuff, since it is built on and hence inherits some features of the original BFT-SMaRt. However, the difference is not significant and the performance of the algorithm is not affected.

5.4.3 Comparison of the Consensus Mechanisms

Consensus Mechanisms	Best-case Latency	Normal-case Communication	View-Change Communication	View-Change Responsive
BFT-SMaRt	2	$O(n^2)$	$O(n^2)$	yes
Linear BFT-SMaRt	2	$O(n)$	$O(n^2)$	yes
Hotstuff-inspired BFT-SMaRt	3	$O(n)$	$O(n)$	yes

Table 5.1: Performance comparison of the three consensus mechanisms

Table 5.1 displays the comparison of the three mentioned consensus mechanisms based on different performance characteristics. As illustrated, Linear BFT-SMaRt improves

the performance of the original BFT-SMaRt achieving linear complexity in the normal-case operation, however the view-change process is still $O(n^2)$. Hotstuff-inspired BFT-SMaRt achieves a linear complexity for both normal-case and view-change processes, however it pays for that the same price that Basic Hotstuff does, which is having one more round trip than previously described consensus algorithms, which worsens the best-case latency. It is also worth noting that all the algorithms are view-change responsive.

Chapter 6

Implementation

The tool described in this section was implemented in order to simulate the system proposed in chapters 4 and 5.

6.1 Overview of the Implementation

The tool was implemented as a package which was added into three libraries that implement the three consensus mechanisms, which are original BFT-SMaRt, Linear BFT-SMaRt and Hotstuff-inspired BFT-SMaRt. For each of the three consensus mechanisms, separate GitHub repositories were created. It is important to note that for the original BFT-SMaRt, the stable branch 1.2 was downloaded from the official github repository cite repo instead of the master branch, since Gradle was not required to build the project when using the code from the stable branch, which means that the configuration for the original BFT-SMaRt library is the same as for the other two consensus libraries.

The package, in which the tool was implemented, was placed into the *demo* folder, alongside with other programs that demonstrate how a consensus mechanism is run. The design of the tool inherited the key design features of the other programs in the *demo* folder. The tool partially consists of four java files, which are responsible for client-server communication. These files are *EVInteractiveClient*, *EVclient*, *EVserver* and *EVRequestType*. The functionality of these files can be described as follows.

The *EVInteractiveClient* file implements a class of the same name which is responsible for interaction with the user of the tool. It presents options to the user and, based on the choice of the user, takes all the necessary input from the user and calls a function from the *EVclient* file that matches the request of the user.

The *EVclient* file implements an eponymous class which contains functions that correspond to options in the EVINTERACTIVECLIENT class. The constructor of the class takes the current client id and creates an object of SERVICEPROXY class from the tom package, that is responsible for total order multicast. In every function, BYTEARRAYOUTPUTSTREAM and OBJECTOUTPUT objects are created (except the CLOSE function which is responsible for terminating the execution if such option was chosen in the

EVINTERACTIVECLIENT). The OBJECTOUTPUTSTREAM was used to write in and then flush the request type that corresponds to the function and parameters that were passed to the function. It is worth mentioning that all the request types are listed in the enum EVREQUESTTYPE that is implemented in a file of the same name, where each request type correspond to an option in the EVINTERACTIVECLIENT. Then, a function calls the INVOKEORDERED function of SERVICEPROXY class. This function takes a byte array and sends ordered requests to all replicas, hence starts interacting with the EVSERVER class.

The EVSERVER file implements the class EVSERVER that extends the DEFAULTSINGLERECOVERABLE class, which is a class that provides a basic state transfer protocol. The constructor of the class initialises a replica based on the passed ID value and a logger that is used to log messages, as well as creates two HASHMAP objects that store information about registered users and registered vehicles using user or vehicle ID as a key. It then creates an EVSERVER object in the MAIN function. The class also overrides two methods of the DEFAULTSINGLERECOVERABLE class. The APPEXECUTEORDERED function executes operations which have to be ordered first or, in other words, that require a consensus to be reached prior to their execution. The APPEXECUTEUNORDERED function executes operations which do not require consensus to be reached. Since all the functionality of the tool require a consensus to be reached, all the operations were implemented in the APPEXECUTEORDERED function while the APPEXECUTEUNORDERED function was not used and returns null. In the APPEXECUTEORDERED function, the request type and other data received from a client, and then a certain option is executed based on the request type. After processing one of the options, the function returns a reply, which is a byte array message.

Eventually, a function in EVCLIENT class receives the reply and if it is empty returns null. Otherwise, it creates an object of BYTEARRAYINPUTSTREAM based on the reply and an object of OBJECTINPUTSTREAM, that are used to read out what was returned by replicas and then pass the reply to the EVINTERACTIVECLIENT. The result is then printed in EVINTERACTIVECLIENT.

The tool also contains other three files, namely User, Vehicle and Quartet. User and Vehicle files implement eponymous classes that contain parameters and methods necessary to represent users and vehicles in the system.

The USER class contains such fields as USERID, USERBALANCE, IDSOFVEHICLESUSED and CURRENTVEHICLEACCESSCODE. USERID stores an ID of the user, which is a 9 digit numerical String value, and USERBALANCE stores a float value that represents the balance of the user. IDSOFVEHICLESUSED is an ArrayList that stores vehicle IDs of all the vehicles used by the user and is initially empty. CURRENTVEHICLEACCESSCODE stores an access code to the vehicle that is currently booked or used by the user, initially it is an empty String. All the fields are initialised in a constructor and have corresponding getter and setter methods.

The VEHICLE class contains such fields as VEHICLEID, VEHICLEACCESSCODE, VEHICLEOWNERBALANCE, ISAVAILABLE, CURRENTUSERID, DEPOSITPRICE, VEHICLEPRICEPERHOUR, VEHICLEPRICEPERKM, VEHICLEREPAIRPERCENTAGEOFFEE and IDSOFUSERSTHATUSEDVEHICLE. VEHICLEID and CURRENTUSERID store an

ID of the vehicle and an ID of the user that is currently using the vehicle respectively, which are 9-digit numerical String values. VEHICLEACCESSCODE stores an access code for the vehicle that is randomly generated. ISAVAILABLE stores a boolean value that indicates if a vehicle is currently available and is initially true. VEHICLEOWNERBALANCE and DEPOSITPRICE store float value for the balance of the owner of the vehicle and an integer value to indicate the deposit price to be paid when the vehicle is booked, respectively. VEHICLEPRICEPERHOUR and VEHICLEPRICEPERKM store integer values that indicate a price per each hour and a price per each kilometer respectively. IDSOFUSERSTHATUSEDVEHICLE is an ArrayList that stores user IDs of all the users that have used the vehicle and is initially empty. Finally, VEHICLEREPAIRPERCENTAGEOFFEE stores an integer value that represents a percentage of deposit price that is automatically deducted for maintenance and repair services when a vehicle gets booked. All the fields are initialised in a constructor and have corresponding getter and setter methods.

The file *Quartet* implements an eponymous helper-class that creates a tuple of four elements, each of which can be of any data type. It was chosen to implement this class from scratch, since surprisingly it was hard to find a java library that implements such a data structure. Considering the fact that the class has a very basic functionality, it was faster to implement it from scratch rather than importing third-party libraries.

It is also worth mentioning that the *User*, *Vehicle* and *Quartet* files were serialized, which has solved a problem with getting IOEXCEPTION and CLASSNOTFOUNDEXCEPTION. Since functions in EVCLIENT has used byte streams and had to deal with objects of custom classes, these classes had to be serialized to convert states of the objects into byte streams.

An important note to add is that all random numbers were generated using a RANDOM object from JAVA.UTIL.RANDOM library, since it allows to create random seeds when generating random numbers. Hence, the same random numbers will be generated within each seed in every program execution. This was very important for the evaluation part of the project, since it has allowed to use the same numbers when the tool was tested under different consensus mechanisms.

6.2 Implementation of the Use Case

The eight procedures described in the Section 4.3 were compressed into five commands that could be inputted through the console. A user of the tool can pick one of those commands or options, which are **register a vehicle**, **register a user**, **book a vehicle**, **return a vehicle**, and **dispute**.

If **register a vehicle** option is chosen, the program asks to provide the ID of the vehicle, the balance of the owner of the vehicle, the deposit price, price per hour, price per kilometer and a percentage of the deposit price, which is a maintenance and repair fee. Based on those values, a Vehicle object is created which is then passed to the REGISTERVEHICLE function of EVCLIENT class, from where it is passed to the servers, as described in the previous section. If the vehicle is already registered, a server decines

this operation by sending a negative reply. Otherwise, the reply is positive and the Vehicle object gets appended to the VEHICLESREGISTERED HashMap.

An option **register a user** is executed in an almost identical fashion to **register a vehicle** option. The only difference is that the tool user is asked to input the User ID and the initial account balance of the user, after which a User object is created and is passed to the REGISTERUSER function of EVCLIENT class. Then it is passed to servers, and if the reply is positive, this User object gets appended to the USERSREGISTERED HashMap.

If **book a vehicle** option is chosen, the program ask to provide a user ID and a vehicle ID of the vehicle to book, and passes them to the BOOKVEHICLE function of EVCLIENT class, where an ABSTRACTMAP.SIMPLEENTRY object is created to represent a pair of the two ID values and is passed to the servers. If both user and vehicle are registered and the vehicle is available, a server updates the details of both user and vehicle by changing the appropriate values of the fields in the objects stored in VEHICLESREGISTERED and USERSREGISTERED that correspond to the passed IDs. A transaction is simulated by subtracting the deposit price value from the user balance and adding that value to the balance of the owner of the vehicle. Then a positive reply is returned by servers.

In case of **return a vehicle** option, a tool user is asked to provide a user ID, a Vehicle ID, the time in hours for how long the vehicle was used and the distance that the user drove for in kilometers. Both time and distance are represented by float variables. The four values are passed to the RETURNVEHICLE function of EVCLIENT class, where an object of QUARTET class is created to store the values. This object is then passed to the servers. If a vehicle with such vehicle ID is registered and the user ID provided corresponds to the user ID of the current vehicle user, the vehicle price is calculated by the following formula:

$$price = pricePerHour * time + pricePerKm * distance - depositPrice +$$

$$+ \frac{percentageFee * (pricePerHour * time + pricePerKm * distance - depositPrice)}{100}$$

Then, it is randomly determined if the vehicle needs repair. If it does, an additional repair cost, which is also randomly generated, is added to the final price. For simplicity, it was decided to generate an additional repair cost such that it lies in a range between the deposit price of the vehicle and that price multiplied by two. After the final price is determined, it is subtracted from the balance of the user and added to the balance of the vehicle owner, which simulates a transaction. Then, all the necessary fields of vehicle and user objects are updated or initialised, and re-stored in the corresponding HashMaps. Then, servers return a positive reply.

If **dispute** option is chosen, a tool user is asked to provide a user ID and a Vehicle ID of the vehicle that was used, and passes them to the DISPUTE function of EVCLIENT class, where an ABSTRACTMAP.SIMPLEENTRY object is created to represent a pair of the two ID values and is passed to the servers. If both user and vehicle are registered, as well as the user ID is stored in the history of users of the vehicle and vice versa, an outcome of the dispute is randomly generated. If the outcome is positive, the user gets a compensation. For simplicity, it was assumed that the compensation is a refund that is equal to the deposit price of the vehicle. Then the compensation is subtracted from the

balance of the owner of the vehicle and added to the balance of the user. Then, objects and the corresponding HashMaps are updated. If the outcome of the dispute is negative, there are no changes made. Eventually, a positive reply is returned by servers.

It is important to note that several major assumptions were made for the sake of simplicity of the simulation. Firstly, it was assumed that users of the system is honest. For instance, it is assumed that users inputs true values for usage distance and time when returning a vehicle or that a user does not select the **dispute** option if they already got the compensation. Secondly, it was assumed that both users and owners of vehicles have enough money to make transactions, since it would have required a lot of extra functionality to be implemented, while making a tool more complex was not the goal of the project. Thirdly, the program does not handle all the cases of an incorrect input to the console, since the goal was to simulate the system and hence it was expected that a user knows the format of the input and does not make mistakes. Lastly, the prices are given in pounds, hence it is assumed that the system uses fiat currency for transactions, however the choice of currency is purely conditional and can be substituted with any other currency, since this is just a simulation and no real transactions take place. As a potential extension, a token could be introduced, as well as functionality to enable users buy these tokens to then use them as an internal currency.

6.3 Running and Testing the Tool

As metioned earlier, the tool was designed to be plugged into each of the consensus libraries. In order to run the tool within each consensus library locally, configurations in the Intellij IDE were edited as follows. Firstly, four configurations were added as applications, each of which ran an EVSERVER class and hence simulated the behaviour of replicas. It is important to note that the tool can be run on a bigger number of replicas, and hence more applcications could be added. Each EVSERVER application was given a number from 0 to 3 as a program argument. Then, a compound was also added as a configuration to run all EVservers one after another with just one click. Finally, another application was added as a configuration to run the EVINTERACTIVECLIENT class, which was given a nominal client ID 1001.

Since the tool is not complex, it was not hard to test all the possible scenarios by interacting with the tool. It was made sure that during the interaction that all the IF statements were triggered and that the program handles all the edge cases that it was designed to handle.

Chapter 7

Performance Evaluation

7.1 Experiment Design

The design of the experiments was inspired by the code written in the benchmark package, which was used to measure the latency of the consensus algorithm when two trivial operations, put or get, are executed. All the necessary experiment functionality was implemented in *EVThroughputLatencyClient* and *EVThroughputLatencyServer* files, which were placed into the *EVsharing* folder along with the tool implementation files. The idea can be explained as follows. The implemented EVTHROUGHPUTLATENCYCLIENT class was designed to substitute EVINTERACTIVECLIENT and EVCLIENT classes by generating random data and randomly choosing an option instead of asking a user to provide an input, and then to take latency measurements. The *EVThroughputLatencyServer* class was designed to follow almost the same functionality as the EVSERVER class, but also to have some extra functionality to make throughput measurements.

The EVTHROUGHPUTLATENCYCLIENT class consists of the public MAIN function and a private helper class CLIENT. The MAIN function takes in the parameters of the class, namely the number of clients, number of operations per client and the size of the request, and then creates a Client object for each client and processes each client in a separate thread. It also generates a random seed, such that in every execution generated random values are the same, so that there is no bias when testing for different consensus algorithms. The helper CLIENT class consists of a constructor, the RUN function and the same functions that were implemented in the EVCLIENT class which correspond to use-case options. The constructor of the class initialises the fields using the passed parameters, as well as creating a new SERVICEREPLICA object and setting the timeout to 100 seconds. The RUN function generates a random option number and executes it for every operation. It also times each option execution using SYSTEM.NANOTIME() method and then calculates the average latency once all operations were processed. Every option follows the same functionality as in EVINTERACTIVECLIENT and EVCLIENT classes for the tool implementation described in Section 6.1, except that the values are randomly generated instead of being inputted by a user.

The EVTHROUGHPUTLATENCYSERVER class was implemented in an almost identical way to the EVSERVER class, except that it keeps track of message senders and the number of requests in the beginning of the APPEXECUTEORDERED function, as well as times the execution of the function body. Those values are then used in the function PRINTMEASUREMENT, which is called at the end of the APPEXECUTEORDERED function. The PRINTMEASUREMENT function calculates an average and a maximum throughput for every two seconds of the execution and prints a corresponding message.

Values for both vehicle and user IDs were generated as 9-digit numerical values. Values for the balances of both users and owners of vehicles were generated in the range from 100 million to a billion to ensure that they have enough balance to make transactions. The deposit price of a vehicle was generated in the range from a thousand to 10 thousands pounds. Both vehicle price per hour and vehicle price per distance were generated in the range from 5 to 50 pounds. It was chosen to generate a percentage for repair services of a vehicle in the range between 1 and 5 percent. Lastly, both the time spent using a vehicle and distance driven using a vehicle were generated in the range between 1 to 100 hours and kilometers, respectively.

7.2 Measurements and Analysis

7.2.1 Measurement Plan and Expectations

The aim was to observe how average throughput and latency of the three consensus algorithms change with an increase in the amount of nodes in the network, as well as for different request sizes, and compare the results. It was chosen to have three options for the request size, which are 0, 128 and 1024 bytes, and the maximum number of faulty replicas was set to 1, both as in the evaluation part in [31]. The number of clients was chosen to be 1 and the number of operations per client was set to 1000, making it 1000 operations in total. The number of clients, the number of operations per client and a value for the request size were passed as arguments to a created EVTHROUGHPUTLATENCYCLIENT application, which was run to simulate a client and to measure average latency in milliseconds.

It was chosen to take the measurements for 4, 8, 16, 32 and 64 nodes. The replicas were run by executing several compound configurations in Intellij, each of which consisted of an amount of EVTHROUGHPUTLATENCYSERVER applications which corresponded to one of the chosen number of nodes and measured the average throughput in operations per second. It is worth noting that every time the tool was tested on different number of nodes, the *currentview* file had to be deleted to refresh current view and the number of servers had to be updated, as well as their IDs had to be re-listed in the *system.config* file.

It was expected that latency would increase and the throughput would decrease with an increase in the number of nodes, as well as with an increase in the request size, for all the consensus algorithms. It was also expected that Linear BFT-SMaRt would outperform the original BFT-SMaRt, while the Hotstuff-inspired version of BFT-SMaRt was expected to outperform both Linear BFT-SMaRt and the original BFT-SMaRt in

terms of throughput. It was interesting to determine whether the Hotstuff-inspired version of BFT-SMaRt would outperform the other two consensus mechanisms in terms of latency too, even though its best-case latency parameter is equal to three round trips (as it was shown in Table 5.1) which is bigger than in other algorithms.

7.2.2 Results

Since the laptop on which the experiments were run locally has appeared to be not powerful enough to run experiments for 32 and 64 nodes, it was decided to take measurements for 4, 8, 16 and 24 nodes. The results are presented in Figures 7.1, 7.2, 7.3 and 7.4.

As it was expected, the average throughput is decreasing and the average latency is increasing with the increase in the number of nodes for all the algorithms and request sizes. Also, at some point, Hotstuff-inspired BFT-SMaRt outperforms both the original BFT-SMaRt and Linear BFT-SMaRt, even in terms of latency. Linear BFT-SMaRt outperforms the original BFT-SMaRt in terms of latency for bigger number of nodes, which also was expected; however, in terms of throughput, the two consensus mechanisms perform roughly the same for a bigger number of nodes, which was not expected. It is also remarkable how the original BFT-SMaRt outperforms the other two algorithms for 4 nodes but then its performance worsens with an increase in the number of nodes.

When the performance of each algorithm was tested for different request sizes, the most interesting outcome to observe was that in almost all the cases algorithms performed better with a non-zero request size, which was not expected. A potential explanation to this is that the process of handling empty messages or requests consumes more resources than normal processing of a non-zero requests. However, for very big request sizes, algorithms would still be expected to perform less efficiently than for a 0 bytes request size.

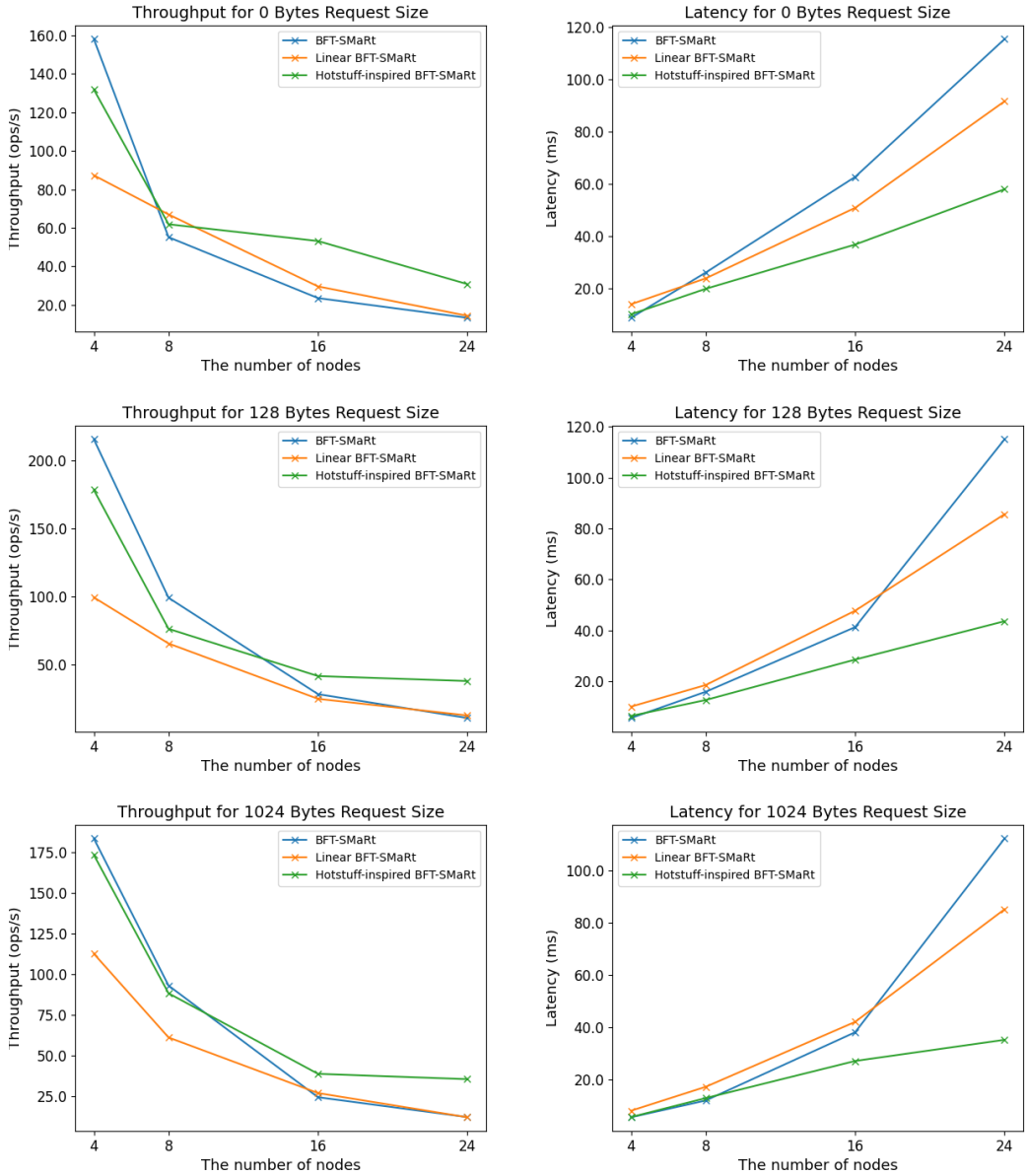

Figure 7.1: Performance comparison of the three consensus algorithms for different request sizes.

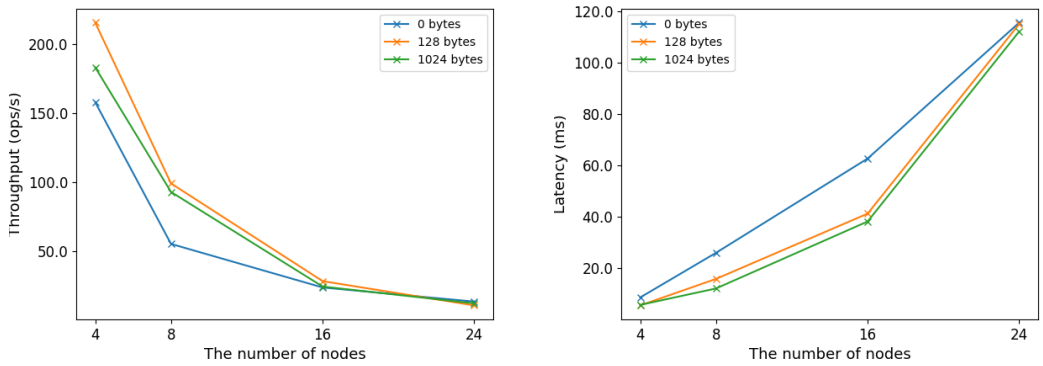

Figure 7.2: How different request sizes affect the performance of BFT-SMaRt.

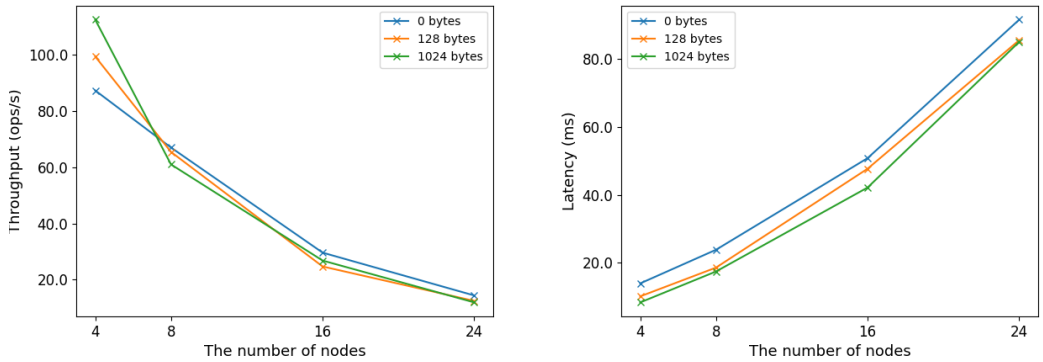

Figure 7.3: How different request sizes affect the performance of Linear BFT-SMaRt.

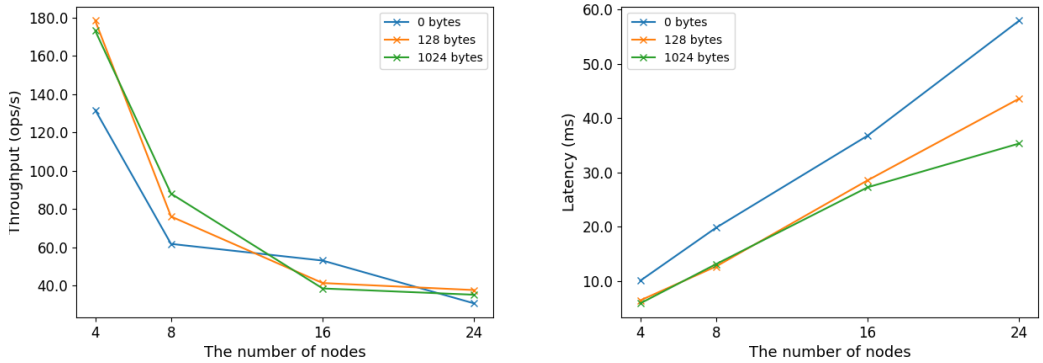

Figure 7.4: How different request sizes affect the performance of Hotstuff-inspired BFT-SMaRt.

Chapter 8

Conclusion and Future Work

8.1 Achievements

In this project, a blockchain-based EV sharing management model was proposed and a potential design of a system based on the model was introduced. A proposed system provides a solution to EV sharing management which uses various advantages of blockchain technology, which are decentralized architecture and transparency of the system. A tool was then successfully designed and implemented in order to simulate the functionality of the designed system. Lastly, several experiments were designed and implemented to test the produced tool under the three consensus mechanisms: the original BFT-SMaRt, Linear BFT-SMaRt and Hotstuff-inspired BFT-SMaRt.

During the execution of experiments, several results were obtained. It was determined that Hotstuff-inspired BFT-SMaRt outperforms the other two algorithms for a bigger number of nodes in terms of both average throughput and latency, as well as for various request sizes. Also, it could be observed that the original BFT-SMaRt outperforms Linear BFT-SMaRt for a smaller number of nodes, however, after roughly 16 nodes, they perform similarly in terms of throughput and Linear BFT-SMaRt performs better than the original BFT-SMaRt in terms of latency. Lastly, it could be seen that all three algorithms overall perform better when the request size is non-zero but not very large, than when the request size is zero.

8.2 Limitations

Most of the limitations of this project come from the time constraint. Too much time was spent on a research of the consensus libraries. As a result, there was no time to implement a more advanced version of the developed tool. For instance, if there was more time available, a more complex backend could be developed and linked to an existing blockchain, as well as some frontend could be implemented. Also, an evaluation part of the project is limited to testing the overall operation in terms of throughput and latency with only one varying parameter, which is request size. More testing based on more different metrics could be produced.

Another limitation is the lack of resources. It would not be possible to fully implement and test the designed system without the use of EVs with IoT devices or a collaboration with some EV companies.

8.3 Future Work

Most of the ideas for potential extensions for this project come from the previously descibed limitations. The whole system could be implemented by creating a complex backend or API, and UI, connecting it to a real blockchain system with a functioning consensus mechanism and implemented smart contracts. It could then be attempted to commercialize the system by attracting users and EV sharing companies.

In terms of evaluation of consensus algorithms, more complex experiment designs could be introduced even for an implemented tool, for instance, similar to the ones carried out in [31]. View-change and normal-case operations could be tested separately, as well as more parameters could be altered for testing such as the batch size, number of faulty replicas, number of clients and number of operations per client.

Bibliography

[1] Muneeb Ali and Zartash Afzal Uzmi. Csn: A network protocol for serving dynamic queries in large-scale wireless sensor networks. In *Proceedings. Second Annual Conference on Communication Networks and Services Research, 2004.*, pages 165–174. IEEE, 2004.

[2] Elli Androulaki, Artem Barger, Vita Bortnikov, Christian Cachin, Konstantinos Christidis, Angelo De Caro, David Enyeart, Christopher Ferris, Gennady Laventman, Yacov Manevich, et al. Hyperledger fabric: a distributed operating system for permissioned blockchains. In *Proceedings of the thirteenth EuroSys conference*, pages 1–15, 2018.

[3] Alysson Bessani, João Sousa, and Eduardo EP Alchieri. State machine replication for the masses with bft-smart. In *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 355–362. IEEE, 2014.

[4] Ethan Buchman, Jae Kwon, and Zarko Milosevic. The latest gossip on bft consensus. *arXiv preprint arXiv:1807.04938*, 2018.

[5] Vitalik Buterin et al. A next-generation smart contract and decentralized application platform. *white paper*, 3(37):2–1, 2014.

[6] Miguel Castro, Barbara Liskov, et al. Practical byzantine fault tolerance. In *OsDI*, volume 99, pages 173–186, 1999.

[7] Xiao Chen, Btissam Er-Rahmadi, Tiejun Ma, and Jane Hillston. Parbft: An optimised byzantine consensus parallelism scheme. *IEEE Transactions on Computers*.

[8] Wikimedia Commons. Illustration of the byzantine generals problem. `https://upload.wikimedia.org/wikipedia/commons/f/fc/Byzantine_Generals.png`, 2007. File: Byzantine_Generals.jpg.

[9] Noam Copel and Tal Ater. Dav white paper. *tech. rep*, 2017.

[10] Himadry Shekhar Das, Mohammad Mominur Rahman, S Li, and CW Tan. Electric vehicles standards, charging infrastructure, and impact on grid integration: A technological review. *Renewable and Sustainable Energy Reviews*, 120:109618, 2020.

[11] W Entriken, D Shirley, J Evans, and N Sachs. Non-fungible token standard, document erc-721, sep. 2018, 2018.

[12] Tiago M Fernández-Caramés and Paula Fraga-Lamas. A review on the use of blockchain for the internet of things. *Ieee Access*, 6:32979–33001, 2018.

[13] Guy Golan Gueta, Ittai Abraham, Shelly Grossman, Dahlia Malkhi, Benny Pinkas, Michael Reiter, Dragos-Adrian Seredinschi, Orr Tamir, and Alin Tomescu. Sbft: a scalable and decentralized trust infrastructure. In *2019 49th Annual IEEE/IFIP international conference on dependable systems and networks (DSN)*, pages 568–580. IEEE, 2019.

[14] Long He, Ho-Yin Mak, Ying Rong, and Zuo-Jun Max Shen. Service region design for urban electric vehicle sharing systems. *Manufacturing & Service Operations Management*, 19(2):309–327, 2017.

[15] Mike Hearn and Richard Gendal Brown. Corda: A distributed ledger. *Corda Technical White Paper*, 2016, 2016.

[16] Luqman Hussain and Adil Bashir. Hirego white paper. `https://web.archive.org/web/20180304202631/http://www.hirego.io/lib/HireGo_Whitepaper.pdf`, 2018.

[17] Srdjan Krco, David Cleary, and Daryl Parker. P2p mobile sensor networks. In *Proceedings of the 38th Annual Hawaii International Conference on System Sciences*, pages 324c–324c. IEEE, 2005.

[18] Jae Kwon. Tendermint: Consensus without mining. *Draft v. 0.6, fall*, 1(11), 2014.

[19] Leslie Lamport, Robert Shostak, and Marshall Pease. The byzantine generals problem. In *Concurrency: the works of leslie lamport*, pages 203–226. 2019.

[20] Xin Luan, Lin Cheng, Yang Zhou, and Fang Tang. Strategies of car-sharing promotion in real market. In *2018 3rd IEEE International Conference on Intelligent Transportation Engineering (ICITE)*, pages 159–163. IEEE, 2018.

[21] Satoshi Nakamoto. Bitcoin whitepaper. *URL: https://bitcoin. org/bitcoin. pdf-(: 17.07. 2019)*, 2008.

[22] Karen Rose, Scott Eldridge, and Lyman Chapin. The internet of things: An overview. *The internet society (ISOC)*, 80:1–50, 2015.

[23] Fahad Saleh. Blockchain without waste: Proof-of-stake. *The Review of financial studies*, 34(3):1156–1190, 2021.

[24] Julio A Sanguesa, Vicente Torres-Sanz, Piedad Garrido, Francisco J Martinez, and Johann M Marquez-Barja. A review on electric vehicles: Technologies and challenges. *Smart Cities*, 4(1):372–404, 2021.

[25] David Schwartz, Noah Youngs, Arthur Britto, et al. The ripple protocol consensus algorithm. *Ripple Labs Inc White Paper*, 5(8):151, 2014.

[26] Victor Shoup. Practical threshold signatures. In *Advances in Cryptology—EUROCRYPT 2000: International Conference on the Theory and Application of Cryptographic Techniques Bruges, Belgium, May 14–18, 2000 Proceedings 19*, pages 207–220. Springer, 2000.

[27] Peter Triantafillou, Nikos Ntarmos, S Nikoletseas, and P Spirakis. Nanopeer networks and p2p worlds. In *Proceedings Third International Conference on Peer-to-Peer Computing (P2P2003)*, pages 40–46. IEEE, 2003.

[28] Viktor Valaštín, Kritian Košt'ál, Rastislav Bencel, and Ivan Kotuliak. Blockchain based car-sharing platform. In *2019 International Symposium ELMAR*, pages 5–8. IEEE, 2019.

[29] Fabian Vogelsteller and Vitalik Buterin. Eip 20: Erc-20 token standard. *Ethereum Improvement Proposals*, 20, 2015.

[30] Dylan Yaga, Peter Mell, Nik Roby, and Karen Scarfone. Blockchain technology overview. *arXiv preprint arXiv:1906.11078*, 2019.

[31] Maofan Yin, Dahlia Malkhi, Michael K Reiter, Guy Golan Gueta, and Ittai Abraham. Hotstuff: Bft consensus in the lens of blockchain. *arXiv preprint arXiv:1803.05069*, 2018.

[32] Qihao Zhou, Zhe Yang, Kuan Zhang, Kan Zheng, and Jie Liu. A decentralized car-sharing control scheme based on smart contract in internet-of-vehicles. In *2020 IEEE 91st Vehicular Technology Conference (VTC2020-Spring)*, pages 1–5. IEEE, 2020.