Interplanetary Smart City

Marcel Zak

A dissertation submitted in partial fulfilment
of the requirements for the degree of
Master of Engineering
of the
University of Aberdeen.

Department of Computing Science

2018

Declaration

No portion of the work contained in this document has been submitted in support of an application for a degree or qualification of this or any other university or other institution of learning. All verbatim extracts have been distinguished by quotation marks, and all sources of information have been specifically acknowledged.

Signed:

Date: 2018

Abstract

This dissertation reports the development, testing and evaluation of Interplanetary Smart City (IPSC), a fully decentralised peer-to-peer system designed to allow secure data exchange. This system can be deployed in a variety of scenarios, from smart homes to interplanetary smart cities. I proposed, developed and evaluated a new data exchange mechanism. It is built with the use of Hyperledger Fabric v 1.0 blockchain technology. IPSC enables secure, robust, reliable, scalable and space aware communication without a central point of failure. The evaluation shows that the system is able to provide all the mentioned advantages without the need for the traditional trusted third party. However, it also shows that blockchain mechanism and consensus protocol introduces additional overhead. Therefore, such a system is not suitable for time-critical or real-time applications.

Acknowledgements

I would like to express my sincere thanks and gratitude to all those who helped me with this project. First of all, I would like to acknowledge the unfailing support, expertise and enthusiasm of my supervisor, Professor Wamberto Vasconcelos. Then I would like to thank for all the support and care of my lovely wife Andrea Zak.

Contents

Chapter 1

Introduction

The challenges of cities are changing. As we use more of that finite resource of clean drinking water, as we create more waste and use more energy, we have to think very differently how to solve the problems. In 2014 United Nations estimated that 54 % of the world's population lived in urban areas and predicted that the number increases to 66 percent by 2050 [1]. Air pollution in cities is growing. World Health Organization estimated that in 2012 died 3.7 million people because of outdoor air pollution exposure [2].

So what is a smart city? Is it a solution to these problems? In 2018 there is no agreed definition of a smart city. In [3] is a smart city defined as "an urban area that uses different types of electronic data collection sensors to supply information which is used to manage assets and resources efficiently." These data collections help to improve life, health, comfort and resource management of the city. They can help to use resources more efficiently and bring new insights to issues we are facing. For example, they may help to improve public transportation and traffic light control. As a result, it can indirectly decrease air pollution. The same approach can be used in homes, villages or even on a global and interplanetary scale. In 2024 first crew should begin their mission to Mars[1] and set up first Mars base. It will be the first chance for humanity to build an interplanetary city and eventually a self-sustaining civilisation. An interplanetary smart city will require communication technology that is secure, scalable, distributed and aware of the physical distance between information and request location.

This dissertation reports the development, testing and evaluation of Interplanetary Smart City (IPSC), a fully decentralised peer-to-peer (P2P) system designed to allow secure data exchange between parties. This system can be deployed in a variety of scenarios from smart homes to interplanetary smart cities. The project aims to explore the possibilities of utilising blockchain technology, the best ideas from multiple P2P protocols and ideology of IPFS [4]. Such an application can bring numerous advantages in comparison to traditional cloud-based solutions. IPSC should allow secure, robust, reliable and space aware communication without a central point of failure and possible savings on server hosting and cloud services.

1.1 Motivation

Data are, supposedly, the currency of the Internet age. Companies are increasingly allowing payments for their digital services with information rather than money [5]. Majority of the existing

[1] http://www.spacex.com/mars

technologies used in smart cities do not provide any mechanisms to trade collected data easily. This approach worsens all mentioned criteria for security, reliability, robustness and no physical distance awareness. On the other hand, it is beneficial to the companies providing these services. They have easy and usually free access to the data collections. In order to use such services as Oracle Cloud IoT[2], Google Cloud IoT[3], Salesforce IoT[4] or Microsoft Azure IoT[5] devices must be connected to the Internet. This dependency is a major disadvantage because, in a case of Internet connection failure, we cannot access our data. Furthermore, if the service provider is hacked or experiencing technical issues, data collections can be stolen or inaccessible. This poses high dependency on the service provider and Internet connectivity. One example of service provider dependency can be Logitech that decided to intentionally brick all Harmony Link devices remotely on 16 March 2018 [6].

The next issue is that IoT devices need to communicate with each other often. The standard client-server model introduces a bottleneck and increases latency. Moreover, it is a single point of failure. On the contrary, P2P network is ideal for a smart city. The workload is spread across multiple devices and requested data can be retrieved directly from the closest local node that is in possession of them. In the case of a future interplanetary city, the physical distance of required data is critical. A radio wave requires between 4 to 21 minutes to travel between Earth and Mars. This depends on the position of the planets. Furthermore, interplanetary space is a very different environment. High-energy ionising particles (electrons, heavy ions and protons) of the space environment causes Single Event Effects (SSE) such as Single Event Upset, Single Event Transient, Multiple Bit Upset and many other destructive and nondestructive SSE. They cause an arbitrary behaviour of electronics and corruption of memory [7]. From these examples, it is clear that the current approach of IoT and a smart city communication is not suitable for the future use.

1.2 Objectives

Given the current issues with Smart Cities and IoT presented above, the objectives of the project are:

1. **Development of a data exchange mechanism that does not require third, trusted party.** The main goal of this research is development of an exchange mechanism using distributed ledger and peer-to-peer technology. This mechanism must not require trusted third party in order to exchange paid or unpaid data between two parties.

2. **Testing and evaluation of the developed solution.** The second goal is to test and evaluate the developed solution in terms of performance.

3. **Modularity and Scalability** To simplify future work, the system should be modular and using microservice architecture. It means that the solution is a collection of loosely coupled services, that implement business capabilities and scales well horizontally.

[2] https://cloud.oracle.com/iot
[3] https://cloud.google.com/solutions/iot/
[4] https://www.salesforce.com/products/salesforce-iot/overview/
[5] https://www.microsoft.com/en-us/internet-of-things/azure-iot-suite

Chapter 2

Background & Related Work

This chapter is dedicated to background and work related to this project. The chapter covers advantages and disadvantages of several peer-to-peer mechanisms and current approach for communication between IoT devices in a smart city. Moreover, high-level description of blockchain mechanism with different approaches is provided.

2.1 Smart City

As I mentioned in the Chapter 1 a smart city can be defined as "an urban area that uses different types of electronic data collection sensors to supply information which is used to manage assets and resources efficiently." [3]. These data collections help to improve life, health, comfort and resource management of the city. The difference between a regular city and a smart one is in sensors that gather all sort of data and AI (artificial intelligence) algorithms analyse the data and propose improvements to the current system. One example of a device that has already positively impacted cities is a smart rubbish bin called Bigbelly[1]. It uses sensors to estimate the amount of rubbish inside and it sends a request for collecting the rubbish if it is getting full. Moreover, it can serve as public Wi-Fi hot-spot or broadcast useful information via Bluetooth.

IoT devices and other data collection sensors in cities connect to a network mainly via Wi-Fi, Bluetooth, LoRa, 6LoWPAN, 4G or Ethernet technology. These devices are built to be energy-efficient because often a battery is the only source of power available in a place of operation. Therefore, they rarely have a lot of computing power and, thus, are unsuitable for peer-to-peer nodes.

2.2 Peer-To-Peer

Peer-To-Peer (P2P) system is different from a client-server system. Devices connected in a P2P system behave as both, server and client at once. Therefore, every peer in the P2P system can share some of its resources. Be it bandwidth, storage capacities, files or computing power (CPU/GPU cycles). Nodes are devices connected to the P2P system. They are often considered unreliable and even untrusted. The P2P system is generally a virtual overlay network on top of the physical topology of the network in which the node is connected. The application layer peers are able to communicate directly via the logical overlay network. Moreover, this communication network can be without central control. Based on the architecture of the overlay network we distinguish between three types of P2P systems. The first is centralised, the second is decentralised and the third

[1] http://bigbelly.com/

is hybrid. Furthermore, based on the topology, a decentralised P2P system can be unstructured or structured [8]. See Figure 2.1.

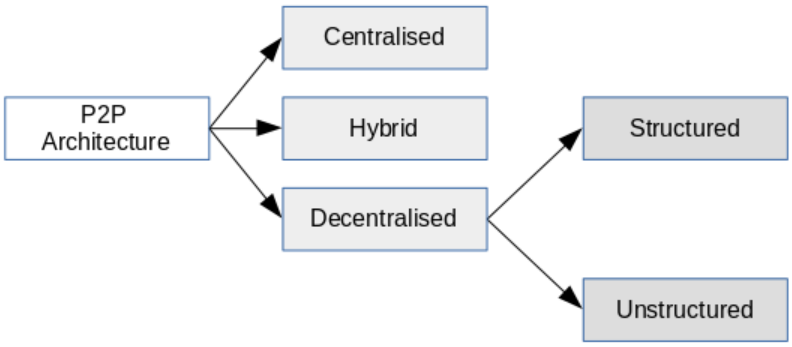

Figure 2.1: P2P Architecture

2.2.1 Centralised P2P system

Centralised P2P system combines both architecture patterns. Client-server and P2P. A node first connects to the central server, called a broker, to locate the desired resource on the P2P network or the server acts as a task scheduler that coordinates tasks among peers. The first example of such a network is well known Napster [9]. It was originally founded as P2P music sharing service. A node connected to the Napster asked for a location of the desired resource and the server sends an address that has it. Unlike client-server architecture, once a node has the address it communicates directly with the peer that holds the required data. See Figure 2.2. Napster was shut down and later reopened as a legal[2] music streaming service. The next examples are BOINC [10] or SETI@home [11]. In their case, the server acts as a task scheduler. Nodes fetch work units from the server directly. The advantages of this architecture are good control over the network, cheap discovery of peers that have the required resource because the server holds the central index of all peers and their resources. However, the disadvantages are similar to client-server architecture. The server is a single point of failure. This type of P2P network does not scale well with a large number of nodes connected to the network. Finally, centralised P2P networks are less robust in comparison to decentralised P2P networks, due to the mentioned single point of failure.

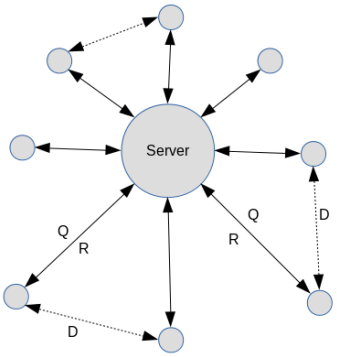

Figure 2.2: (Q) node queries the server. (R) response from the server. (D) data exchange.

[2] https://gb.napster.com/

2.2.2 Decentralised P2P system

Decentralised P2P system does not have any central index of resources and therefore, no central point of failure. Every node in the decentralised P2P system has equal rights and responsibilities. A node operates with only a partial knowledge about the network. If every node had complete information about all other peers in the network, every message would travel only one hop[3]. On the other hand, every node would have to maintain a routing table of an O(N) size (where N is a number of nodes in the network). This would be unrealistic (in P2P network of bigger size) because each join and leave of a node would need to propagate to every peer in the network. The other extreme is if a node would have only two connections to each other in a ring topology. The cost of maintaining routing table would be minimal, but routing performance would be O(N). Use of a decentralised system radically improves robustness and scalability in comparison to a centralised P2P system. However, it introduces a new challenge of locating the desired resource (discovery service) on the P2P network. Two main approaches attempt to solve this issue. Based on the architecture of peers connecting in the overlay network, the approaches can be divided into structured and unstructured decentralised P2P systems. [12]

Structured overlay network maintains placement of resources or pointers to nodes that have desired resources under predefined rules. Generally, in structured P2P network this knowledge is maintained as distributed hash table (DHT[4]) [13]. A DHT provides lookup service for connected peers in a similar way as a regular hash table. A node is responsible only for a subset of key-value pairs of the DHT in such a way that nodes joining and leaving cause a minimal amount of disruption. The number of connections that a node have with its peers (degree of a node) influences the routing performance and structure of the overlay network. For more information about routing algorithms and their properties see Appendix A.1.

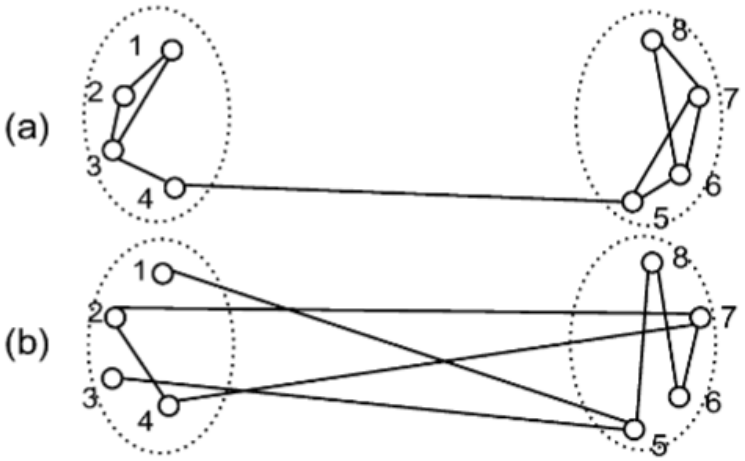

Figure 2.3: Illustration for locality-aware overlay and (b) randomly connected overlay [14]

Another important phenomenon affecting the performance of routing is locality. It is a relationship between overlay network and underlying physical network. If the physical network is not

[3]Hop – In P2P networks it represents how many times a message must be forwarded among nodes until it arrives to the desired destination node.

[4]https://en.wikipedia.org/wiki/Distributed_hash_table

taken into account while constructing the overlay network, peers that are neighbours in the overlay network can be far away and a message has to do many hops in the physical network. In addition, peers that are on the same physical network can be very distant in the overlay network and a message has to do many hops in the overlay network. Consequently, this results in undesirable network traffic. [14]. This problem is illustrated in Figure 2.3.

Unstructured overlay network utilizes different approach for forwarding queries between peers. Every node maintains only its own data and keeps track of its connected neighbours that it can send queries to. However, this maintenance of a list of neighbours comes with a huge cost of bandwidth. Approximately 55% of all traffic is due to PING and PONG messages that serve for maintaining the list of neighbours [15]. As the name suggests there is no underlying structure that maps resources to nodes. Therefore, it is challenging to locate the desired resource because it is difficult to predict which node has it. Other difficulties are that there are no guarantees of completeness of answer (unless the whole network is searched) and response time [8]. Examples of such P2P networks are famous Gnutella[5] and FastTrack[6]. There are two main algorithms used for implementing unstructured P2P networks. They are flooding and random walk.

The first routing strategy used in unstructured overlay P2P network is flooding. Consider an overlay network in Figure 2.4 where every node has degree between two and five (number of connected neighbours). With a higher degree, the distance between nodes reduces and a query has to do fewer hops in the overlay network. On the other hand, each node has to maintain a bigger list of its neighbours [12]. This list of neighbours can be shared between nodes. Once a node requires specific information it queries all its neighbours because it does not know the location of requested information. This process repeats further at each queried node until requested information is found or the maximum number of hops is reached. This limit for the maximum number of hops a message can do is called time-to-live (TTL). The TTL is a parameter of every message (query) and at each hop, it is decremented by one. It prevents queries circulates endlessly. Each node also keeps a list of queries that answered and if it receives the same query again it simply drops it [16].

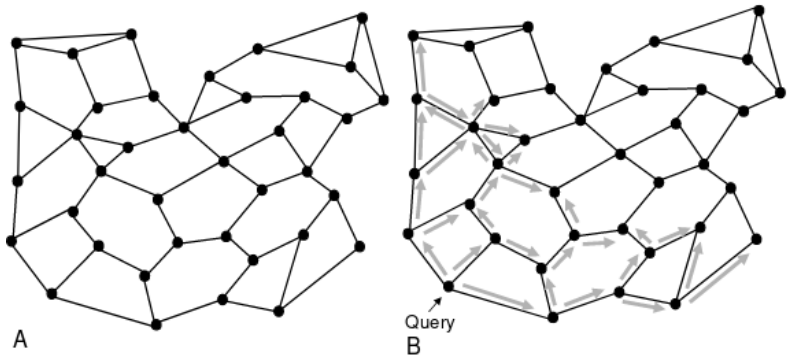

Figure 2.4: (A) Unstructured topology showing connections between peers and (B) query flooding to four hops. [12]

[5] http://rfc-gnutella.sourceforge.net/
[6] https://en.wikipedia.org/wiki/FastTrack

The second approach used for routing queries in unstructured overlay P2P networks is random walk algorithm. It is similar to flooding technique but it significantly decreases the communication cost. When a node issues or receives a query, it randomly selects neighbour (except the originator) and send the query further. However, the disadvantage is that the query processing time is very long. In order to improve the query processing time, the initiator could send k messages instead of only one [8]. See Figure 2.5.

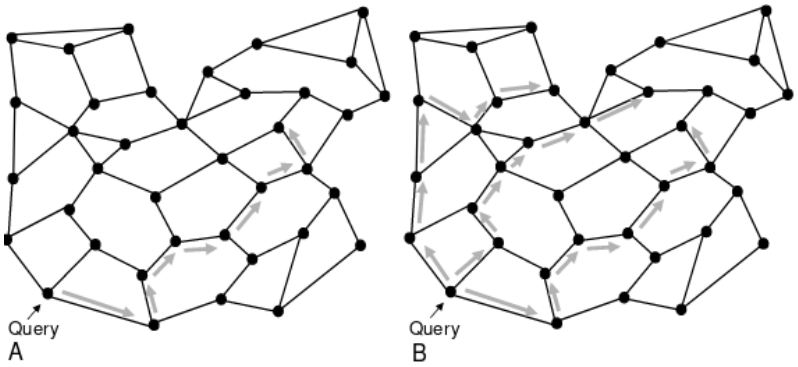

Figure 2.5: (A) Random walk and (B) k-way parallel random walk, k=3. [12]

2.2.3 Hybrid P2P system

Hybrid P2P system combines both, centralised and decentralised P2P systems. The main advantage of the centralised P2P system is fast lookup time but it has scalability issues. On the other hand, decentralised P2P system scales better but it requires longer time in resource locating. In Hybrid systems a node can be selected as *super node* also known as *super peer* and serve other nodes as a server. There can be many criteria for selection of *super node*. Be it bandwidth, number of connections, longevity and many others. Therefore, resource locating can be done in centralised fashion (through supernodes) and also decentralised fashion [8]. It is clear that different P2P systems have their advantages and disadvantages. Therefore it is crucial to choose the right one or even combination of multiple approaches.

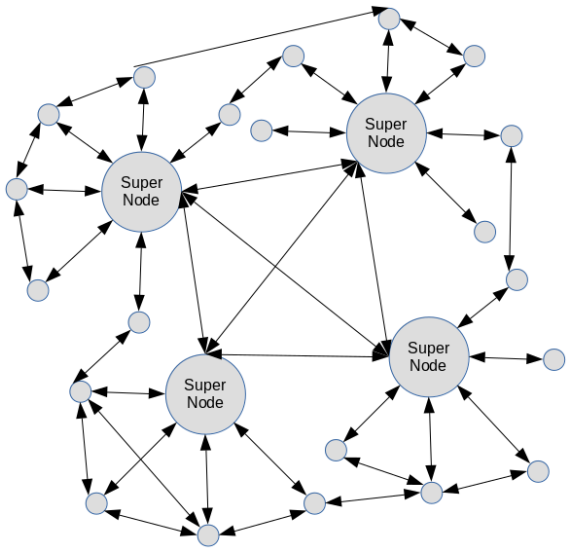

Figure 2.6: Hybrid P2P overlay network. Super nodes act as server for peers.

2.3 Distributed ledger

Distributed ledger technology (DLT) is a consensus of shared, synchronised and replicated records (financial or non-financial) spread across multiple geographic locations. There is no central data storage or single central authority. Users of DLT use it to settle transfers of data, money or assets without the need for trusted central authority. In traditional point of view, the central authority can be a financial institution such as a bank. DLT allows spreading the trust among participants instead of the traditional third party. There are different types of DLTs. We can distinguish DLTs based on participation in the ledger, validation method and data structure of the shared records [17].

2.3.1 Permissioned VS Permissionless DLTs

There are two types of participation in the ledger. The first is unrestricted, also known as permissionless. The second is restricted, also known as permissioned[18]. In a permissionless DLT anybody can participate in ledger update and validation. By definition, permissionless DLT is public, meaning that the ledger is publicly available. On the other hand, in permissioned DLT only authorised participants can validate and update the ledger. Permissioned DLT can have private or public ledger [17].

The choice of permissionless or permissioned DLT have an effect on maintenance costs as well as the range of possibilities to enforce truthful behaviour of validators. In permissioned DLTs, validators are known and they can be punished for malicious behaviour. It can be outside of the DLT (e.g. legal contracts, fines, etc.) as well as inside the DLT (e.g. disqualification from validation process, credibility, etc.). However, in permissionless DLTs validators are unknown and may be punished only inside the DLT. Therefore, permissionless DLTs use tools based on game theory to discourage potential adversarial participants [18]. Table 2.1 shows trade-offs between different types of DLT architectures. However, this table is only a simplified version. While choosing the right DLT, it is crucial to get an excellent understanding of the specific DLT and its technical details.

Table 2.1: Trade-offs between different types of DLT architectures(simplified)

Properties of DLT	**Permissioned**	**Permissionless**
Speed	Faster	Slower
Energy efficiency	Better	Worst
Scalability	Better	Worst
Censorship resistance	No	Yes
Tamper - proof	No	Yes

2.3.2 Blockchain

In 2008 Satoshi Nakamoto published a white paper called "Bitcoin: A Peer-to-Peer Electronic Cash System" [19]. This paper revolutionized many industries. The idea of a cryptocurrency was not new but there was always a problem with double spend. Traditionally, this problem is solved via trusted third central authority, such as a bank, instead of cryptographic proof. Satoshi Nakamoto did not invent new cryptographic methods, but he combined existing cryptographic methods, from 80's and 90's, in a new innovative way that allowed the dawn of modern cryptocurrencies. This new invention is called a blockchain. Even though that in the original paper Satoshi

Nakamoto did not mention the word blockchain, he describes the underlying principles of keeping transactions in *blocks* that are *chained* together via cryptographic hash (specifically Bitcoin uses SHA256). He argued that the only way of preventing double spending is to know about all transactions. Even though that many people use terms blockchain and DLT interchangeably, it is considered to be only one type of DLT.

The fundamental principle of a generalised blockchain can be described as following. Records are grouped into blocks that one can imagine as a single page in a ledger. The next step is to create a cryptographic hash, such as SHA256, from this block and widely publishing it. The time when the hash is published is crucial. It proves that the records must have existed at the time to get into the hash. This is a timestamp of the block. Since each block contains a hash of the previous block, it forms a chain with each additional block reinforcing the ones before it. See Figure from the original Bitcoin paper [19].

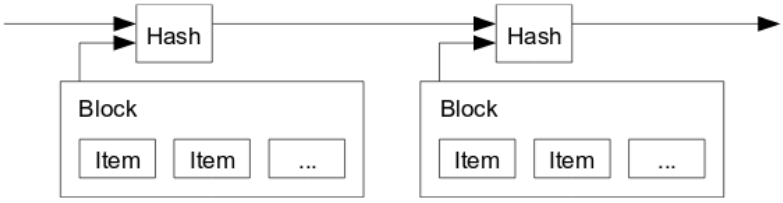

Figure 2.7: A chain of blocks with records [19].

In order to add a new block to the existing blockchain, the majority (in some cases at least 2/3) of the validators in the overlay P2P network have to agree on the newly created block. In permissioned blockchain, where system authenticates trusted validators, this proposal and validation of a new block is a straightforward process. On the other hand, in a permissionless blockchain where the validators are unknown and they can behave maliciously, we have to introduce countermeasures. An adversarial entity could create multiple nodes in the overlay P2P network and overvote the majority of honest nodes. This is known as Sybil attack[7]. To prevent this situation, Satoshi Nakamoto proposed a solution in a form of Proof-of-Work consensus algorithm. This is further discussed in Subsection 2.3.3

A public blockchain is by its definition traceable and linkable. Everybody can see and track every record, source and destination address from the first (genesis) block up to the latest one. The first cryptocurrency that implemented and used public blockchain is Bitcoin[8]. However, CriptoNote[9] introduced the idea of completely anonymous transactions on a public blockchain. This is achieved with help of multiple cryptographic methods such as ring signature and ringCT, stealth address, ITP router and Pedersen commitment [20]. Explanation of how this type of anonymous blockchain works is out of the scope of this work. Examples of such anonymous cryptocurrencies are Monero[10], AEON[11] and Bytecoin[12]. In these cryptocurrencies, it is very difficult (almost impossible) to find the address of the sender and receiver, observe any transaction or even

[7] https://en.wikipedia.org/wiki/Sybil_attack
[8] https://bitcoin.org/
[9] https://cryptonote.org/
[10] https://getmonero.org/
[11] http://www.aeon.cash/
[12] https://bytecoin.org/

determine how much money was sent. Nevertheless, it is possible to validate the transaction and prevent double spending.

2.3.3 Consensus Protocol

Consensus algorithm is essential for DLT. Since the ledger is distributed, it is crucial to reach consensus between nodes about every single transaction. This is exceptionally difficult in unreliable environments that P2P networks are. In fact, it turned out to be impossible to reach consensus even with one faulty process in an asynchronous environment where no assumptions about message delivery delays or relative speeds of processes are made [21]. This proof is known as FLP impossibility. Therefore, all consensus protocols used in DLT are partially synchronous. Meaning that there are hard deadlines for validators.

In the case of permissioned DLT, any consensus protocol can be used to replicate the machine state. For example, a famous protocol is Paxos[13]. This protocol is widely used in the industry for state machine replication. Its disadvantage is that it is difficult to understand and not Byzantine fault tolerant[14] (BFT) [22]. In 1999, Miguel Castro and Barbara Liskov published a paper "Practical Byzantine Fault Tolerance" (PBFT) that describes a new replication algorithm [23]. This was a breakthrough. Before, there was not fast and efficient BFT protocol that could be used in real life. A new effort has been made to further improve the existing BFT state machine replication protocols. Tendermint[15] is one example that is already implemented as a pluggable consensus protocol for blockchains [22]. Another example of a blockchain that is planning to provide PBFT protocol among others is Hyperledger Fabric [24]. These examples of consensus protocols solve only the issue of state machine replication. However, as it was mentioned previously, this would not be enough for permissionless DLTs due to the Sybil attack. In order to mitigate it, Satoshi Nakamoto proposed to use Proof-of-Work (PoW) for generating a new block on the blockchain [19].

Proof-of-work (PoW) is not a new idea. In 1992, Cynthia Dwork and Moni Naor invented the concept of "a computational technique for combatting junk mail in particular and controlling access to a shared resource in general" [25]. In short, the requester of a service first has to do some computational work in order to use the requested service. Essentially, it introduces a cost of accessing the requested service via economic measure because the user's hardware, time and electricity are not for free. In the case of permissionless DLT, a Proof-of-Work is used to prevent the Sybil attack and flooding of the P2P network with fake new blocks.

Bitcoin was the first cryptocurrency that used the idea of Proof-of-Work in the system. A node, in order to propose a new block, first has to solve a cryptographic puzzle. To be more specific, Bitcoin uses SHA256 cryptographic hash of a block (instead of a whole block it contains a Merkle tree root that is explained further). This hash of a newly proposed block must fulfil requirements of *difficulty* which is a specific number of leading zero bits in the hash. This is achieved by adding a nonce that can be altered into the input of the hash. See Figure 2.8 that shows how a previous hash, nonce and transactions creates a blockchain. With more zeros, the difficulty of a generating such a hash is exponentially increasing. The whole process is as follows.

[13] https://en.wikipedia.org/wiki/Paxos_(computer_science)
[14] https://en.wikipedia.org/wiki/Byzantine_fault_tolerance
[15] https://tendermint.com/

New transactions are broadcast to all nodes. Then each node collects new transactions into a block and it works on finding a difficult Proof-of-Work for its block. When a node finds it then it broadcasts the block with the hash to all nodes. Nodes should accept the block only if all the transactions are valid. They can easily verify that the work has been done by hashing the block and comparing those two hashes. Nodes accept the new block by using the hash of the accepted block as a previous hash in a next block in the chain [19]. This is a simplified description of consensus protocol with Proof-of-Work that Bitcoin uses.

Permissionless DLT works with an assumption that majority of validators are hones. In the case of Bitcoin and similar DLTs that utilize Proof-of-Work in their consensus protocol, not the majority of validators but the majority of the CPU power have to be honest. The longest chain represents the majority because there was invested the most of the Proof-of-Work effort. Therefore, the longest chain will grow the fastest and outpace any competing chains. If an attacker wants to modify a past block, she would have to redo the Proof-of-Work of the specific block and all blocks after it and overtake the chain produced by honest nodes. Satoshi Nakamoto showed that the probability of a slower attacker catching up decreases exponentially as subsequent blocks are added [19]. Moreover, validators (miners) also have economic incentive, in form of reward, to continue the work and remain honest. Imagine that an attacker is in control of more computing power than all the honest nodes. She would have to decide between using the computation power to defraud people or earn new tokens (reward for creating new blocks and/or transaction fees). Remember that she would still have to spend a considerable amount of her own resources due to Proof-of-Work. By using it to steal back her payments, she would depreciate the market value of the tokens. Since this would be an undesirable situation for the attacker, she is disincentivised to do so.

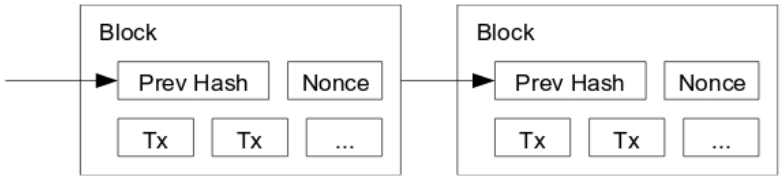

Figure 2.8: A chain of blocks with records that also contains a hash of the previous block and nonce [19].

Merkle tree is a cryptographic hash tree where every leaf node contains a hash of a data block and every node, that is not a leaf, contains a hash of its children's hashes [26]. Merkle tree is used to verify data that are transferred, stored or handled between computers or storages. It is often used in P2P networks (Gnutella[16]), file systems(Btrfs[17], IPFS[4], ZFS[18]) and DLTs (Bitcoin[19], Ethereum[19], etc.). The advantage of using Merkle tree, instead of hashing the whole file, is that single block of data can be verified faster without the need of having all blocks of the file. To be more specific, in the case of a blockchain, separate transactions are hashed. These hashes are leaves of the graph as shown in the Figure 2.9. Then it is not necessary to store all the transactions

[16] http://rfc-gnutella.sourceforge.net/
[17] https://btrfs.wiki.kernel.org/
[18] https://en.wikipedia.org/wiki/ZFS
[19] https://www.ethereum.org/

in a computer in order to verify a single transaction. This saves space and speed up verification process because the old blocks can be compacted by pruning the tree and leaving only the unspent transactions [19].

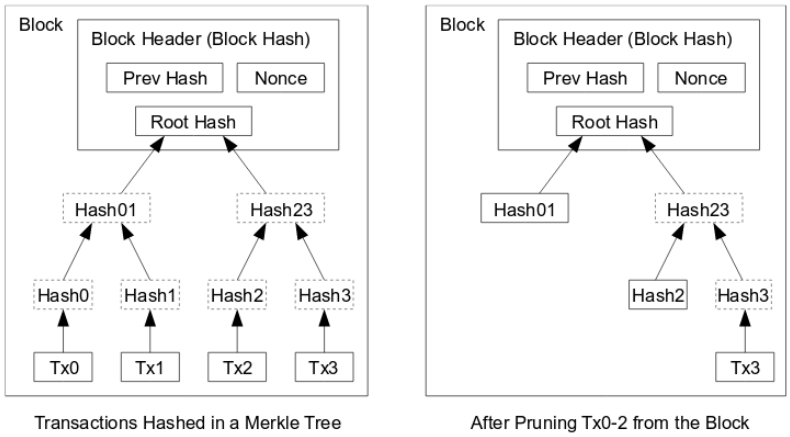

Figure 2.9: Transactions are hashed in a Merkle Tree and root hash is included in the block hash instead of the whole block. For verifying if Tx3 is part of the block, we need only root hash, hash01, hash2 and the Tx3 [19].

Proof-of-Stake (PoS) is another category of consensus algorithms that is used in permissionless DLTs. The basic concept is that validators have to make a secure deposit (bond some stake) if they want to participate in consensus. The weight of a vote depends on the size of validator's deposit (stake). In contrast to Proof-of-Work, Proof-of-Stake algorithms have significant advantages in terms of security, decreased the risk of centralization and energy efficiency. However, they have a major disadvantage that is "nothing at stake" problem discussed in Appendix A.2. The first cryptocurrency that used proof-of-stake (with the combination of PoW) is Peercoin [27]. Ethereum should also soon change its consensus algorithm to proof-of-stake. This new implementation is called Casper[20].

Above mentioned consensus protocols are the most used ones in DLTs. However, consensus protocols in DLTs are currently important research topic and we can expect new inventions and breakthrough. There are also proposed and being implemented many other categories of consensus protocols such as Proof-of-Space[21], Delegated Proof-of-Stake[22], Proof-of-Activity[23], Proof-of-Importance[24], Proof-of-Burn[25], Proof-of-Brain[26] and many combinations of them.

2.3.4 Smart Contract

Smart contract was first proposed by Nick Szabo in 1995. The article "Smart Contracts: Building Blocks for Digital Markets" was published in magazine Extropy in 1996 [28]. He defined a contract to be "a set of promises agreed to in a meeting of the minds, is the traditional way to formalize a relationship." Contracts are mainly used in business relationships. Moreover, they are also used

[20] https://github.com/ethereum/casper
[21] https://en.wikipedia.org/wiki/Proof-of-space
[22] https://bitshares.org/technology/delegated-proof-of-stake-consensus/
[23] https://eprint.iacr.org/2014/452.pdf
[24] https://www.nem.io/wp-content/themes/nem/files/NEM_techRef.pdf - section 7
[25] https://en.bitcoin.it/wiki/proof_of_burn
[26] https://en.wikipedia.org/wiki/Steemit

in personal relationships such as marriages, politics and other areas [28]. Szabo predicted in his article that digital revolution will dramatically change the traditional contracts. He called this new digital contracts "smart contracts".

The basic principle of a smart contract is that contractual arrangements between parties are written in a programming language. By storing this piece of code that defines the contractual arrangements into a blockchain, the smart contract becomes temper proof, self-executing and automatically enforceable. This reduces the need for the trusted third party and human intervention in the case of disagreements. Therefore the whole process of around traditional contracts is made less risky and more cost-effective. In order to create a smart contract, it must be able to be represented in logical flow such as "If X Then Y Else Z" [18]. Moreover, smart contracts must by deterministic otherwise nodes on the network would not reach consensus.

Ethereum was the first DLT that natively supported smart contracts. Ethereum provides a Turing complete virtual machine. It is called Ethereum Virtual Machine (EVM). Developers also created new programming language Solidity that is deterministic and suitable for smart contracts by its design [29]. Users can create their own contract, send them to Ethereum network where it is replicated via BFT algorithm. For every execution of the contract a small fee, called Gas, has to be paid. Then the smart contract is sequentially executed on every node on the network.

Hyperledger Fabric is another example of a DLT that support smart contracts. In contrary to Ethereum, Fabric is a permissioned blockchain. It rethinks the design and concepts used for permissionless DLTs and adapts them to suit better for permissioned DLT. It tackles existing limitations on permissionless DLTs such as execution throughput of smart contracts or the need for currency in public DLTs for smart contracts [30].

2.3.5 Distributed Ledger Technology in IoT & Smart Cities

With the rise in popularity of modern cryptocurrencies many researchers and companies started to explore the unexplored possibilities of DLT. During my research of related work I encountered numerous research papers and articles that made false claims or misunderstood the problematics. Often "Blockchain" technology is used in many proposed solutions without any proper explanation or logical reasons. However, In this section I will focus on the work I consider to be very useful and inspiring for this project.

In [31] authors did excellent summary of current blockchain technology and smart contracts. Moreover, they explored and discussed how smart contracts can be used in IoT and what should be considered for such deployment. One of the important things mentioned was that smart contracts are not legally binding (in permissionless DLT) and there is work being done towards solving this issue called "dual binding". On the other hand, I missed discussion about suitable consensus protocols for IoT devices. Despite this, I consider it to be well written paper that served to me as introduction to this problematic.

In [32] an interesting concept was proposed to integrate Low-Power IoT devices to a Blockchain. Since IoT devices are often powdered by battery and they have low computation power, they are not suitable for blockchain integration. Authors proposed decoupled model where IoT devices communicate with a gateway that is connected to a blockchain network and acts as a node. This gateway (a full blockchain node) can be queried remotely via a smart contract running on the blockchain. As a proof of concept they used private Ethereum network and decreased PoW

difficulty in order to reach faster speed. The code is available on GitHub [27]. It is a good example how low power IoT devices can be integrated to a blockchain. However, in a real world scenario, a private Ethereum network would not be the best choice. Since the network is private, there is no need for PoW consensus and existence of currencies. Therefore, more efficient solutions, such as Hyperledger Fabric [30] could be used. A discussion about this is missing in the paper.

One of the closest related work to this project is [33]. In this paper authors propose "a decentralized access model for IoT data, using a network architecture that we call a modular consortium architecture for IoT and blockchains." With this architecture, they aim to provide IoT data privacy via blockchains and address the challenges associated with implementing blockchains to IoT. A single blockchain that would be responsible for logging every IoT data operation would not scale well. Therefore, they broke down the network into a smaller private network called *sidechains*. These *sidechains* are interconnected via main *consortium* network. A consortium network runs own blockchain. It is responsible for access control and prevents any unauthorized access from one sidechain to another member of the consortium network. A sidechain is responsible for the creation of legitimate IoT records and access control of incoming request transactions. These records are further stored in the fully decentralised content addressed file system IPFS [4]. A hash of this record is updated in the smart contract that is responsible for the data transaction. Moreover, the same decoupled pattern of IoT devices and blockchain is presented. In this scenario, IoT devices hold encrypted communication with the gateway (called "validator node") that is running the smart contract.

However, authors proposed this solution where "The private IoT network consists of IoT devices and one validator node running the sidechain". There is no benefit or reason for running the sidechain with the smart contract on a single computer. Either, there should be multiple nodes on the sidechain for decentralisation (increased reliability) or there is no need for the sidechain because it is only a waste of resources. In addition, there is a problem with using IPFS address for content sharing. Either every node that validates this smart contract would gain access to this content for free (another slight variation of the tragedy of the commons[28]) or if the IPFS address was encrypted, it would be impossible to prove that the content was provided. Also, there is no implementation of the proposed solution, but only network traffic and processing overhead of existing candidate blockchain solutions. The evaluated blockchains were Monax and Ethereum.

Similarly in [34], authors proposed a solution where single blockchain is responsible for access control that is decoupled from storage layer. They consider IoT data to be streams and therefore introduced interesting concept of storing them in a specific data structure. Here are the data chunked, compressed and encrypted. Furthermore, these chunks are cryptographically chained together. This preserves the time-line of produced data. They also advocate distributed data storage for saving these data streams. They relay on P2P overlay network and DHT that serves as general-purpose private key-value data store interface. This paper presents satisfactory solution to IoT secure data storage with promising initial evaluation results. However, I think that this proposed solution would not scale well due to single blockchain. Moreover, it does not integrate payments for shared (sold) data.

[27] https://github.com/kozyilmaz/blocky
[28] https://en.wikipedia.org/wiki/Tragedy_of_the_commons

There are also other papers and new technologies that are related to this work and discusses IoT and blockchain integration in smart cities. These are :

- CitySense: blockchain-oriented smart cities[35]
- Mind my value: a decentralized infrastructure for fair and trusted IoT data trading[36]
- Towards an Optimized BlockChain for IoT[37]
- Slock.it[29]
- Filament[30]
- chain of things[31]

2.4 Summary

This chapter briefly explained background about P2P networks and a basic principle of DLT. It also introduces the most popular technologies that are used for blockchain technology. In addition, this chapter also critically evaluate and discusses the cutting-edge research papers about blockchain technology used in smart cities and IoT devices. All of this was necessary to understand before I could decide on the architecture, methodology and technology used for this project.

[29] https://slock.it/
[30] https://filament.com/
[31] https://www.chainofthings.com/

Chapter 3

Requirements & Architecture

The functional and non-functional requirements are specified and justified in this chapter. Both of them are sorted from the most to the least important requirements. Moreover, the system architecture and the data exchange procedure is described and justified against requirements.

3.1 Functional Requirements

The goal of this project is to provide a decentralised, secure, robust and physical distance aware solution for data exchange between IoT devices in an interplanetary smart city. The solution has to provide easy means for trade with the data collection among untrusted parties. To achieve this goal, I need functional requirements from FR1 to FR5.

FR1 - The solution will be fully decentralised peer-to-peer network with different options of connectivity.

In order to create a solution that does not have a central point of failure, it has to utilize a decentralised peer-to-peer network. This addresses the goal for robustness and physical distance awareness. The solution should not depend on specific connectivity technology because it would introduce limitation for future interplanetary cities. Different options for connectivity should include Internet, Wi-Fi, Ethernet, etc.

FR2 - Nodes should be able to exchange data without a centralised authority in a safe, secure, confidential manner and in an accountable fashion.

Since IoT data can consist of sensitive or valuable data, the parties must be able to exchange the data in a safe, secure and confidential manner. Accountability of the mechanism is a crucial part in the case of a dispute over a transaction between parties. These features should be achieved without the traditional trusted third party. This requirement is also necessary to achieve the goal of providing easy means for trade with data collections.

FR3 - The system should records every transaction between nodes.

In order to achieve data exchange in an accountable fashion, the system has to record every transaction in a tamper-proof log. This log will serve as a proof for involved parties. This requirement is necessary to provide easy means for trade among untrusted parties.

FR4 - There will be provisions for two nodes to pair up adequately based on multiple parameters.

Multiple data parameters can play a vital role in the decision which provider (node) serves

better, more suitable, data for a specific use. For example freshness of the data, bandwidth, latency, etc. In some cases, the frequency of data is more critical then accuracy, in others reliability plays the most crucial role. Therefore, the solution has to provide the possibility to pair nodes adequately.

FR5 - The node should be able to publish what data it can provide.
To provide an environment for data exchange, there has to be a way to get a list of available data and their parameters.

Due to the nature of this research topic, the above functional requirements (FR1 - FR5) are sufficient for reaching the stated goal. This is an open-ended cutting-edge research topic and very detailed functional requirements would limit exploration possibilities.

3.2 Non-Functional Requirements

I list and justify non-functional requirements (NFR1-3) here:

NFR1 - The solution will be efficiently scalable
The environment and size of interplanetary smart cities can vary. The solution should operate at many different scales. From smart houses to interplanetary cities. Therefore, it needs to be easily scalable.

NFR2 - The solution will be robust.
To be more specific, it will not fail as a result of individual components failing. To address the problem of central point of failure, the solution has to be able to continue working even if individual components fail. This improves the robustness of the whole system.

NFR3 - The solution will be resistant to a number of attacks.
To address the possible threats in smart cities, the solution must be resistant to multiple attacks that are common in P2P networks and IoT infrastructure. For example Sybil, Churn and DoS attack.

NFR4 - All the technologies used in this project should be free of charge.
All the technologies and tools should be free. However, they do not have to be necessary open source.

3.3 System Architecture

The following section describes high-level system architecture that is designed to fulfil the functional and non-functional requirements. The architecture diagram 3.1 aims to explain the underlying structure of the system and show flow of the data. The following sub-sections present each component and its role in the whole system.

Due to the importance of security and resistance against several attacks, I decided that the used DLT has to be private and not public. This improves performance because there is no need for Proof-of-Work or any other resource demanding algorithms.

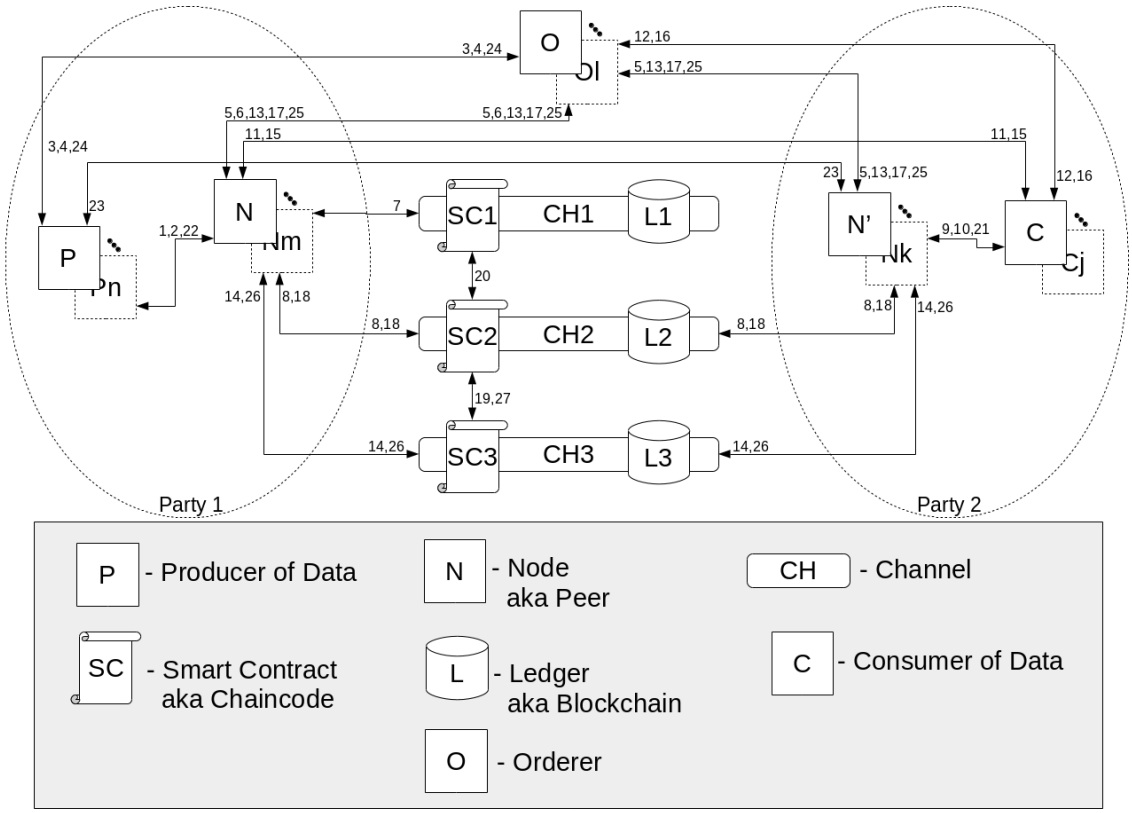

Figure 3.1: System Architecture

3.3.1 Producer of Data

The producer of data is an application that is captures data from one or multiple sensors and IoT devices. Moreover, it communicates with one or multiple nodes and orderers to query or update the state of the ledger. In the scenario that involves payments for data, it is also responsible for releasing tokens that were temporarily blocked until the transaction of data entry is finished. This process is further described in 3.3.8. This component is required to partially fulfil **FR2**.

3.3.2 Consumer of Data

Similarly, the consumer of data is an application that communicates with one or multiple nodes and orderers to update the state of the ledger. The consumer of data does not publish any data entry but it can request and pay for them. An application can be both, consumer and publisher of data entries at the same time. This component is required to partially fulfil **FR2**.

3.3.3 Ledger

A ledger, aka blockchain, immutably records all the transactions generated by smart contracts. It is hosted by a single or multiple nodes. Therefore it is distributed across them. Nodes are responsible for maintaining consistency of its state. This component is crucial for storing data and recording all transactions as a temper-proof log to completely fulfil **FR3**. In addition, this component together with the channel are required to completely fulfil **FR5** because without the storage a node could not publish anything.

3.3.4 Smart Contract

A smart contract, aka chaincode, is a code whose copy is held and executed by a node. This program execution can query or generate an update proposal for the ledger that is maintained by the node. A smart contract is invoked by an application (publisher or consumer). This component together with producer, consumer and channel, are required to completely fulfil **FR2**. Moreover, this component also completely fulfils **FR4** because a smart contract can be designed such that it provides functionalities for two nodes to pair up adequately based on multiple parameters.

3.3.5 Node

A node, aka peer, is a fundamental building block of the network because it hosts one or more smart contracts and ledgers. A node maintains consistency of ledgers and executes smart contracts. A node can maintain multiple ledgers and contracts. Furthermore, in comparison to orderers, it specialises in verifications of transactions. If an application wants to access a ledger, it has to do it via the node that maintains the ledger and has installed a copy of a smart contract. Because the overall design is a fully decentralised peer-to-peer network that uses multiple nodes, this partially fulfill the **FR1**. Furthermore, this component is also necessary to partially fulfil **NFR2** and **NFR1**.

3.3.6 Orderer

An orderer is a specialised type of a node. In contrast to a regular node, its role is to collect transaction proposals from applications (publisher or consumer), create and distribute new blocks to all nodes that maintain the state of the specific ledger. There can be a single or multiple orderers. In the case of multiple orderers, they use a consensus protocol to agree on the strict order and results of transactions. The consensus protocol is not hard-coded, but it is a pluggable module. Therefore, it also supports BFT[1]. This component, together with a node, completely satisfies **FR1**, **NFR1** and **NFR2**. Since the roles of a regular node and orderer are decoupled, this design provides a scalable decentralised solution.

3.3.7 Channel

A channel is a mechanism through which a set of nodes, orderers and applications within a blockchain network can communicate and transact privately. By joining a channel, the mentioned components agree to collectively share and manage identical copies of the ledger or multiple ledgers for the specific channel. This module is required to partially fulfil **FR2**, **FR5** and completely fulfil **NFR3**. It is worth noting that even though Figure 3.1 shows SC and L as if it was in the channel, it simply means that every component that joins the channel shares and manages an identical copy of the ledger.

3.3.8 Data Flow

This section describes scenarios in which data production, advertisement, purchase and retrieval take place. The complete data flow is graphically shown on the architecture diagram 3.1. The data flow is represented by arrows with numbers. These numbers are steps explained in the following list.

1. P creates data and contacts N (Party 1) using SC1 to generate a transaction proposal response

[1] Byzantine Fault Tolerant consensus protocol will play crucial role in an harsh environment such as interplanetary space is. SSE was briefly described in Chapter 1

for L1 on CH1 for new data entry. N (Party 1) responds to P with a signed endorsement for ledger update (new entry).

2. P contacts N (Party 1) using SC2 to generate a transaction proposal response for L2 on CH2 for new data entry advertisement. N (Party 1) responds to P with a signed endorsement for ledger update (new entry).
3. P sends the transaction to update L1 on CH1 with the signed and endorsed proposal from node N (Party 1) to O.
4. P sends the transaction to update L2 on CH2 with the signed and endorsed proposal from node N (Party 1) to O.
5. O creates a new block for L2 on CH2 and sends it to N (Party 1) and N' (Party 2). L2 contains a data entry advertisement (without values) produced by P.
6. O creates a new block for L1 on CH1 and sends it to only N (Party 1). L1 contains the data entry (with values) produced by P.
7. N (Party 1) validates the new block received from O and updates L1 on CH1.
8. N (Party 1) and N' (Party 2) validates the new block received from O and updates their copy of L2 on CH2.
9. C is notified by N' (Party 2) that a new data entry advertisement was created in L2 on CH2 and decides that it wants to purchase it.
10. C contacts N' (Party 2) to generate a transaction proposal response for L3 on CH3 to transfer tokens from C's account to P's account.
11. C also contacts N (Party 1) to generate a transaction proposal response for L3 on CH3 to transfer tokens from C's account to P's account. C must have a valid, signed endorsement from both N (Party 1) and N' (Party 2).
12. C sends the transaction to update L3 on CH3 with signed and endorsed proposals from node N (Party 1) and N' (Party 2) to O.
13. O creates a new block for L3 on CH3 and sends it to N (Party 1) and N' (Party 2). L3 contains account with tokens that are used for payments.
14. N (Party 1) and N' (Party 2) validates a new block received from O and updates their copy of L3 on CH3. Tokens that are transferred from C's account to P's account are marked as on hold because data entry value was still not revealed in L2 on CH2. Therefore, P cannot use the received tokens yet.
15. C contacts N (Party 1), with valid token transaction ID, to generate a transaction proposal response for L2 on CH2 to reveal data entry value for the advertisement.
16. C sends the transaction to update L2 on CH2 with signed and endorsed proposals from node N (Party 1) to O.

17. O creates a new block for L2 on CH2 and sends it to N (Party 1) and N' (Party 2).

18. N (Party 1) and N' (Party 2) validates the new block received from O and updates their copy of L2 on CH2. In order to finish this step, the validating nodes have to execute a special function in SC2 that invokes inter-channel communication described in next two steps.

19. SC2 query L3 via SC3 if the provided token transaction ID is already spent or still on hold. If it is still on hold (unspent for data purchase) then the flows continue.

20. SC2 query L1 via SC1 if the data entry ID is present in L1. If it is then the flow continues, data entry value is retrieved from L1 and updated into L2.

21. C receives revealed data entry value that paid for.

22. P contacts N (Party 1) using SC3 to generate a transaction proposal response for L3 on CH3 to release the tokens used for the revealed data entry value.

23. P also contacts N' (Party 2) using SC3 to generate a transaction proposal response for L3 on CH3 to release the tokens used for the revealed data entry value. C must have valid, signed endorsements from both N (Party 1) and N' (Party 2).

24. P sends the transaction to update L3 on CH3 with signed and endorsed the proposal from node N (Party 1) and N' (Party 2) to O.

25. O creates a new block for L3 on CH3 and sends it to N (Party 1) and N' (Party 2).

26. N (Party 1) and N' (Party 2) validate new block received from O and update their copy of L3 on CH3. In order to finish this step, the validating nodes have to execute special function in SC3 that invokes inter-channel communication described in next steps.

27. SC3 queries L2 via SC2, if the data entry value was revealed. If it is true then the tokens are released.

3.4 Summary

This chapter described and justified the system architecture that fulfils all the functional and non-functional requirements except **NFR4** that are completely fulfilled in Chapter 4. Furthermore, the Figure 3.1 describes system architecture and shows data flow. In addition, every component and its role in the system is described and justified against requirements. This chapter also shows how all the requirements are completely fulfiled, with the exception of **NFR4**, by various system architecture components. Moreover, complete data flow with typical scenarios is explained.

Table 3.1: Table of all system architecture components and their contribution towards functional requirements fulfilment. Where (X) represents contribution and (-) represents no contribution.

Component	**FR1**	**FR2**	**FR3**	**FR4**	**FR5**	**NFR1**	**NFR2**	**NFR3**	**NFR4**
Producer	-	✓	-	-	-	-	-	-	-
Consumer	-	✓	-	-	-	-	-	-	-
Ledger	-	-	✓	-	✓	-	-	-	-
Smart Contract	-	✓	-	✓	-	-	-	-	-
Node	✓	-	-	-	-	✓	✓	-	-
Orderer	✓	-	-	-	-	✓	✓	-	-
Channel	-	✓	-	-	✓	-	-	✓	-

Chapter 4

Methodology, Technologies & Implementation

The methodology, technology, development tools and implementation are further described in this chapter. In addition, I also explain the major reasons behind the decisions.

4.1 Methodology

This project aims to explore the possibilities of utilizing blockchain technology and the best performing concepts from multiple peer-to-peer protocols for interplanetary smart cities. This topic has gained much interest and has become an active field of research. As mentioned in Section 2.3.5, there are many different approaches to address these problems. Therefore, it is open research topic and the traditional software development techniques are not be suitable. Hence, I decided to adopt an exploratory programming approach [38], whereby I followed these steps:

1. Background literature reading and understanding the research topic.
2. Literature review of existing similar research projects.
3. Developing a project plan.
4. Learning about relevant software technologies, programming languages and tools used in this research area.
5. Establishing functional and non-functional requirements for the system.
6. Designing an over-arching architecture for the system which catered for the requirements.
7. Experimenting and exploring potential technologies, languages and tools suitable for the system.
8. Implementing and testing functionalities, integrating these with existing technologies.
9. Evaluating the developed system.

4.2 Technologies

This section describes the technology and tools that were used for the development of the system. It also presents motivation for adopting specific techniques and tools.

4.2.1 Distributed Ledger Technology

In terms of DLTs, I considered several popular options and assessed their suitability for the task. Since I knew that smart contracts are crucial for the whole concept, I immediately focused on DLTs that support them. The final candidates were Ethereum[1], EOS[2], and Hyperledger Fabric[3]. Even though Ethereum uses Proof-of-Work consensus protocol, it is possible to change it for a different one. For example, Tendermint[4] that is BFT consensus protocol for blockchains mentioned in Section 2.3.3. However, this approach would require a significant amount of work and time. Moreover, there would still be the need for coins (gas) to use smart contracts. In addition, there is no support for concurrent execution of smart contracts.

The second DLT is EOS that appeared to be an ideal candidate. It could be even more suitable than the chosen one because it is specially designed for the high performance of smart contracts from the beginning. However, it is still under a heavy development and in alpha phase at the time of writing this report. Therefore, to avoid unnecessary risk, I decided to use a more mature technology.

The last DLT is Hyperledger Fabric. It has a strong community support and a stable release version (v1.0 at the time of writing). In addition, there is no need for cryptocurrency to deploy smart contracts because it is designed as a private DLT. Furthermore, it was developed, since the beginning, for business purposes in mind. It supports concurrent execution of smart contracts, a node can have multiple of them and ledgers are separated by secure channels. Moreover, it uses a scalable P2P data distribution among peers. It is called Gossip data dissemination protocol[5]. Peers leverage gossip to broadcast ledger and channel data in a scalable fashion. For these reasons, I decided to use Hyperledger Fabric.

4.2.2 Scripting and Programming Language

Choosing the right programming language for a project is very important. This decision can improve readability, performance, future maintainability and decrease development time. For these reasons I carefully considered several languages that can be used for writing smart contracts for Hyperledger Fabric and also programming languages that can be used for interaction with the API of nodes.

For writing smart contracts I could use either Java[6] or Go[7] as programming languages. The native language of Hyperledger Fabric is Go and the majority of smart contract examples are written in this language. Even though I already knew Java, I realised that it would be better if the smart contracts are written in Go. In the case of some problems, I could ask in the community that always provides examples in Go, rather than in Java. Moreover, if I used only Go, there is no need to run JVM. This decreases the memory usage and increase the performance because running JVM introduces additional overheads[8]. For these reasons, I decided to learn a completely new language that I did not know before. It turns out that it was a great choice.

[1] https://github.com/ethereum/go-ethereum
[2] https://github.com/EOSIO/eos
[3] https://github.com/hyperledger/fabric
[4] https://tendermint.com/
[5] https://hyperledger-fabric.readthedocs.io/en/release-1.1/gossip.html
[6] http://openjdk.java.net/
[7] https://golang.org/
[8] https://en.wikipedia.org/wiki/Java_performance

For interacting with the smart contract API I could choose from four languages. These are Java, Node.js[9], Go and GNU Bash[10]. Again, I carefully considered the options and tried each of them by writing simple application that invokes a smart contract. Because the main focus of this project is not a development of a client application (publisher or subscriber) but the underlying mechanism, I decided to use Bash scripting language that interacts via CLI (command line interface). This allowed me to quickly adjust developed scripts during the phase of fast prototyping of smart contracts. Furthermore, any other language can be used for the development of a client application in the future.

4.2.3 IDE and Compiler

For the development of smart contracts, I decided to use Microsoft Visual Studio Code[11] because I am using it every day. In addition, it has a good integration with Git version control that I also used in the development process. It also provides a convenient way for debugging because it integrates with The GNU Project Debugger[12].

I considered two most widely used compilers. The first one is the original native ‘‘`go tool compile`’’ also known as ‘‘`gc`’’ and the second one is ‘‘`gccgo`’’[13]. The main difference is that gccgo compiles slower but can increase performance for some specific CPU architectures. In this project, the more important factor for the development is fast compilation time because after every change, the source code has to be recompiled. This allowed faster prototyping of smart contracts.

4.2.4 Source Control

As a source control, I decided to use Git[14] version control because it is stable an widely used in the industry. Moreover, I am familiar with Git as a daily user. For the development of the smart contracts and bash scripts, a single master branch was set up. After every significant change, the repository was backed up to a remote location on Github.com.

4.2.5 Containers

For better scalability and encapsulation Hyperledger Fabric uses Docker[15] containers. Every component such as a node, orderer and even smart contract is running in a separate environment inside a Docker container. If there is a need to use more nodes or orderers to scale up, we can just run new instances. It is a very simple but powerful concept that is widely used in the industry.

4.2.6 Storage for The Ledger

There were two options that could be used as a storage for the ledger. The first one was CouchDB[16] and the second was LevelDB[17]. CouchDB allows rich query against the data when values are modelled data, as JSON data. Even though it seemed to be a better choice, during the evaluation I found out that it posed a limitation for transaction throughput in my set up. Simply, some

[9] https://nodejs.org/en/
[10] https://www.gnu.org/software/bash/
[11] https://code.visualstudio.com/
[12] https://www.gnu.org/software/gdb/
[13] https://blog.golang.org/gccgo-in-gcc-471
[14] https://git-scm.com/
[15] https://www.docker.com/
[16] https://couchdb.apache.org/
[17] https://github.com/google/leveldb

transactions were not committed to the new state. Therefore, I had to switch to LevelDB that turned out to be better suited for the developed system.

4.3 Implementation

This section describes the implementation. At the beginning I started to implement the most important smart contract for storing and retrieving data from the ledger. The second smart contract that was developed provide basic functionality for accounts and payments between parties. This is necessary only if the data exchange involves payment. Having finished these two smart contracts, I could continue with the third one that interacts with both of them. The third smart contract provides functionality for publishing an advertisement for data entry as described in Figure 3.1. These three smart contracts provide an API for the whole system.

4.3.1 Throughput-Related Issues

After implementing all three smart contracts I performed an initial evaluation and encountered major drawbacks of the implementation. Therefore, I decided to completely rewrite all three smart contracts so that they are now able to handle the high throughput. The major change was in the way the key-value pair in the ledger is updated. The first implementation[18] used a single key-value pair in the ledger for a single asset. This approach is perfectly fine for an asset that is not updated tens of times per minute. However, when sensors produce new data tens of times per second, this causes major issues. In the time between when the transaction is simulated on the node and it is ready to be committed to the ledger, another transaction could have already updated the same value. Therefore, the read-set version will no longer match the version in the orderer. For this reason, a large number of parallel transactions will fail and the value will not be updated. Moreover, this issue becomes even more evident in the case of a single account reacquiring an update after every payment for data collection. This could require thousands of transactions per second.

The solution is that a frequently updated value must be stored as a series of deltas (differences of the value). When the most recent value is required, all the series of deltas are aggregated and the result returned as a single value. For example, an account has the following transactions +100, -10, +5. These transactions would be stored as separate key-value pairs in the ledger. To retrieve the current state of the account the smart contract aggregates all the deltas like this (+100-10+5)=95. Hence, the current value of the account is 95 and the order or the transactions does not matter. This approach solves the problem with high throughput. On the other hand, the solution comes at a cost. If the smart contract does not check the most recent value of the account, it can be overdrawn. The smart contract cannot do it because the read-set will change as described above and there is the same issue again. Nonetheless, there are several countermeasures to mitigate this issue. I explain how I designed and implemented them for each of the smart contracts in the following sections.

4.3.2 Smart Contract for Data Entry

The main purposes of this smart contract are straightforward. There have to be functions that create data and retrieve data from the state of the ledger dedicated for this purpose. As the first

[18] This is possible with simple command ‘‘`$ git checkout 8cb9365c5e4070568fbb6112c6c5eb1b80ab655b`’’ and to get back to latest commit use command ‘‘`$ git checkout master`’’

thing, I had to design a structure for the data that should be stored in the ledger. This data structure is shown in Figure 4.1 using Go language.

Figure 4.1: Data structure of type DataEntry in Go language.

```
type DataEntry struct {
    RecordType   string // RecordType is used to distinguish the various
                        // types of objects in state database
    DataEntryID  string // unique compound key that is ID~CreationTime
    Description  string // human readable description
    Value        string // data value
    Unit         string // optional units for the data value
    CreationTime uint64 // Time when the data was created.
                        // It can differ from the blockchain entry time
    Publisher    string // publisher of the data
}
```

Every variable except `CreationTime` is string data type. This was done deliberately to avoid any possible limitations. For example, `DataEntryID` could be a hash string or Value could be in hexadecimal etc. On the other hand, `CreationTime` is unsigned integer for a specific reason. I wanted to provide a function that can return the latest entry and this is done using comparison operator on `CreationTime` member between two data entries.

For this smart contract, I solved the issue with throughput as follows. When a sensor produces new data, instead of an update of an existing key-value pair, the smart contract creates new `DataEntry` with a unique `DataEntryID` that is a compound key. It means that `DataEntryID` is a combination of unique ID of a sensor and `CreationTime`. This guarantees uniqeness of the key, but allows retireval based on sole partial key identifier (sensor ID). Moreover, the smart contract checks for a collision. In the case of two identical `DataEntryID`, the smart contract fails to update the ledger and returns an adequate error message.

Functions that can be invoked in this smart contract are:

1. `createData` – creates new `DataEntry` based on input arguments and save it to the ledger.
2. `getDataByIDAndTime` – returns single `DataEntry` based on sensor ID and `CreationTime`.
3. `getAllDataByID` – returns all `DataEntry` objects, based on sensor ID, which are stored in the ledger.
4. `getLatestDataByID` – returns single `DataEntry`, based on the sensor ID, that has the latest `CreationTime`.
5. `getDataByPub` – useful when a single publisher application publishes `DataEntry` from multiple sensors. It is possible to query the ledger based on the specific publisher.

The concept of a compound key is used in all implemented smart contracts. It provides the possibility to index data for fast lookup in the ledger. For example, the function `createData` creates an additional entry in the ledger for the sole purpose of fast lookup for function `getDataByPub`. `createData` creates a compound key with value `0x00` that has three parts.

The first is `Publisher`, the second is `DataEntryID` and the third is `CreationTime`. The function `getDataByPub` first queries the ledger to retrieve a set of keys that match first parts of the key that is `Publisher`. Then it has all the `DataEntryID` that the `Publisher` produced.

This smart contract with a single ledger and a channel provides functionality for data exchange between two parties where cryptocurrency is not required. With the basic functionality provided by an API, it is possible to build complex applications with extended functionalities. Afterwards, I continued and developed another smart contract that provides cryptocurrency and all the related functionality.

4.3.3 Smart Contract for Tokens

The purpose of this smart contract is to provide cryptocurrency, accounts and standard functionality related to them. First of all, I designed a data structure that represents an account. See Figure 4.2. This structure `Account` is simple and easily expandable with new members. It is important to note that member `Tokens` represents the cryptocurrency and it is an `int64` data type. The reason why it is an `int64` and not `float64` is the difficulty with floating point arithmetic[19]. This is a simple and efficient solution without the need for a new data structure that would store the decimal places as an `int` data type. Furthermore, I decided to create all tokens at once during the first initialisation. They are assigned to a single account and then they can be distributed to the economics all at once or continuously to deploy an inflationary policy. The number of initial tokens can be changed during the initialisation process.

Figure 4.2: Data structure of type Account in Go language.

```
// Account represents account of a member
type Account struct {
RecordType string // RecordType is used to distinguish the various types
                      //of objects in state database
AccountID  string // unique id of the account
Name       string // name of the account (holder)
OwnerID    string // Cryptographic account holder identity
Tokens     int64  // amount of tokens (money)
}
```

This smart contract provides the following functions as a callable API:

1. `createAccount` – creates an account with zero tokens.
2. `deleteAccountByID` – deletes an `Account` based on provided `AccountID` only when an account have zero tokens. This preserves the initial number of tokens.
3. `getAccountByID` – returns an object `Account` based on provided `AccountID`.
4. `getAccountByName` – returns all accounts that have the provided `Name`.
5. `sendTokensFast` – transfers tokens from one account to another and returns the transaction ID. Further explained in the following paragraphs.

[19] https://en.wikipedia.org/wiki/Floating-point_arithmetic

6. `sendTokensSafe` – transfers tokens from one account to another and returns the transaction ID. Further explained in the following paragraphs.

7. `updateAccountTokens` – updates an `Account Tokens` member based on returned value from a function `getAccountTokens`

8. `getAccountTokens` – aggregates all deltas and returns the most recent amount of tokens that the `Account` has.

9. `getAccountHistoryByID` – returns the complete history of a specific `Account` even if the `Account` was deleted.

10. `getTxDetails` – returns participants' `AccountID`s of the transaction, the amount of tokens transfered and the state of the transaction. State of the transaction can be `PendingTx` meaning that the tokens are on hold and the recipient cannot use them yet. Or it can be `ValidTx` which means that the transaction is finished and the tokens are released to the recipient.

11. `changePendingTx` – changes a `PendingTx` to a `ValidTx` and returns the new transaction ID. Further explained in the following paragraphs.

12. `pruneAccountTx` – Prune all transaction for a specific `AccountID` and aggregates them to a single one. It returns the new transaction ID. Further explained in the following paragraphs.

For this smart contract, I solved the issue with throughput as it was described in Subsection 4.3.1. Moreover, I decided to implement two functions for sending tokens. The first is `sendTokensFast`. Every change of the amount of `Tokens` is stored as a sequence of deltas. In order to mitigate the problem that an `Account` can be overdrawn, there is a fixed limit for the number of tokens that can be sent via this function. By default, it is only one token, but it can be easily adjusted for an individual needs. For example, the limit could be increased based on solvency of an account holder. Separately from these transactions, there could be a specialised dedicated node that regularly pools the aggregate value and update accounts. This countermeasure helps when an account overdraw occurs, it can be detected within a specific amount of time. For this purpose there is a function `updateAccountTokens`. In the case that an account has thousands of transactions with deltas the value aggregation slows down. To solve this problem I wrote a function `pruneAccountTx` that cleans up the ledger and prune all existing transactions of an account into a single one. Again this can be regularly done by the dedicated node that updates accounts.

The second function is `sendTokensSafe`. This function does not pose any limit on the number of tokens that should be transferred. It calls a function `getAccountTokens` and verifies if the account has enough tokens. Therefore, by using this function there cannot occur an account overdraw. On the other hand, this function is not suitable for the concurrent execution of transactions on the same account because it creates the read-set. This read-set can change as a result of a commit of a concurrent transaction. This was described in Subsection 4.3.1. Therefore, the function can fail to update the ledger and it has to be called again. However, it is a safe way how to send tokens from one account to another.

Additionally, `sendTokensSafe` and `sendTokensFast` functions provide an option that a transaction transferring tokens from one account to another is indexed as payment for data. Thereafter, such tokens cannot be immediately used by the recipient but have to be released after the data retrieval via function `changePendingTx`. This function has a built-in protection mechanism that first verifies via inter-channel communication querying the smart contract for data entry advertisement if the `DataEntry Value` was indeed revealed. It provides the assurance for trading parties that in the case that advertised data does not exist the buyer is protected.

4.3.4 Smart Contract for Data Entry Advertisement

The purpose of this smart contract is to provide a place where the producer of data can advertise the data for sale. On the other side there is the consumer of data that should be able to look up and select specific data that wants to purchase. For this purpose I designed and implemented a data structure that is shown in Figure 4.3.

Figure 4.3: Data structure of type DataEntryAd in Go language.

```
type DataEntryAd struct {
    DataEntry        // anonymous field
    Price     int64  // Price for data value
    AccountNo string // account number where to transfer tokens
}
```

This data structure extends `DataEntry`. The first additonal member is `Price` for the `DataEntry`. The second member is `AccountNo` that informs the potential buyer where to send tokens.

The smart contract provides the following functions as a callable API:

1. `createDataEntryAd` – creates new `DataEntryAd` based on input arguments and saves it to the ledger.

2. `getDataAdByIDAndTime` – returns single `DataEntryAd` object based on sensor ID and `CreationTime`.

3. `getAllDataAdByID` – returns all `DataEntryAd` objects, based on sensor ID, which are stored in the ledger.

4. `getLatestDataAdByID` – returns single `DataEntryAd`, based on the sensor ID, that has the latest `CreationTime`.

5. `getDataAdByPub` – returns all `DataEntryAd` objects, based on the specific publisher.

6. `revealPaidData` – reveals a specific data value for a chosen `DataEntryAd`. Further explained in the following paragraphs.

7. `checkTXState` – checks if a transaction ID is already `Used` for data purchase or `Unused`. Further explained in the following paragraphs.

This smart contract puts all together and it cannot function properly without those two smart contracts described earlier. The function `revealPaidData` reveals a specific data value for a chosen `DataEntryAd`. As an input it also requires a transaction ID (is unique) that was returned either from the function `sendTokensFast` or `sendTokensSafe`. Moreover, this transaction must be marked as payment for the data purchase. The function `revealPaidData` has a built-in protection mechanism that firstly verifies (via inter-channel communication querying the smart contract for tokens) if the recipient `AccountID` matches the `AccountID` in `DataEntryAd`. In addition, it checks if the transaction ID has still status as `PendingTx` and if the `Price` matches with the amount of transferred `Tokens`. In order to decrease the inter-channel communication, the function before querying another smart contract checks its own index in the ledger by calling a function `checkTXState`. If it is `Unused` then it continues.

It is possible to imagine that the transaction ID serves the purpose of a unique ticket that can be used for data purchase only one time and then the transferred tokens can be released. Since the function execution is an atomic operation, it cannot happen that the transaction ID would be indexed as an `Used` one and data would not be revealed. The atomicity guarantees that either all steps in the function are executed and then the ledger is updated or nothing is updated in the ledger. Furthermore, the function invocation can be limited by a policy which explicitly states that in order to invoke functions in this smart contract, the application has to collect signed proposals by both (can be more than two or an exact number of) parties. If In simple words, data value can be revealed only if both parties agree on the result of the function execution. Afterwords, the ledger can be updated and the `Value` revealed. For this smart contract, I solved the issue with throughput the same way as it was described in the Subsection 4.3.2.

4.4 Summary

This chapter discusses methodology, technologies and implementation. The methodology that I decided to adopt is an exploratory programming approach, whereby all the steps taken were described. Among many DLTs, I decided to use Hyperledger Fabric v1.0. As a programming language for the smart contracts, I used Go language. GNU Bash scripts are used for interfacing with a node and an orderer. All of the code base was written in Microsoft Visual Studio Code. As a version control, I used Git and every component is containerised with Docker. As a storage for the ledger I firstly used CouchDB, but later I decided to swap it for LevelDB that turned out to be more suitable. All of the mentioned technologies and tools are free. Therefore, they completely fulfil **NFR4**.

The Implementation Section describes the high-level implementation and all the functionality of separate smart contracts provide. In addition, it explains the specific decision made about how to update the ledger for allowing high throughput of the whole system.

Chapter 5

Testing & Evaluation

This chapter describes tests conducted on the system. It also describes the architecture, tools, and process used for the evaluation of the developed system. Further, all the results are presented, statistically analysed, explained and discussed.

5.1 Testing

The system was developed within a month of continuous writing of new code, improving existing and testing both. Since I used exploratory programming approach, I needed very often to try new ideas and test them. The testing of smart contracts turned out to be very time-consuming. The reason is that the smart contracts have to be executed and tested on a complete network containing all the components described in Section 3.3. For this reason, I wrote a GNU Bash script that automated the complete set up of a fully functioning network. This network contains four nodes (two for each party), one orderer, publisher and consumer of data. It automates installing smart contracts, channels, membership service policy and cryptographic suites necessary for each component. With this script, I started rapid prototyping phase.

There were dozens of prototypes that turned out to be a dead end. During this stage, the main approach for testing was using diagnostic print out messages that reported intermediate values to the console. This is considered a white-box testing of an internal structure of the functions. As the first functions started to emerge, I extended the GNU Bash script for basic tests that every time it created the network, it also tested the existing functions as a black-box. This provided me with a quick feedback in the case that I accidentally broke the existing functionality by introducing a new one or changing the existing one. In such a scenario I used the diagnostic print out messages to debug the code.

At the end, when the smart contracts were finalised, I wrote more than two hundreds unit tests. They thoroughly test almost every function of all three smart contracts mentioned in Section 4.3. Each function is tested for boundary values, unexpected inputs and expected inputs. The output of the tested function is automatically evaluated against the expected output. With the help of these unit tests, I patched dozens of minor bugs in the code. The coverage of these unit tests is following.

Output of the command "`go test -cover`" for the smart contracts:

- `chaincode_data.go` – 84.6% (Smart Contract for Data Entry)
- `chaincode_tokens.go` – 71.2% (Smart Contract for Tokens)

- `chaincode_ad.go` – 66.3% (Smart Contract for Data Entry Advertisement)

The coverage is not 100% because some branches of the code cannot be tested. For example, an imported function from a library returns two values. The first one is a pointer and the second is an error type. Therefore, there is a single `if err != nil` statement that ensures the value of the error variable does not indicate a failure. This branch of the code is not executed during unit testing and cannot be tested for such a case because the input to the tested function is sanitised. However, this does not decrease the quality of unit tests. Firstly, they are designed to test a function as a black-box. Secondly, some functions cannot be tested with unit tests, but they are tested with GNU Bash scripts that I developed. They simulate complex scenarios such as data production, publishing, advertising, retrieval, tokens transfer and data purchase. Therefore, I have to state that all of the developed code is fully and thoroughly tested. The coverage and richness of tests highly exceeds standards employed in typical commercial software engineering projects.

Tables 5.1, 5.2 and 5.3 show how specific functions are tested and the results of tests.

Table 5.1: Table of tests for all functions in the smart contract for data entry

Function Name	Test type	State
`Init`	Unit Tests & Scenario Script	Passes
`createData`	Unit Tests & Scenario Script	Passes
`getDataByIDAndTime`	Unit Tests & Scenario Script	Passes
`getAllDataByID`	Unit Tests	Passes
`getLatestDataByID`	Unit Tests	Passes
`getDataByPub`	Unit Tests	Passes

Table 5.2: Table of tests for all functions in the smart contract for tokens

Function Name	Test type	State
`Init`	Unit Tests & Scenario Script	Passes
`createAccount`	Unit Tests & Scenario Script	Passes
`deleteAccountByID`	Unit Tests	Passes
`getAccountByID`	Unit Tests & Scenario Script	Passes
`getAccountByName`	Unit Tests	Passes
`sendTokensFast`	Unit Tests & Scenario Script	Passes
`sendTokensSafe`	Unit Tests & Scenario Script	Passes
`updateAccountTokens`	Unit Tests & Scenario Script	Passes
`getAccountTokens`	Unit Tests & Scenario Script	Passes
`getAccountHistoryByID`	Unit Tests	Passes
`getTxDetails`	Unit Tests & Scenario Script	Passes
`changePendingTx`	Unit Tests & Scenario Script	Passes
`pruneAccountTx`	Unit Tests	Passes

Table 5.3: Table of tests for all functions in the smart contract for data entry advertisement

Function Name	Test type	State
`Init`	Unit Tests & Scenario Script	Passes
`createDataEntryAd`	Unit Tests & Scenario Script	Passes
`getDataAdByIDAndTime`	Unit Tests & Scenario Script	Passes
`getAllDataAdByID`	Unit Tests	Passes
`getLatestDataAdByID`	Unit Tests	Passes
`getDataAdByPub`	Unit Tests	Passes
`revealPaidData`	Unit Tests & Scenario Script	Passes
`checkTXState`	Unit Tests & Scenario Script	Passes

5.2 Evaluation

5.2.1 Game Theoretical Analysis

As mentioned previously, after the first implementation of all three smart contracts I conducted the initial evaluation. This showed that the smart contracts function as intended. The main idea and architecture design proved to be correct. They worked with serial updates of the same key-value pair in the ledger. However, when multiple processes tried to update the key-value pair the majority of these transactions were never committed to the ledger. This issue is described in detail in Section 4.3.1. After I completely rewrote all three smart contracts so that they did not experience this issue, I conducted the following evaluation.

First of all, I had to evaluate if the system fulfils all the functional and non-functional requirements. Since they were fulfilled by the architecture, and the implementation followed the architecture, this implies that all the requirements must be fulfilled.

It is important to critically evaluate and explain the limitations of the architecture and implementation. There are two critical sections in the data exchange procedure between two trading parties that have to be analysed.

1. In Section 3.3.8 Point 15 sates "C contacts N (Party 1), with valid token transaction ID, to generate a transaction proposal response for L2 on CH2 to reveal data entry value for the advertisement." At this point, the Party 1 can refuse to sign the proposal and simply not reveal the Data value. Considering only two trading parties, it is important to say that nobody can access the sent tokens. This situation can be further studied with the help of economic game theory[39]. As Table 5.4 shows, Party 1 has an incentive to reveal data and does not have any incentives to misbehave.

2. In Section 3.3.8 Point 23 sates "P also contacts N′ (Party 2) using SC3 to generate a transaction proposal response for L3 on CH3 to release the tokens used for the revealed data entry value. C must have valid, signed endorsement from both N (Party 1) and N' (Party 2)". Considering only two trading parties, Party 2 can refuse to sign the proposal and effectively keep blocking the release of tokens. At this point, Party 1 revealed the Data to Party 2 and Party 1 paid for it. However, this is a feature of this implementation because if Party 1 decided that the revealed Data value is not what it was advertised it can keep the sent tokens on hold. Importantly, neither of the parties can access the tokens until they come to an agreement. This situation is further studied with the help of economic game theory. Table

5.5 shows that Party 2 has an incentive to release Tokens and does not have any incentives to misbehave.

Table 5.4: Payoff Table for Point 15 in Section 3.3.8

	Party 1 reveals Data	**Party1 does not reveal Data**
Party 2 waits	Party 2 \ Party 1 data \ promise of tokens	Party 2 \ Party 1 no data \ no promise of tokens

Table 5.5: Payoff Table for Point 23 in Section 3.3.8

	Party 2 releases Tokens	**Party 2 does not release Tokens**
Party 1 waits	Party 1 \ Party 2 tokens available \ future trade available	Party 1 \ Party 2 no tokens \ no future trade

On the other hand, people sometimes do not behave rationally. Therefore, there can be a dispute between two trading parties. In such situation, a third judging party can be invited to solve the dispute for some fee. Furthermore, considering more than two parties that participate in the ledger for Tokens, there can be a policy that says explicitly how many (for example at least 51% or 2/3) participants have to agree to update the ledger. All of this is supported by the developed system due to the architecture and technologies that I decided to use.

The next thing that is emphasised in this report is physical data locality awareness. In this case, the chosen technologies do not provide any direct support. However, Hyperledger Fabric uses Gossip data dissemination protocol[1] that was already mentioned in Section 4.2. This can be cleverly used for physical data locality awareness. Gossip protocol has a feature called "leader election" where a set of leader nodes within an organization maintain the connection with ordering service and initiate distribution of new blocks across nodes of their own organization. Therefore, the set of nodes within one organisation can be used to distinguish between separate physical locations.

5.2.2 Statistical Performance Analysis

The evaluation was conducted on fully functioning network that could be used immediately in a real-world scenario. This network consisted of two parties running two nodes. In addition, there a was single orderer node, publisher and subscriber of data. All these components were running in inside Docker[2] containers on a single computer. There were three channels, ledgers and smart contracts as described in Architecture Section 3.3. It is worth noting that the smart contracts are executed in separate Docker containers too.

Table 5.6 shows how much memory separate containers require during *idle* and *load*. These values are in mebibytes[3]. *peer1*s require significantly less memory than *peer0*s because *peer0*s have installed the smart contracts. *Peer1*s only maintain the ledger but do not execute any smart contract. These values server only for illustrative purpose. They were measured only single time.

Specification of the hardware and software used during measurements: HP EliteBook 8470p, CPU: Intel Core i7-3520M CPU locked at 2893.47 MHz, Memory: Micron 8GB

[1] https://hyperledger-fabric.readthedocs.io/en/release-1.1/gossip.html
[2] https://www.docker.com/
[3] https://en.wikipedia.org/wiki/Binary_prefix

Table 5.6: Memory usage of the system measured with Docker statistics.

Container Name	Idle memory usage	Load memory usage
peer0.city2.zak.codes-chaincode_tokens	3.418MiB	6.086MiB
peer0.city1.zak.codes-chaincode_tokens	3.898MiB	6.137MiB
peer0.city1.zak.codes-chaincode_ad	3.664MiB	6.07MiB
peer0.city1.zak.codes-chaincode_data	3.512MiB	6.04MiB
orderer.zak.codes	13.51MiB	15.51MiB
peer0.city1.zak.codes	71.86MiB	72.68MiB
peer0.city2.zak.codes	67.12MiB	67.93MiB
peer1.city1.zak.codes	30.91MiB	35.56MiB
peer1.city2.zak.codes	29.71MiB	34.43MiB

2x4GB DDR3 1600 MHz, OS Kernel: 4.15.15-300.fc27.x86_64, VM Swappiness: 0, Go version and compiler: go version go1.10.1 linux/amd64, Hyperledger Fabric v1.0: commit da14b6bae4a843dfb3fcece5a08ae0ea18488a7a

In the performance evaluation, I focused more on smart contracts rather than the whole system for the following reasons.

1. The real world performance of the whole system highly depends on many external factors and variables that I could not simulate or take into account on a single computer. For example, network topology, the number of nodes and orderers in the network, used consensus protocol, network performance, network technology (WiFi, Gigabit Ethernet, Fibre connection, etc.).

2. There is a constant time that specifies how often a new block should be created (Batch Time). This can be adjusted by the owner of the network and tuned for a specific situation. In my case, I decided to use the default value "2s"

Hence, the features that I can reliably measure and show how they perform are the most frequently used functions in smart contracts. They are `createData`, `createDataEntryAd` and `sendTokensFast`. For measuring the execution time I used Linux command `time`[4]. The output of this command was appended to a file and further processed with a Python 3 script (extract_usr_plus_sys_time.py). This script sums *User* plus *Sys* times and saves them into a new file that has a suitable input format for RStudio[5] which I used for the statistical analysis. *User* plus *Sys* time represent only the CPU time that was dedicated to the measured process. Even if the process is preempted by scheduler, this time only influence the *Real* but not *User* and *Sys* times.

The execution time of functions `createData` and `createDataEntryAd` was measured on sample of 1 000 transactions with different sizes of the message payload. Table 5.7 and 5.8 show the different message payload sizes, minimum, maximum, mean and standard deviation of execution time.

It is important to explain why the minimum message size differ. The reason is that every functions have a different numbers of input parameters and this defines the minimum message

[4] https://linux.die.net/man/1/time
[5] https://www.rstudio.com/products/rstudio/

Table 5.7: `createData` execution time with different size of a message payload. Calculated from sample of 1 000 transactions.

Message Size in Bytes	MIN	MAX	MEAN	STDEV
53	85.4ms	90.8ms	87.464ms	0.9399678ms
1024	85.6ms	90.1ms	87.867ms	0.9649467ms
8192	90.9ms	97.1ms	93.284ms	1.074283ms
16384	94.3ms	101.7ms	98.135ms	1.598129ms
32768	100.9ms	109.9ms	104.543ms	1.718953ms
65536	108.9ms	120.2ms	115.178ms	2.376041ms

Table 5.8: `createDataEntryAd` execution time execution time with different size of a message payload. Calculated from sample of 1 000 transactions.

Message Size in Bytes	MIN	MAX	MEAN	STDEV
73	92.5ms	102.1ms	96.211ms	1.556277ms
1024	92.3ms	101.7ms	97.271ms	1.835029ms
8192	94.8ms	103.7ms	99.259ms	1.80152ms
16384	97.7ms	112.3ms	104.776ms	2.654713ms
32768	105.7ms	128.7ms	116.391ms	4.956703ms
65536	112ms	128ms	119.809ms	3.635879ms

size. In addition, Figures 5.1 and 5.2 show mean values of execution time for the different message payload sizes. Based on this, it is possible to see that the increase in execution time, in the measured range, grows linearly for both functions `createData` and `createDataEntryAd`.

Functions `sendTokensFast` was evaluated on sample of 10 000 transactions. The message payload can vary only with the range of tens of byte. Therefore, it does not dramatically influence the execution time. Table 5.9 shows minimum, maximum, mean and standard deviation of the measurements.

Table 5.9: `sendTokensFast` execution time. Calculated from 10 000 samples

Transaction	MIN	MAX	MEAN	STDEV
sendTokensFast	91.7ms	101.7ms	95.2ms	1.779854ms

These results look interesting. I intended to compare them to some other system that provides similar functionality for data exchange with integrated blockchain technology. However, as mentioned in Section 2.3.5, to my best knowledge, there is no such a system implemented. All of the research papers only proposed some solutions but never actually implemented any of them. For this reason, this developed system establishes the baseline for all the future research related to this topic.

5.3 Summary

This chapter showed how the system was tested during the whole time of development. Furthermore, it also explained the architecture, hardware, software, tools and process used for the evaluation. Then all the measured data are presented, explained and discussed.

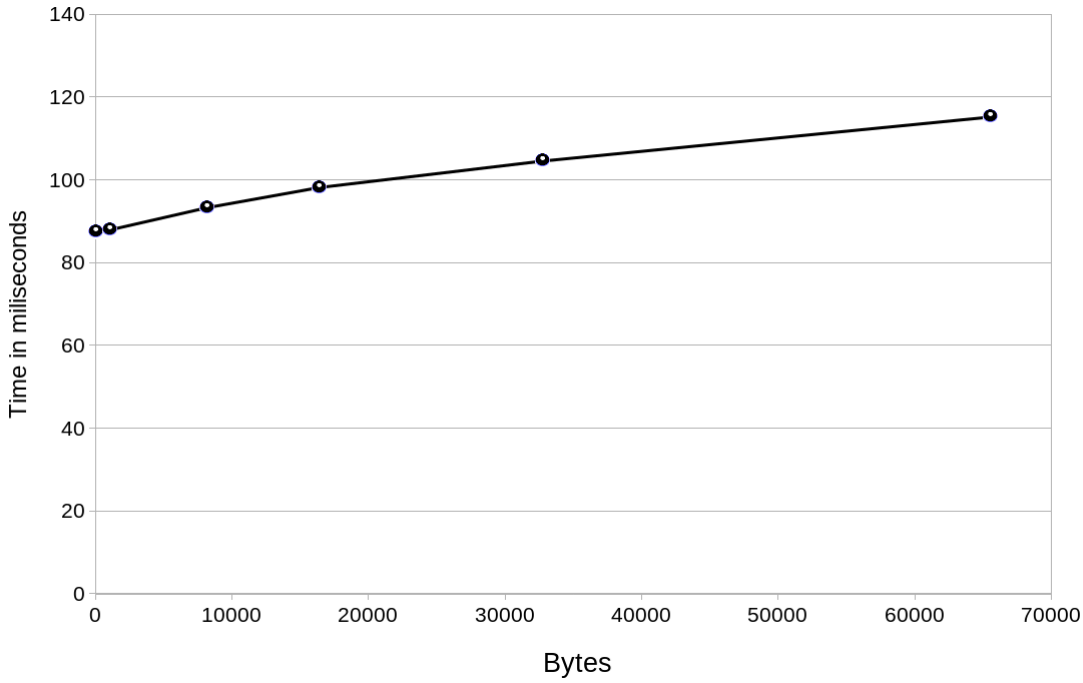

Figure 5.1: `createData` mean execution time with different size of a message payload. Calculated from sample of 1 000 transactions.

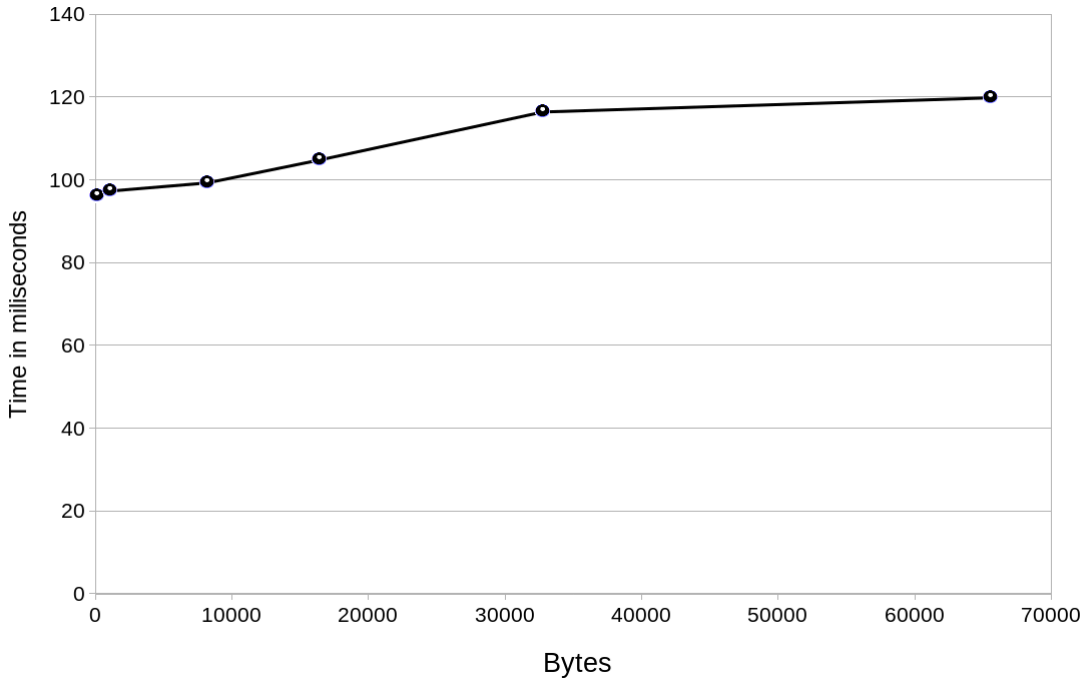

Figure 5.2: `createDataEntryAd` mean execution time execution time with different size of a message payload. Calculated from sample of 1 000 transactions.

Chapter 6

Conclusions, Discussion & Future Work

This chapter concludes the work of this project and discusses the developed system. In addition, it suggests the possible future work, improvements and tests that can be done.

6.1 Conclusions

Overall, the system fulfils all the functional and non-functional requirements specified in Chapter 3. In addition, there is a possibility to enhance this system for physical data locality awareness with existing features. The most important finding of this project is the ability to exchange paid or unpaid data between two parties without the need of third trusted party. This is enabled by the data exchange mechanism I developed. It is described in details in Section 3.3.8. Moreover, this data exchange mechanism was implemented and shown its functionality in real-world scenarios. However, it was not possible to compare the system to any other solution because they do not exist.

There is an evidence that the system is deployable in real-world settings. It is scalable, secure, accountable and robust. On the other hand, Table 5.8 shows that a single transaction execution can take approximately up to 128ms and smaller payloads have mean values about 97ms. These values are high due to the complexity of the data exchange protocol and multiple ledgers used. Therefore, the system would not be suitable for time-critical or real-time applications. I will go even further and claim that any system that incorporates (variations of) the blockchain mechanism would not be suitable for such applications. An additional reason for this claim is that we also have to take into account the time required for creation and distribution of the new block. This is measured in seconds. So the data propagation from `Producer` to `Consumer` can take about 2 seconds. However, it can be misleading to think that single data value transfer would take that amount of time. Due to the improved implementation that solved the issues with throughput, it is possible to transfer thousands of data values during this time period at once. Though, the introduced time delay because of the creation and distribution of a new block is still going to be there. For such time-critical application, there are better P2P communication protocols. One example can be DDS[1] protocol.

Nonetheless, the system was designed from the beginning for other purposes and it completely fulfils them. Therefore, I can conclude that the project was successful. I would recommend using this system in situations where the exchanged data have value and it is worth to introduce the additional overhead (time, resources, etc.) in comparison to less demanding data exchange

[1] http://www.eprosima.com/

protocols.

6.2 Discussion

From the beginning, it was very ambitious and time-demanding project. There were several aspects that made this project difficult. For example, the number of new technologies and programming languages used. Moreover, the development required a broad range of knowledge and skills that I had to gain during this time period. Since I did not have any courses that would introduce me to this problematic, the project required a lot of time and effort to finish. Also, searching for good sources that would explain blockchain technology into technical details is very difficult. Majority of the search results are automatically related to Bitcoin or they are simply not enough technical.

On the other hand, I am glad that I decided to work on this project because I worked with cutting-edge blockchain technology that is changing the world. What is even more important, I gained in-depth understanding of possibilities and limitations of blockchain technology. Furthermore, I learned about P2P networks, how they work and their advantages and disadvantages in comparison to traditional client-server model. I also improved my programming skills and added a new language to my skill set.

As my reflection on the development process, I have to say that I spent a substantial amount of time with reading existing research papers. This turned out to be very important because I did not repeat the same errors. I could stand on the shoulders of other researchers and continue where they stopped. During the development phase, I experienced difficulties with the first implementation, but I was able to overcome the limitations even though it meant rewriting more than 2/3 of the existing code. I've learned that sometimes it is necessary to explore a path even if it can result in failure because only then we can establish and evaluate the next possible steps in the development.

6.3 Future Work

I finish almost everything I wanted with one exception. Owner verification with the help of X.509[2] certificate. This should be finished before any real-world use. It has to be implemented in the smart contract for tokens for functions `sendTokensFast` and `sendTokensSafe`. There is already partial code which, since not finished, is commented out in the submission.

There is also a new stable version of Hyperledger Fabric v 1.1. This version brought new features, among which "Channel Private Date"[40] is particularly relevant to this project. It is only an experimental feature that provides private data on a ledger with multiple participants. This may reduce the 3 channels and 3 ledgers in the data exchange protocol to only two and increase the speed. Hence, it is necessary to research further this functionality.

Afterwards, it would be good to evaluate the performance of multiple devices connected together via multiple networking technologies connected together via multiple networking technologies. In addition, I think that there is still plenty of room for improvements, optimisation and additional functionality that could be added.

[2] https://en.wikipedia.org/wiki/X.509

Appendices

Appendix A

Background & Related Work Extras

A.1 P2P Routing Algorithms

As it was described previously, the number of connections that a node have with its peers (degree of a node) influences the routing performance and structure of the overlay network. Since networks can be modelled as graphs, they can be studied and evaluated with the help of graph theory [41]. The table A.1 and Table A.2 show the specific properties of widely used structured P2P overlay networks. Therefore, It is important to choose the right one based on specific criteria.

Table A.1: Asymptotic degree-diameter properties of the different graphs. (N is number of nodes in the graph)[41]

Graph	**Degree**	**Diameter D**
de Bruijin	k	$log_k N$
Trie	k+1	$2log_k N$
Chord	$log_2 N$	$log_2 N$
CAN	2d	1/2 dN 1/d
Pastry	(b-1) $log_b N$	$log_b N$
Classic butterfly	k	2 $log_k N(1-o(1))$
Note: Degree is a number of connection every node has.		
Note: Diameter D is max distance (hops) that a message must travel.		

Table A.2: Graph diameter for N = 10^6 (cells with a dash indicates that the graph does not support the corresponding node degree). [41]

k	**de Bruijin**	**Trie**	**Chord**	**CAN**	**Pastry**	**Classic butterfly**
2	20	-	-	huge	-	31
3	13	40	-	-	-	20
4	10	26	-	1,000	-	16
10	6	13	-	40	-	10
20	5	10	20	20	20	8
50	4	8	-	-	7	7
100	3	6	-	-	5	5

The next thing that has to be considered while choosing the graph structure is churn rate[1] on the P2P network. With every join and leave of a node, the routing table has to be updated. Usually, P2P networks such as Bittorrent[2] have high churn rate. Majority of nodes do not stay connected

[1] https://en.wikipedia.org/wiki/Churn_rate
[2] http://www.bittorrent.com/

for more than 1 hour [42]. This introduces higher maintenance cost than in P2P network where nodes have incentives to be connected for a longer time period. One example can be Skype in early days where the median lifetime of a node was 5.5 hours [43]. As we can see churn rate is an important factor that needs to be considered while choosing the graph structure of P2P overlay network.

There are many implementations of DHT that were developed. CAN [44], Chord [45], Pastry [46] and Tapestry [47] were the first one. Later, important improvements in terms of performance were made. CoralCDN[3] achieved the improvement through "a latency-optimized hierarchical indexing infrastructure based on a novel abstraction called a distributed sloppy hash table, or DSHT" [48]. DSHT helped also prevent hot-spot congestion (overloading a node when a specific key becomes very popular). Moreover, Coral maintained a hierarchy of DSHT clusters based on region and size and therefore, allowed "nodes to locate nearby cached copies of web objects without querying more distant nodes" [48]. Also, important security improvements in DHT were made with the introduction of S/Kademlia [49] that has high resilience against common attacks. It uses cryptographic puzzles in order to limit free NodeId generation. S/Kademlia also uses parallel lookups over multiple disjoint paths over the network. Initial evaluation has shown that even with 20% of adversarial nodes in the network, there is still 99% chance of a successful lookup [49]. More comprehensive discussion about DHT and structured P2P is above the scope of this work. More information can be found in this book [13].

A.2 PoS Nothing at Stake

Proof-of-Stake consensus algorithms have many advantages but in many implementations, including Peercoin, validators can be only rewarded for producing new blocks but not penalized. This is causing the problem of nothing at stake. In the situation where multiple chains are competing, the validator's incentive is to vote for every chain at the same time. Imagine a situation with two competing chains where the validator can vote on a chain A and get reward P = 0.9 or on a chain B where the reward is P = 0.1 or on both at once if possible. This is shown in Figure A.1. This results in a violation of safety and there is no incentive to converge into a single growing blockchain [50].

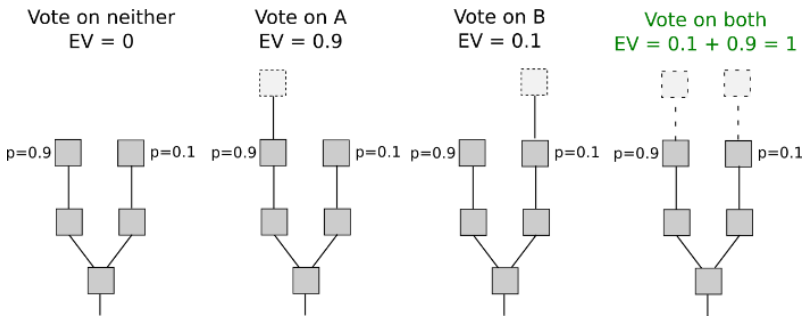

Figure A.1: Nothing at stake problem. [50].

In comparison to Proof-of-Work, doing so would require splitting one's computing power in half. Therefore, this approach is not lucrative. It is shown in Figure A.2. In this situation when the validator votes for both competing chains, the reward for voting on chain A and B is decreased by 50% and the combined reward is, therefore, smaller than voting only for original chain.

[3] http://www.coralcdn.org/

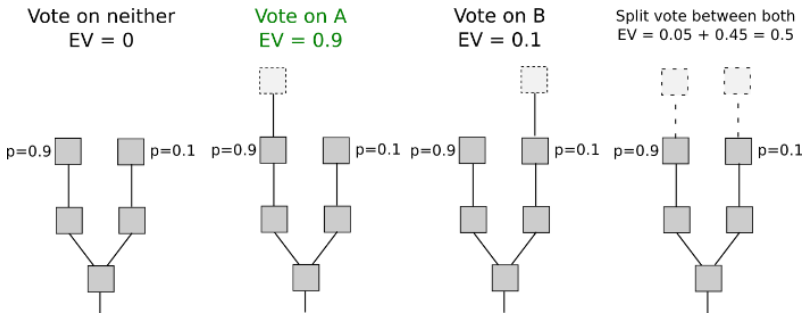

Figure A.2: Proof-of-Work, splitting one's computing power in half [50].

Nothing at stake can be prevented with a mechanism called *slasher* that was first proposed by Vitalik Buterin (Co-Founder of Ethereum) [51]. This mechanism penalizes validators that vote for multiple blocks simultaneously. When such situation happens then the validator's deposit is deduced appropriately. Figure A.3 shows how *slasher* works. This mechanism is not simple to implement and it is still under heavy development for Ethereum Casper [50].

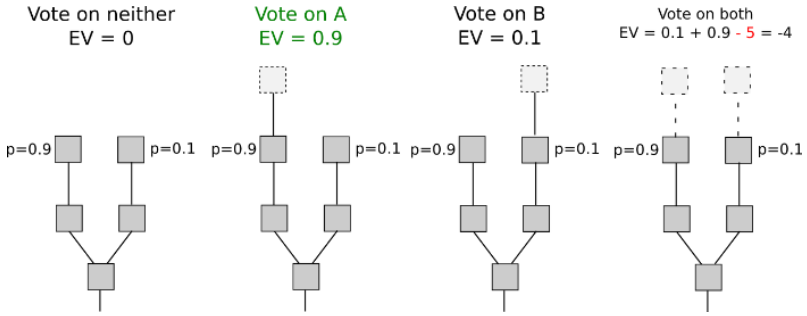

Figure A.3: Nothing at stake problem with slasher [50].

Some people argue that even without *slasher* mechanism validators (stakeholders) have an incentive to act correctly and only vote for the original chain and not attacker's. The reason is that it is in their best interest to preserve the value of their investment because by supporting multiple chains at once the value of tokens on the market would decrease. On the other hand, this idea ignores that this incentive suffer from tragedy of commons[4] problems [50].

[4] https://en.wikipedia.org/wiki/Tragedy_of_the_commons

Appendix B

User Manual

In order to use the developed system, it is necessary to create your own application that will call the exposed API. The following tables shows the required input parameters and output values for the API. As a part of of submission there are scenario GNU Bash scripts that serve as examples how to use the API.

Table B.1: Table of API functions for chaincode_data.go

Function Name	Input parameters	Output
`Init`	empty string	nil
`createData`	"DataEntryID", "Description", "Value", "Unit", "CreationTime", "Publisher"	nil
`getDataByIDAndTime`	"ID", "creationTime"	DataEntry
`getAllDataByID`	"ID"	JSON array of DataEntry
`getLatestDataByID`	"ID"	DataEntry
`getDataByPub`	"Publisher"	JSON array of DataEntry

Table B.2: Table of API functions for chaincode_tokens.go

Function Name	Test type	State
`Init`	"Initial amount of tokens"	Transaction ID
`createAccount`	"AccountID", "Name"	String "Account created"
`deleteAccountByID`	"AccountID"	String "Account deleted"
`getAccountByID`	"AccountID"	Account as byte array
`getAccountByName`	"Name"	JSON array of Account
`sendTokensFast`	"fromAccountId", "toAccountId", "Amount" "dataPurchase"	Transaction ID
`sendTokensSafe`	"fromAccountId", "toAccountId", "Amount" "dataPurchase"	Transaction ID
`updateAccountTokens`	"AccountID"	Account
`getAccountTokens`	"AccountID"	tokens
`getAccountHistoryByID`	"AccountID"	All transaction for AccountID
`getTxDetails`	"TransactionID"	details as string
`changePendingTx`	"channelAd", "chaincodeAdName" "TransactionID"	"TransactionID"
`pruneAccountTx`	"AccountID"	"TransactionID"

Table B.3: Table of API functions for chaincode_ad.go

Function Name	**Test type**	**State**
`Init`	empty string	nil
`createDataEntryAd`	""DataEntryID", "Description", "Value", "Unit", "CreationTime", "Publisher", "Price", "AccountNo"	DataEntryAd
`getDataAdByIDAndTime`	"ID", "creationTime"	DataEntryAd
`getAllDataAdByID`	"ID"	JSON array of DataEntryAd
`getLatestDataAdByID`	"ID"	DataEntryAd
`getDataAdByPub`	"Publisher"	JSON array of DataEntryAd
`revealPaidData`	"channelData", "chaincodeDataName", "dataEntryID", "creationTime", "channelTokens", "chaincodeTokensName", "txID"	DataEntryAd
`checkTXState`	"TransactionID"	("Used"|"Unused")

Appendix C

Maintenance Manual

In order to run the system follow these instructions:

1. Install Go. Follow this manual: https://golang.org/doc/install
2. Install Docker. Follow this manual: https://docs.docker.com/install/
3. Install Hyperledger Fabric v 1.0 to your home directory. Follow this manual: https://hyperledger-fabric.readthedocs.io/en/latest/prereqs.html Then follow this manual: https://hyperledger-fabric.readthedocs.io/en/latest/install.html
4. Open the root directory of the compressed file that contains the system.
5. Open directory network
6. Execute GNU Bash script 3channels_network_launcher.sh

This will completely set up the network with four nodes and single ordering service. To edit the number of nodes and orderers, edit the config files configtx.yaml and docker-compose-cli.yaml

Bibliography

[1] "World's population increasingly urban with more than half living in urban areas | UN DESA | United Nations Department of Economic and Social Affairs," Jul. 2014. [Online]. Available: https://www.un.org/development/desa/en/news/population/world-urbanization-prospects.html

[2] "WHO | 7 million premature deaths annually linked to air pollution," Mar. 2014. [Online]. Available: http://www.who.int/mediacentre/news/releases/2014/air-pollution/en/

[3] "Smart city - Wikipedia." [Online]. Available: https://en.wikipedia.org/wiki/Smart_city

[4] P. Labs, "IPFS is the Distributed Web." [Online]. Available: https://ipfs.io/

[5] S. Curtis, "How much is your personal data worth?" Nov. 2015. [Online]. Available: http://www.telegraph.co.uk/technology/news/12012191/How-much-is-your-personal-data-worth.html

[6] "Logitech Will Intentionally Brick All Harmony Link Devices Next Year." [Online]. Available: https://www.bleepingcomputer.com/news/hardware/logitech-will-intentionally-brick-all-harmony-link-devices-next-year/

[7] S. Duzellier, "Radiation effects on electronic devices in space," *Aerospace Science and Technology*, vol. 9, no. 1, pp. 93–99, Jan. 2005. [Online]. Available: http://linkinghub.elsevier.com/retrieve/pii/S1270963804001129

[8] Q. H. Vu, M. Lupu, and B. C. Ooi, *Peer-to-Peer Computing*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010. [Online]. Available: http://link.springer.com/10.1007/978-3-642-03514-2

[9] "Napster," Jan. 2018, page Version ID: 820643659. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Napster&oldid=820643659

[10] "BoincPapers âĂŞ BOINC." [Online]. Available: https://boinc.berkeley.edu/trac/wiki/BoincPapers

[11] "SETI@home." [Online]. Available: https://setiathome.ssl.berkeley.edu/

[12] J. F. Koegel Buford, H. H. Yu, and E. K. Lua, *P2P networking and applications*, ser. The Morgan Kaufmann series in networking. Amsterdam ; Boston: Elsevier/Morgan Kaufmann, 2009, oCLC: ocn267167232.

[13] D. Korzun and A. Gurtov, *Structured Peer-to-Peer Systems*. New York, NY: Springer New York, 2013. [Online]. Available: http://link.springer.com/10.1007/978-1-4614-5483-0

[14] X. Zhang, Q. Zhang, Z. Zhang, G. Song, and W. Zhu, "A Construction of Locality-Aware Overlay Network: mOverlay and Its Performance," *IEEE Journal on Selected Areas in Communications*, vol. 22, no. 1, pp. 18–28, Jan. 2004. [Online]. Available: http://ieeexplore.ieee.org/document/1258112/

[15] M. Ripeanu, I. Foster, and A. Iamnitchi, "Mapping the gnutella network: Properties of large-scale peer-to-peer systems and implications for system design," *arXiv preprint cs/0209028*, 2002.

[16] I. J. Taylor and A. B. Harrison, *From P2P and grids to services on the web: evolving distributed communities*, 2nd ed., ser. Computer communications and networks. London: Springer, 2009, oCLC: 254593378.

[17] A. Pinna and W. Ruttenberg, "Distributed Ledger Technologies in Securities Post-Trading Revolution or Evolution?" 2016.

[18] J. Mattila, *The blockchain phenomenon.* Berkeley Roundtable of the International Economy, 2016. [Online]. Available: https://www.researchgate.net/profile/Juri_Mattila/publication/313477689_The_Blockchain_Phenomenon_-_The_Disruptive_Potential_of_Distributed_Consensus_Architectures/links/589c31caa6fdcc754174493a/The-Blockchain-Phenomenon-The-Disruptive-Potential-of-Distributed-Consensus-Architectures.pdf

[19] S. Nakamoto, "Bitcoin: A Peer-to-Peer Electronic Cash System," 2008.

[20] S. Noether, "Ring Signature Confidential Transactions for Monero," Tech. Rep. 1098, 2015. [Online]. Available: http://eprint.iacr.org/2015/1098

[21] M. J. Fischer, N. A. Lynch, and M. S. Paterson, "Impossibility of distributed consensus with one faulty process," *Journal of the ACM (JACM)*, vol. 32, no. 2, pp. 374–382, 1985.

[22] E. Buchman, "Tendermint: Byzantine Fault Tolerance in the Age of Blockchains," PhD Thesis, 2016.

[23] M. Castro and B. Liskov, "Practical Byzantine fault tolerance," in *OSDI*, vol. 99, 1999, pp. 173–186.

[24] C. Cachin, "Architecture of the Hyperledger blockchain fabric," in *Workshop on Distributed Cryptocurrencies and Consensus Ledgers*, 2016.

[25] C. Dwork and M. Naor, "Pricing via processing or combatting junk mail," in *Annual International Cryptology Conference*. Springer, 1992, pp. 139–147.

[26] R. C. Merkle, "Protocols for Public Key Cryptosystems." IEEE, Apr. 1980, pp. 122–122. [Online]. Available: http://ieeexplore.ieee.org/document/6233691/

[27] S. King and S. Nadal, "Ppcoin: Peer-to-peer crypto-currency with proof-of-stake," *self-published paper, August*, vol. 19, 2012.

[28] N. Szabo, "Smart Contracts: Building Blocks for Digital Markets." [Online]. Available: http://www.fon.hum.uva.nl/rob/Courses/InformationInSpeech/CDROM/Literature/LOTwinterschool2006/szabo.best.vwh.net/smart_contracts_2.html

[29] "Introduction to Smart Contracts âĂŤ Solidity 0.4.21 documentation." [Online]. Available: https://solidity.readthedocs.io/en/develop/introduction-to-smart-contracts.html#overview

[30] M. VukoliÄĞ, "Rethinking Permissioned Blockchains." ACM Press, 2017, pp. 3–7. [Online]. Available: http://dl.acm.org/citation.cfm?doid=3055518.3055526

[31] K. Christidis and M. Devetsikiotis, "Blockchains and Smart Contracts for the Internet of Things," *IEEE Access*, vol. 4, pp. 2292–2303, 2016.

[32] K. R. ÃŰzyÄślmaz and A. Yurdakul, "Integrating low-power IoT devices to a blockchain-based infrastructure: work-in-progress." ACM Press, 2017, pp. 1–2. [Online]. Available:

http://dl.acm.org/citation.cfm?doid=3125503.3125628

[33] M. S. Ali, K. Dolui, and F. Antonelli, "IoT data privacy via blockchains and IPFS." ACM Press, 2017, pp. 1–7. [Online]. Available: http://dl.acm.org/citation.cfm?doid=3131542.3131563

[34] H. Shafagh, L. Burkhalter, A. Hithnawi, and S. Duquennoy, "Towards Blockchain-based Auditable Storage and Sharing of IoT Data." ACM Press, 2017, pp. 45–50. [Online]. Available: http://dl.acm.org/citation.cfm?doid=3140649.3140656

[35] S. Ibba, A. Pinna, M. Seu, and F. E. Pani, "CitySense: blockchain-oriented smart cities," in *Proceedings of the XP2017 Scientific Workshops*. ACM, 2017, p. 12.

[36] P. Missier, S. Bajoudah, A. Capossele, A. Gaglione, and M. Nati, "Mind my value: a decentralized infrastructure for fair and trusted IoT data trading." ACM Press, 2017, pp. 1–8. [Online]. Available: http://dl.acm.org/citation.cfm?doid=3131542.3131564

[37] A. Dorri, S. S. Kanhere, and R. Jurdak, "Towards an Optimized BlockChain for IoT." ACM Press, 2017, pp. 173–178. [Online]. Available: http://dl.acm.org/citation.cfm?doid=3054977.3055003

[38] R. Mall, *FUNDAMENTALS OF SOFTWARE ENGINEERING*. PHI Learning.

[39] M. J. Osborne and A. Rubinstein, *A Course In Game Theory*, 2nd ed. The MIT Press, 1995.

[40] "[FAB-1151] Side DB - Channel Private Data - experimental feature - Hyperledger JIRA." [Online]. Available: https://jira.hyperledger.org/browse/FAB-1151

[41] D. Loguinov, A. Kumar, V. Rai, and S. Ganesh, "Graph-theoretic analysis of structured peer-to-peer systems: routing distances and fault resilience," in *Proceedings of the 2003 conference on Applications, technologies, architectures, and protocols for computer communications*. ACM, 2003, pp. 395–406.

[42] D. Stutzbach and R. Rejaie, "Understanding churn in peer-to-peer networks," in *Proceedings of the 6th ACM SIGCOMM conference on Internet measurement*. ACM, 2006, pp. 189–202.

[43] S. Guha and N. Daswani, "An experimental study of the skype peer-to-peer voip system," Cornell University, Tech. Rep., 2005.

[44] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker, *A scalable content-addressable network*. ACM, 2001, vol. 31, no. 4.

[45] I. Stoica, R. Morris, D. Liben-Nowell, D. Karger, M. Kaashoek, F. Dabek, and H. Balakrishnan, "Chord: a scalable peer-to-peer lookup protocol for internet applications," *IEEE/ACM Transactions on Networking*, vol. 11, no. 1, pp. 17–32, Feb. 2003. [Online]. Available: http://ieeexplore.ieee.org/document/1180543/

[46] A. Rowstron and P. Druschel, "Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems," in *IFIP/ACM International Conference on Distributed Systems Platforms and Open Distributed Processing*. Springer, 2001, pp. 329–350.

[47] B. Y. Zhao, L. Huang, J. Stribling, S. C. Rhea, A. D. Joseph, and J. D. Kubiatowicz, "Tapestry: a resilient global-scale overlay for service deployment," *IEEE Journal on Selected Areas in Communications*, vol. 22, no. 1, pp. 41–53, Jan. 2004.

[48] M. J. Freedman, E. Freudenthal, and D. Mazieres, "Democratizing Content Publication with Coral." in *NSDI*, vol. 4, 2004, pp. 18–18.

[49] I. Baumgart and S. Mies, "S/kademlia: A practicable approach towards secure key-based

routing," in *Parallel and Distributed Systems, 2007 International Conference on*. IEEE, 2007, pp. 1–8.

[50] "Proof of Stake FAQ Âů ethereum/wiki Wiki." [Online]. Available: https://github.com/ethereum/wiki/wiki/Proof-of-Stake-FAQ

[51] V. Buterin, "Slasher: A Punitive Proof-of-Stake Algorithm," Jan. 2014. [Online]. Available: https://blog.ethereum.org/2014/01/15/slasher-a-punitive-proof-of-stake-algorithm/