Date of acceptance Grade

Instructor : Prof. Sasu Tarkoma
Advisor : Oscar Novo

Using Blockchain Technology and Smart Contracts for Access Management in IoT devices

Rupsha Bagchi

Helsinki May 8, 2017
UNIVERSITY OF HELSINKI
Department of Computer Science

HELSINGIN YLIOPISTO — HELSINGFORS UNIVERSITET — UNIVERSITY OF HELSINKI

Tiedekunta — Fakultet — Faculty	Laitos — Institution — Department
Faculty of Science	Department of Computer Science

Tekijä — Författare — Author
Rupsha Bagchi

Työn nimi — Arbetets titel — Title

Using Blockchain Technology and Smart Contracts for Access Management in IoT devices

Oppiaine — Läroämne — Subject
Computer Science

Työn laji — Arbetets art — Level	Aika — Datum — Month and year	Sivumäärä — Sidoantal — Number of pages
	May 8, 2017	80 pages + 9 appendices

Tiivistelmä — Referat — Abstract

The Internet of Things is a proliferating industry, which is transforming many homes and businesses, making them smart. However, the rapid growth of these devices and the interactions between these devices, introduces many challenges including that of a secure management system for the identities and interactions of the devices. While the centralized model has worked well for many years, there is a risk of the servers becoming bottlenecks and a single point of failure, thereby making them vulnerable to Denial-of-Service attacks.

As a backbone of these interactions, Blockchain is capable of creating a highly secure, independent and distributed platform. Blockchain is a peer to peer, distributed ledger system that stores all the transactions taking place within the network. The main purpose of the servers that form a part of the distributed system is to provide a consensus, using various consensus algorithms, on the state of the blockchain at any given time and to store a copy of all the transactions taking place.

This thesis explores the Blockchain technology in general and investigates its potential with regard to access management of constrained devices. A proof of concept system has been designed and implemented that demonstrates a simplified access management system using Ethereum Blockchain. This was done to check whether the concept can be applied at a global level. Although the latency of the network depends on the computing power of the resources participating in the Blockchain, an evaluation of the proof of concept system has been made, keeping in mind the smallest device that can be involved in the consensus process. Docker containers have been used to simulate a cluster of the nodes participating in the Blockchain, in order to examine the implemented system. An outline of the various advantages and the limitations of Blockchains in general, as well as the developed proof of concept system, has also been provided.

ACM Computing Classification System (CCS):
C.2.0 [General]: Security and Protection,
C.2.4 [Distributed Systems]: Distributed applications,
C.3 [Special Purpose and Application-Based Systems],
C.4 [Performance of Systems]: Reliability, availability, and serviceability,
C.5.3 [Microcomputers]: Portable devices,
H.3.4 [Systems and Software]: Distributed Systems

Avainsanat — Nyckelord — Keywords
Internet of Things, Blockchain, Ethereum, Smart Contracts, Device Management, Docker

Säilytyspaikka — Förvaringsställe — Where deposited

Muita tietoja — övriga uppgifter — Additional information

Contents

List of Figures

List of Tables

1 Introduction

Blockchains have been around for 9 years now, with the introduction of Bitcoins by Satoshi Nakamoto. While there is still a lot to learn about the blockchain technology and its potential, increasingly many people are starting to grasp the foundation behind blockchains and realize that the use cases of blockchains lie beyond *cryptocurrencies.* Some organizations such as Ethereum and Rootstock have created decentralized application platforms on top of blockchains. The decentralised platforms have the ability to run blocks of code, thereby giving an opportunity to build decentralised software programs.

This thesis aims to investigate the potential of the Ethereum blockchains to power constrained devices and to find out whether it is possible to implement the *access management* aspect of it, at a global level. This is done through implementation and evaluation of a simplified access management proof of concept over the Ethereum blockchain. Chapter 6 dives deeper into the architectural and implementation aspects of the system built for this thesis.

1.1 Motivation

One potential use of blockchain is its application in the realm of Internet of Things (IoT). Current IoT systems rely on centralized or brokered paradigms with huge computational and storage capacities, also known as the client-server model. The existing Internet of Things setups are therefore expensive, owing to the high costs associated with cloud server infrastructure(s) and maintenance, as well as other factors such as network equipment. Moreover, there is no existing platform that supports communication between all devices, and also a lack of guarantee that the services offered on cloud by different mobile device manufacturers are interoperable. While the client-server paradigm has been instrumental in connecting generic devices with each other for decades, it will not be able to support the challenges that stem from the burgeoning growth of the IoT economy.

Making use of a standard peer-to-peer decentralized communication approach will not only reduce the costs corresponding to maintenance and infrastructure of server clusters, but will also share the processing and space requirements of a huge number of devices on the IoT network, without sharing any additional resources. Blockchain provides a solution that suits the need for such a platform. It is an immutable ledger that is distributed among the participants registered on the Blockchain, and therefore cannot be manipulated or require middlemen.

1.2 Research Questions and Objectives

Usually, centralized Authentication, Authorization and Accounting servers [AbW03] have been used by the management systems to handle user requests for access to different constrained device resources. While a centralized resource management system is a good idea for private entities such as organizations, the Internet of Things is designed to work as a global interoperable system, since various constrained mobile devices can belong to different management communities during their lifetime. One example of this is the different parts of a vehicle manufactured and assembled by different organisations. Registering all the devices in a common blockchain would allow customers as well as sellers to independently communicate with other devices on the blockchain and be able to access the information over time. This in turn would lead to increased trust and transparency.

This thesis aims to investigate how to make Identity and Access Management of IoT devices more autonomous and distributed, without having to trust a single entity to store all the data securely. There are distinct differences between the traditional way device management processes are handled and using the blockchain as a database, which this thesis attempts to show. The main research questions this thesis seeks to answer are the following.

1. *Is it possible to implement IoT device management, specifically in constrained devices, on blockchains, on a global level?*
2. *What advantage does the use of blockchain bring to IoT?*
3. *What are the caveats and challenges of using blockchains for management of constrained devices, specially considered at a global perspective, and how can they be mitigated?*

2 Methodology

This section describes the research methodology used to carry out the research for the questions explained in the previous section *Research Questions and Objectives*. The main technique was adopted from the main guidelines described in [VaK15], which explains Design Science (DSR) as a research methodology usually used in Computer Science. DSR is also extensively used in the fields of education, health-care, and engineering.

This DSR methodology follows a process model, with specific sequential phases as illustrated in Figure 1. The methodology involves a problem statement which is investigated through the design of an artifact, which in turn is evaluated to assess whether the initial problem was solved. The following paragraphs explain about the different phases of DSR in more detail.

Awareness of Problem The methodology starts with a problem statement, the awareness of which, may come from multiple sources such as a new development in the industry or from a reference discipline. For instance, in this case, IoT devices need to be managed globally in a more secure yet transparent way. The blockchain technology can be made use of here.

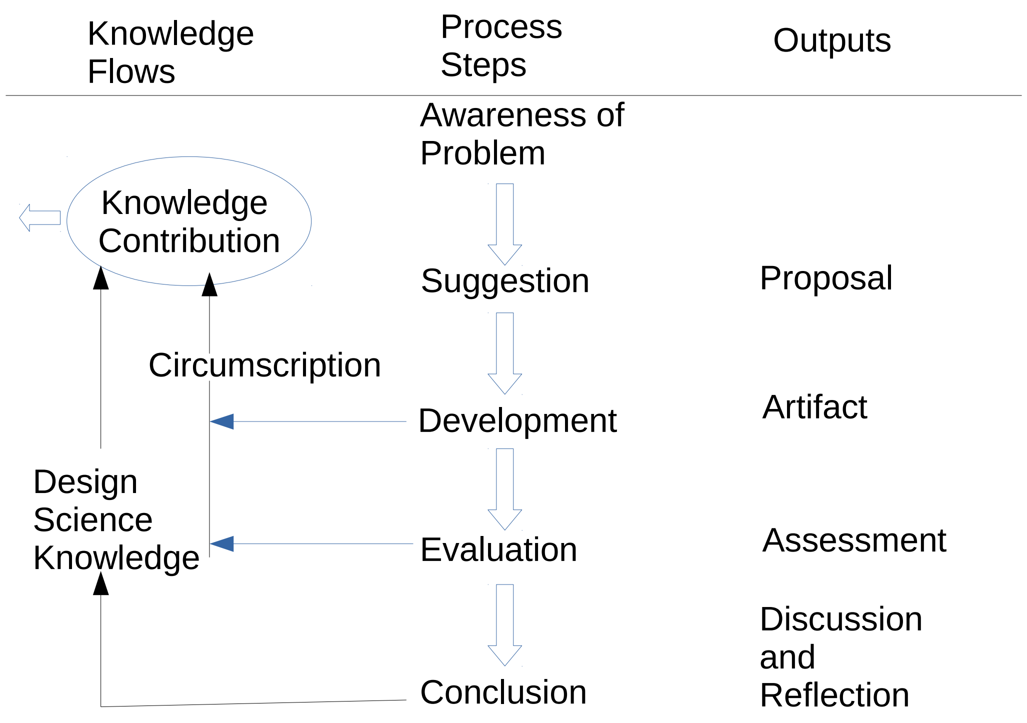


Figure 1: Design Science Research Process Model (DSR cycle)

Suggestion The suggestion phase follows after the awareness phase and includes a proposal along with a tentative design of the proposed solution. The tentative design also comprises of the specifications of the planned artifact. This phase, therefore, is a creative step where new solutions can be visualised based on the combination of existing and new solutions. In the case of this thesis, through researching the existing IoT device management systems and blockchain technology, an attempt has been made to implement a proof of concept that can facilitate device management over blockchains.

Development In this phase, the tentative design proposed in the Suggestion phase is further developed and implemented. During this phase, the proof of concept system is realised based on the suggestions proposed in the previous phase.

Evaluation Once the artifact is implemented, it is analysed and evaluated using various criteria as deemed implicit or as defined in the proposal. Evaluation of the artifact can include various methods such as observing, testing or by the way of experimenting. Deviations from the expected results are examined and discussed about in the scientific document. Section 7.1 under the Evaluation chapter talks in detail about various test methodologies, and the ones considered for this system.

Conclusion This phase is the supposed end of the research cycle, or the ending point of a particular research effort. The results can then serve as a base for further research or for development of a larger system.

3 Blockchain Technology

The introduction of Bitcoin [Sat08] in 2009 introduced a new era in the digital world. It illustrated some novel approaches to computing such as decentralization, where no one server holds control of the network, immutability, where data once written can never be erased, as well as the ability to issue and transfer currency recorded into an open ledger called the Bitcoin blockchain, without any third party. As people started delving more into the blockchain technology underpinning Bitcoin, they realized that the scope of blockchain itself can be extended to a much wider range of applications. Theoretically, the transactions on the blockchain could be used to represent more than money, e.g. complex agreements between two parties.

A few aspects of these solutions that should not be neglected, are the cost of the services and the fact that too much control is given to the data administrators, or middlemen. That is where blockchain technology is at an advantage. Blockchains are immutable, and serve as ledgers that contain records of every transaction ever made among participants in a given network, encrypted into blocks using a cryptographic hash function SHA256[1]. The technology supports a feature called smart contracts that validate whether terms are met by each party.

Blockchains have been classified into three main types according to the level of access granted to the users participating in the network. [Gav14] categorizes it into public blockchains, private blockchains and consortium blockchains. Public blockchains are those that can be participated in and read by anyone; Consortium or permissioned blockchains are ones where the miners are chosen in advance and therefore are partly decentralised; and Private blockchains are those that can be read by anyone but written to only by private parties such as organisations. In this thesis, the intention of the larger system is mainly to have a mix of consortium and public blockchains. The smaller proof of concept system this thesis uses however, makes use of a private blockchain to test the system.

The subsequent sections delve into the details of how Blockchain Technology operates and the real world implementations of Blockchains.

[1]SHA256 stands for Secure Hash Algorithm 256-bit

3.1 Architecture

The network architecture of a Blockchain distributed network is peer-to-peer, and is based on top of the network layer. Peer-to-peer network, also known as P2P, refers to a group of computers acting as a node for sharing files within themselves. The Blockchain, therefore, runs on a distributed network of servers, also known as *nodes*. These nodes in the network serve the purpose of providing a consensus on the state of the blockchain at any given time, and have a copy of the blockchain stored in them. The fundamental application of the Blockchain is a transaction ledger, sort of like a secure public ledger, that stores all the transactions that take place within the network. This makes it a very secure and transparent decentralised system.

3.1.1 Network Architecture

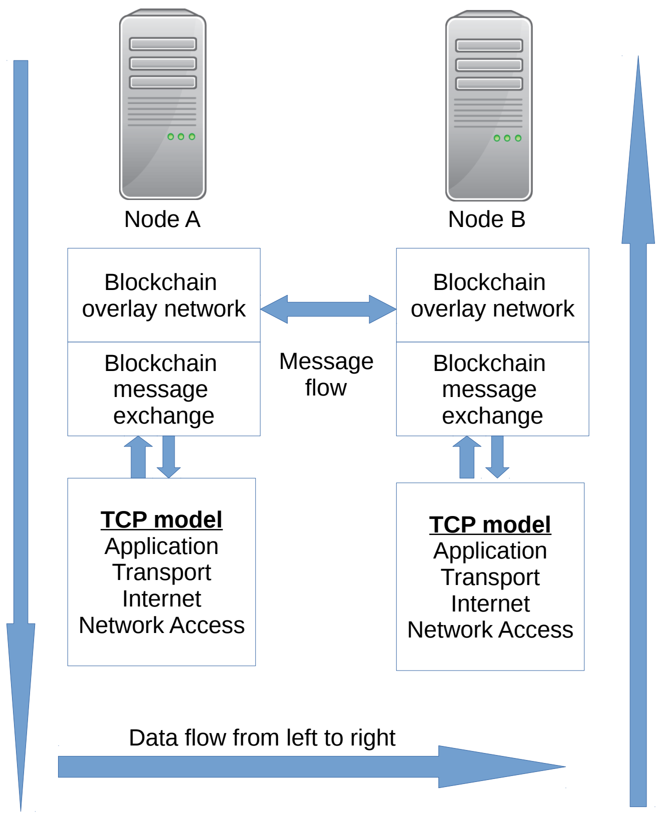

Node A
Node B
Blockchain overlay network
Blockchain overlay network
Message flow
Blockchain message exchange
Blockchain message exchange
TCP model
Application
Transport
Internet
Network Access
TCP model
Application
Transport
Internet
Network Access
Data flow from left to right

Figure 2: Blockchain Network Architecture

3.1.2 Nodes

The nodes or clients connected to the network in the Blockchain system are an essential part of the system. They support various functions such as routing, mining, storing the blockchain data and serving as a wallet. All nodes participate in verification and propagation of transactions, and are enabled with features including discovering and maintaining connection with their peers. They also maintain a copy of the ledger, or the blockchain, which contains data about all the transactions that have ever happened, thereby eliminating the need of having a centralised server to store it. Nodes can also act as **Miners**, and help verify and validate all the transactions made by all the users. All miners are nodes, but all nodes are not necessarily miners.

The nodes are mainly of three kinds namely, full nodes, **simplified payment verification (SPV)** clients and web clients. Web clients, usually called *wallets*, are stored on third party servers and can be accessed through web browsers. SPV nodes usually include clients that do not possess enough hardware capabilities like mobile devices, or more constrained devices like embedded systems. These SPV nodes do not need to store a copy of all the transactions in the blockchain, and instead store only the block headers. These nodes have a slightly different way of verifying transactions from the full nodes, since they do not keep a track of all the transactions taking place on the blockchain. They depend on their peer nodes to provide the transaction information they require, on an on demand basis. Full nodes, on the other hand, are the nodes that store an up to date copy of the blockchain in its entirety.

Once the nodes are connected to the Blockchain network, they start looking for other peers to connect to, on a particular port over TCP. This process of node discovery is also called as the discovery protocol.

3.1.3 Transactions

Transactions are data structures stored in files called blocks. They are not encrypted and are typically linked to previous transactions, thus forming a chain. A digital currency owner digitally signs the previous transaction with their public key and a hash is created out of it. The owner of the previous transaction then signs the hash with their private key. Figure 3 illustrates a simplified version of the ownership chain, as explained in [Sat08]. In more complex cases, the number of inputs and outputs can be multiple. A transaction contains a number of fields as shown in Table 1.

Size	**Field**	**Description**
4 bytes	Version	Specifies which rules this transaction follows
1-9 bytes (VarInt)	Input Counter	How many inputs are included
Variable	Inputs	One or more transaction inputs
1-9 bytes (VarInt)	Output Counter	How many outputs are included
Variable	Outputs	One or more transaction outputs
4 bytes	Locktime	A Unix timestamp or block number

Table 1: Fields in a transaction

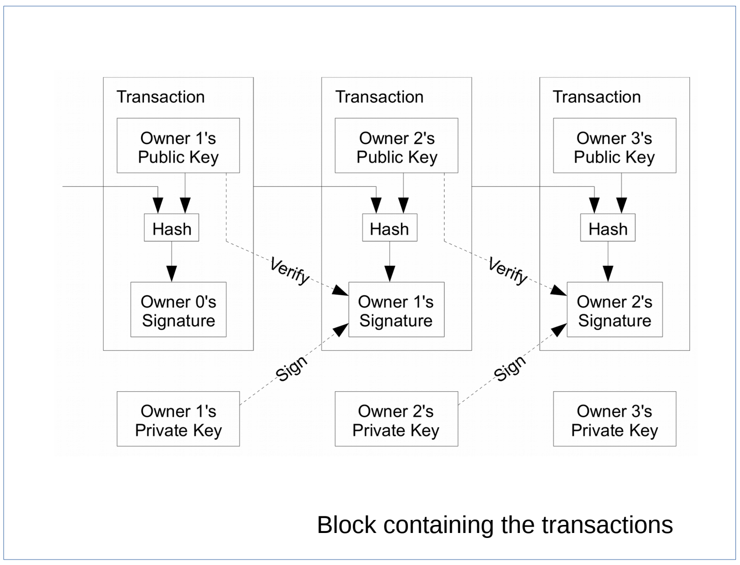


Figure 3: Transaction ownership chain

Most transactions also include a fee that is paid to the miners to keep the blockchain consistent and secure. These miners mine a block which records the transactions on the blockchain and in return receive the transaction cost as an incentive. Most wallets calculate these transaction fees automatically.

3.1.4 Blocks

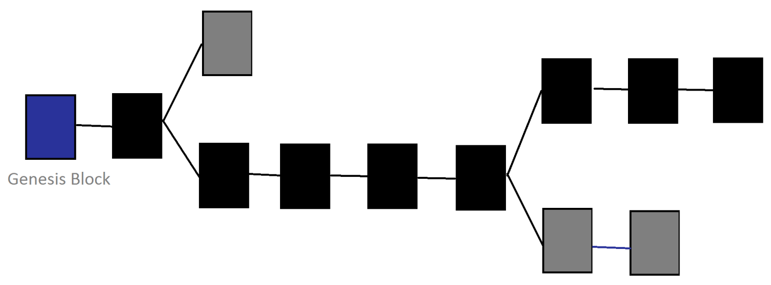


Figure 4: Blocks in the blockchain

Figure 4 illustrates the linkage of blocks in the Blockchain. The blocks marked in black show the current active blockchain. The ones marked in grey are called stale blocks. A detailed explanation about this has been provided in Section 4.1.3. The Blockchain is technically an ordered and timestamped linked list of blocks, that provides a record of all the transactions that have ever happened. Each of these blocks is linked to the previous one, i.e. its parent, through a unique hash. These hashes are generated through the SHA256 hashing algorithm. In other words, the header of each block contains a reference to its parent's hash. This linking continues all the way up to the first block in the blockchain, also known as the **genesis block**. The genesis block is the leftmost blue block indicated in Figure 4.

A block can have multiple children simultaneously. Each child refers to the same parent block's hash. Ultimately, one of these child blocks becomes the part of the main blockchain. This phenomenon is known as **forking**. This happens when multiple miners mine and verify different blocks at the exact same time. The blockchain is also called immutable since it is an astronomical expense to recompute the hashes of all the blocks starting from the genesis block. Table 3.1.4 describes the various fields in a block along with their sizes. Table 2 further describes the structure of a block header, explaining the different kinds of metadata associated with the blocks.

Since the block header is a part of the block, a complete block, including all the transaction information, is much bigger in size than a block header. This is the reason, SPV clients and wallets download just the header files of the blockchain while

Size (bytes)	**Field**	**Description**
4	Block Size	The size of the block
80	Block Header	Consists of several fields, as shown in Table 3
1-9	Transaction Counter	Number of transactions included in the block
Variable	Transactions	Recorded transactions included in the block

Table 2: Structure of a Block

retrieving their desired block's info from the full nodes connected to the network. Table 3 shows the various fields included in a block header.

The *merkle root* field refers to the root of a merkle tree that stores the transaction information in every block. A detailed explanation of Merkle trees is provided at Section 2.1.5.

The *timestamp*, *difficulty* and *nonce* fields are connected to the transaction verification process that the bitcoin miners compete for.

Size(bytes)	**Field**	**Description**
4	Version	Software version
32	Previous Block Hash	Reference to hash of the previous block
32	Merkle Root	Hash of the root of the merkle tree containing all the transactions included in the block
4	Timestamp	Approximate time when the block was created
4	Difficulty Target	Proof of work difficulty for the block
4	Nonce	Proof of work counter

Table 3: Block Header structure

[Gav14] explains the validation process of the blocks and decision making process that leads to the addition of a block to the blockchain, using the blockchain paradigm. The process takes place in the following sequence. The system first checks whether the previous block hash that the current block references to, is valid. The time-stamp of the block is then checked, since its value needs to be greater than that of the previous block and less than two hours into the future. The validity of proof-of-work is then checked. If the state of the previous block when compared with the list of transactions returns true, the new block is added on to the blockchain.

Block hash and height

There are two ways to identify a block. First, by its hash. This hash is computed by the peer nodes in the network every time a block is generated. The hash could be stored in a database included in the block's metadata for faster indexing and retrieval of the blocks from the disk. The second way to identify a block would be by its height. The genesis block is at height 0. This method of identification is not absolute because two or more blocks in the blockchain can have the same block height, and it is also possible that two blocks of the same height have the same parent.

3.1.5 Merkle Trees

Each block in the blockchain contains a *Merkle tree*, that stores the transaction information. Merkle tree [And14], or a hash tree, is a data structure that helps with verification and summarizing of large sets of data. They efficiently store all the transactions in the block, and their root contains a hash of all the transactions in the block. The merkle trees facilitate the SPV nodes to download just the block headers and still be able to identify whether a transaction is included in a particular block, by obtaining the small merkle path from a full peer node. Figure 5 illustrates how merkle trees are used in transactions in a block on the blockchain.

Merkle Tree creation

Merkle trees are created in a bottom up manner by recursively using double SHA-256 Hash. Therefore, the leaves of a Merkle tree store hashes of the transactions pertaining to a block, rather than the transactions themselves. The following code snippet, which uses the bitcoin library, demonstrates how a merkle tree is constructed from the ground up, as mentioned in [And14].

```
#include <bitcoin/bitcoin.hpp>

bc::hash_digest create_merkle(bc::hash_list& merkle)
{
    if (merkle.empty())
        return bc::null_hash;
    else if (merkle.size() == 1)
```

```
        return merkle[0];

    while (merkle.size() > 1)
    {
        if (merkle.size() % 2 != 0)
            merkle.push_back(merkle.back());

        bc::hash_list new_merkle;
        for (auto it = merkle.begin(); it != merkle.end(); it += 2)
        {
            bc::data_chunk concat_data(bc::hash_size * 2);
            auto concat = bc::make_serializer(concat_data.begin());
            concat.write_hash(*it);
            concat.write_hash(*(it + 1));
            assert(concat.iterator() == concat_data.end());
            bc::hash_digest new_root = bc::bitcoin_hash(concat_data);
            new_merkle.push_back(new_root);
        }
        merkle = new_merkle;
    return merkle[0];
}
```

The above-stated snippet of code works in the following way. It creates a Merkle tree from the leaf nodes and calculates SHA-256 hashes until the root of the tree. The first if conditions checks if the list of transaction hashes is empty, and stops in that case. Otherwise, the second condition is executed, and the hash list is evened out. The code then loops two hashes, concatenates them and subsequently hashes the combined hashes to form their parent node. Continuing in this fashion, the code ends up with the root of the Merkle tree, which includes a hashed concatenation of all the hashes of the transactions existing in the Merkle tree nodes and leaves.

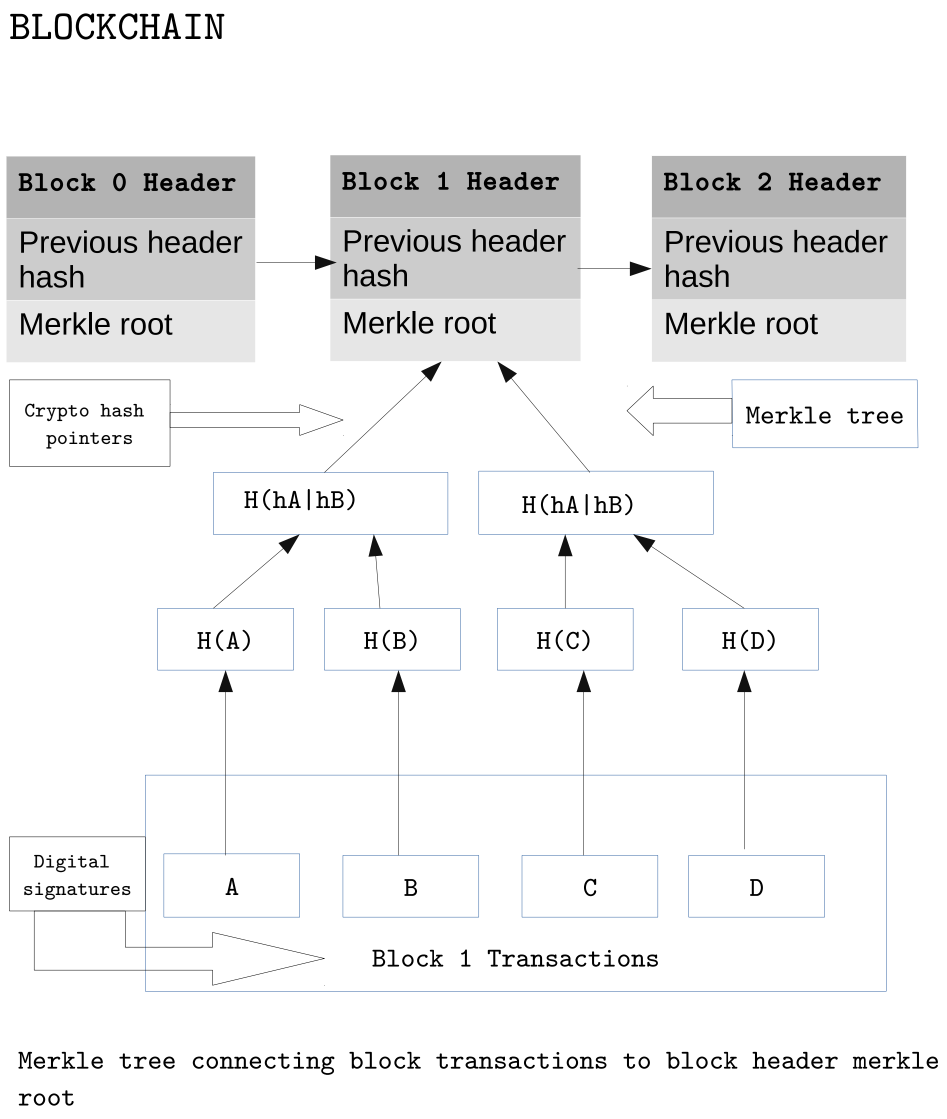


Figure 5: Merkle Trees in blockchain

Simply put, merkle trees are used to connect transactions to blocks in the case of Blockchains. Every block header contains a single hash, i.e. the merkle root, which in turn is the root of a merkle tree that contains all the transaction id hashes. It has several layers of checksum that could, therefore, be used to validate individual transactions or even verify a block. Since every merkle root contains a hash of all the hashes of transactions in the block, it is not necessary to have a copy of the entire tree. This also helps with pruning the blockchains, as we will discuss later in the *Discussion* section.

3.1.6 Mining

There are two major benefits of mining for the blockchain network namely, validation and verification of transactions. Mining also generates new digital currency coins to the network, since these newly minted coins serve as the reward to the miner who solves the next block in the blockchain.

The first step to mining is calculating the *difficulty level* of the blockchain. All the full nodes connected to the blockchain network recalculate this difficulty level after certain intervals. The level can either increase or decrease based on how long it takes to generate the certain interval of blocks. In the case of Bitcoins, which was the first implementation of blockchains, the interval is 2016 blocks. Thus, the full nodes have to rehash the difficulty level after every 2016 blocks, which leads to an average consensus time of 10 minutes. As the number of miners increase, the rate of block creation also increases. This is turn, leads to an increase in the difficulty level, since it pushes the rate of block creation down to the original time of 10 minutes, in the case of Bitcoin.

The miner then downloads all the transaction and block information that happened previously, and constructs a merkle tree out of them, eventually generating a merkle root.

In order to form a block, a miner is free to choose the number of transactions they want. The only limiting factor to that is the *maximum block size.* As an example, in case of Bitcoins, the maximum block size is 1 MB, and so, depending on the transaction size, a maximum number of transactions that occupy upto 1 MB space can be accommodated in the block. In Ethereum, the maximum block size is not in terms of the storage space it occupies, but rather in terms of the maximum gas limit that it can allow. The role of gas and how it is calculated has been discussed at length in Section 4.2.3. A detailed account of how mining takes place in the Ethereum system is mentioned in Section 4.2.4.

The miner then generates the block's hash from the block header values and then compares the hash to the current difficulty level of the network. This hash is generated using one of the few consensus algorithms used in blockchain systems, which we delve into, in Section 2.2.

The pseudo code below, where H(n) and H(b) refer to the difficulty level of the new block and the existing block difficulty level respectively, shows the conditions that a block must satisfy in order to be accepted.

```
If (H(n) > H(b)) && (Block accepted by the network)
    Block is propagated and miner gets rewarded
else
    Block is rejected
```

There can also be a possibility of two or more miners generating a block at the same time. This leads to the formation of *Stale* and *Orphan Blocks*. Different implementations of blockchains deal with these stale and orphaned blocks differently, as will be discussed in Chapter 4 under the blockchain implementation section.

Solo and Pool Mining

There are two ways mining can be done on a blockchain network. The first being, *solo mining* and the second, *pool mining*. As the name indicates, solo mining refers to a single miner trying to generate blocks on the blockchain in exchange for rewards. But owing to the increasing difficulty of generating blocks, and maintaining the mining hardware costs, it is becoming increasingly difficult to mine alone. In case of pool mining, the group needs to have an agreement on reward and risk distribution for each miner. Every pool operates on their own set of rules, sometimes involving extra "fees" for the pool operator.

3.2 Consensus Mechanisms

Consensus is a fundamental problem in Distributed Systems that requires two or more agents to mutually agree on a given value needed for computational purposes. Some of these agents may be unreliable, and therefore the consensus process needs to be reliant. Thus, the need of consensus mechanisms is to facilitate secure updation of a process or a state, in accordance with certain state transition rules, where a distributed set has the right to perform the state transitions [Eth16].

The most commonly used consensus mechanisms in case of Blockchains are discussed in this section.

3.2.1 Proof of Work

There are two kinds of rewards a miner can receive for mining a block. First being, the transaction fees from all the transactions picked by the miner for the block formation, and second, the new digital currency created with every block. To be able to earn these rewards, the miners compete to solve a mathematical problem called *proof of work(PoW).* PoW is a consensus algorithm, and is really a brute force crack on a SHA256 hash algorithm. This works as a proof that the miner has used some heavy duty computing resources. The exact condition is that the double hash of every block must be less than the difficulty target[Eth16].

51% attack

Besides the fact that proof of work algorithm consumes a lot of energy, there is another flaw to it called the *51% attack.* If a single entity were to contribute to more than 51% of the bitcoin network's mining, they would be able to fully control the network and modify the ledger according to their needs. While this attack is theoretically possible, it would cost the miners an enormous amount of money as well as computational power[ABC16].

3.2.2 Proof of Stake

As discussed earlier, PoW algorithm estimates how much of the network agrees on the blockchain's existing state at that point, using the difficulty and nonce fields. This requires a lot of computational resources, which in turn requires a lot of expenditures on the part of the miner. This is the main reason miners are compensated with adequate digital currency, which are created whenever a block is mined and added to the blockchain. Therefore, motivated by the problem of energy efficiency and usage of a huge amount of computational resources, a consensus algorithm called Proof of Stake was suggested.

In case of the proof of stake (PoS) algorithm, the consensus is achieved on how much of the cryptocurrency agrees with the present state of the blockchain. PoS therefore, requires the cryptocurrency owners to prove that they own a certain amount of cryptocurrency which equates to their *stake* in the currency [But16]. The required amount of coins (also known as the target) to be able to be a part of the proof of stake network is usually set in advance by the network using a difficulty adjustment process similar to that of proof of work.

3.2.3 Proof of Burn

Proof of burn[Ste14] is an alternative distributed consensus algorithm to PoW and PoS. The idea is to destroy or *burn* some coins by sending them to an unverifiable address on the blockchain network. This helps reduce the total supply of currency on the blockchain and increase the value of the coins in circulation. Destroying some of their currency would ensure that the miners get lifetime rights to mine, since in this case the chances of being allowed to mine is a lottery among the owners of destroyed coins. Therefore, the more burnt coins, higher are the chances of the miner to win in the lottery. Different implementations of proof of burn handle the lottery system differently. One such system that implements proof of burn in their cryptocurrency system, is called Counterparty [2].

3.2.4 Delegated Proof of Stake

Delegated proof of stake(DPoS)[Org15] combines proof of work and proof of stake characteristics and uses a decentralized voting system. The distinguishing factor in DPoS is that the top 101 delegates are the ones securing and forging the network. These delegates are determined by their stake in the network just as in proof of stake.

3.3 Double Spending

One question that arises from the use of digital currency is that of double spending [Sat08] i.e., currency counterfeit. Can the cryptocurrency owner be sure nobody else is falsely claiming ownership to their coins? How would one know if the amount the sender send, actually goes into the blockchain? Physical currency notes solve the problem easily because the same note cannot be given to two different people at the

[2]Counterparty: For more information - http://counterparty.io

same time. For a virtual currency, the main issue seems to be that of synchronisation. In case of Bitcoins, if a user attempts to spend the same bitcoin twice for example, the miners reject both the transactions to prevent the same bitcoins from going two separate ways. However, if the user does succeed in double spending, one of the receiving parties does not receive any bitcoins. In this case, the accepted transactions is included in the blockchain and is irreversible. Therefore, in effect, Blockchains eliminate the issue of double spending.

4 Blockchain Implementations

This chapter talks discusses about various Blockchain implementations, with more focus on the Bitcoins and Ethereum systems. The chapter is organized as follows. Section 3.1 explains the Bitcoins system, Section 3.2 explains the Ethereum system and Section 3.3 talks about some other well known Blockchain implemetations. Table 5 gives an overview of the various blockchain systems discussed in this chapter and their salient features.

4.1 Bitcoin

Bitcoin is a cryptographic currency (also known as a cryptocurrency), that serves as a digital financial asset, and was created by programmer(s) under the pseudonym of Satoshi Nakamoto. Bitcoin uses public key cryptography, peer to peer networking and proof of work to make transactions and verify them.

The main idea behind proposing Bitcoins was not to rely in the trust of centralised banks, since they invest people's money in the market with very little in reserve[Sat08]. Therefore, Bitcoin focuses on providing total transparency. Every time a bitcoin owner makes a payment, the transaction is broadcast over the network and recorded in the blockchain forever, thereby making it an immutable transaction.

4.1.1 The Bitcoin Network

Bitcoin uses public key cryptography, peer to peer networking and proof of work to make transactions and verify them. It was the first blockchain to be conceptualised and implemented.

The Bitcoin system is programmed such that a new block is created once every 10 minutes. Even if a block were to get created in under the stipulated time of ten minutes, albeit with a very small chance since it is very computationally expensive to be able to do so, it can only happen for a very short time. The difficulty adjusts itself until the block creation time becomes ten minutes again, which is usually after every 2016 blocks in case of Bitcoins. The 10 minute block creation time, as described in [Sat08], was chosen as a compromise between propagation time of new blocks in large networks and the amount of work wasted due to chain splits. A faster block creation time would mean that there could be more chances of many miners mining a block at the same time, and that would lead to more forks. If a fork is not a part of the *longest* chain, it becomes an orphan block. We discuss more about orphan blocks in Section 3.1.3. The choice of ten minutes also takes into account, the slow network speeds in many parts of the world.

4.1.2 Transactions and Scripting language

An interesting fact to note is that there are no balances in bitcoin, or rather there are unspent transaction outputs (UTXO) in the blockchain. Whenever some bitcoins are received, they are recorded as UTXO. The receiver then scans the blockchain and adds up all the UTXO associated with their account, and figures out how many they own. Thus, sending someone a bitcoin actually means creating a UTXO corresponding to the receiver's address, which they can spend. An output typically consists of two fields namely, the amount and a locking script. A *satoshi* is the smallest denomination of the amount that can be sent. The locking script sets out conditions that need to be fulfilled in order to spend the UTXO.

The Bitcoin nodes follow a scripting system to validate transactions. It follows a scripting language called *Script*. When a transaction is validated, an unlocking script runs simultaneously with the UTXO's locking script to verify whether the condition that facilitates spending is satisfied. There are five standard transaction scripts including pay to public key hash, multi signature, pay to script hash, public key and OP_RETURN. OP_RETURN is the only transaction output that allows any data not related to the transaction itself. This data cannot be spent, and can be of 40 bytes. Needless to say, it does not form a part of the unspent UTXO calculations for account balance.

4.1.3 Stale and Orphaned Blocks

Often during mining, there are many blocks that do not end up becoming a part of the blockchain. There can be several reasons causing this phenomenon. *Stale blocks* are those that have a preceding block, but are not a part of the main blockchain. One possibility of a block turning out to be stale is when two miners produce valid blocks simultaneously and one of them is propagated into the network slower than the other one. The slower block ends up being discarded, and the more widely accepted block becomes a part of the blockchain. This is more common when the difficulty is low, which lets many miners solve the proof of work algorithm in a short time frame. Such situations result in a conflict, and usually the computationally

longest chain wins. The **length** of a chain, in this case, is measured by the amount of work it took to find an individual block in the chain. Figure 6 illustrates this with an example. In this case, when block B3 is mined, the second chain, i.e. chain B is chosen because it is longer.

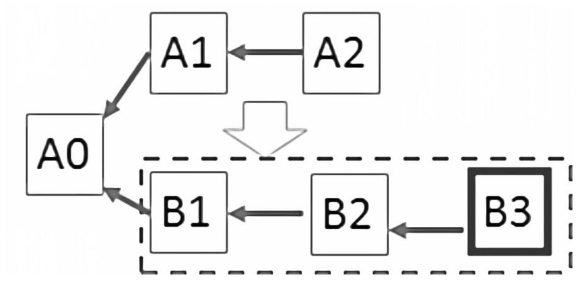


Figure 6: Longest chain selection in Bitcoins

Orphaned blocks, on the other hand are those blocks that do not have a valid parent. When a miner downloads an orphaned block, they first have to find and validate the unknown/missing parent block, and then proceed to validate the former orphaned block.

As many miners compete to generate blocks on the blockchain, the possibility of having stale or orphaned blocks is quite high. The full nodes in the blockchain network, keep such blocks in the memory, while trying to validate the ancestry of the orphaned blocks. These blocks are evetually discarded and the rewards or the transaction fees associated with these blocks become *unspendable*. This phenomenon of generation of orphaned blocks is also known as **Chain Reorganization**. The latest version of the Bitcoin system, at the time of writing this thesis, does not generate any orphan blocks anymore due to a recent change in their download mechanism[3].

4.2 Ethereum

Ethereum was designed by a Bitcoin developer named Vitalik Buterin in 2013, who wanted to build a platform to facilitate development of decentralised applications, also known as *DApps*, on top of the blockchain. Ethereum has its own cryptocurrency called *ether*, and an internal currency to pay for computations and transaction

[3]Change in Bitcoin 1.0: http://bitcoin.stackexchange.com/questions/5859/what-are-orphaned-and-stale-blocks

fees called *gas*. Ethereum uses PoW as its consensus mechanism, but plans to also include PoS mining according to the foundation's development roadmap plans[But16].

The decentralised applications can be programmed with a built in Turing complete language [Tel94] called *Solidity* [Eth16]. An advantage of turing complete languages is the ability to have loops [Gre15], which help perform required operations multiple times without writing too many lines of code. To make the execution more efficient, the code written in Solidity is compiled into an alphanumerical bytecode and then executed by the nodes participating in the blockchain. We discuss more about this later in this chapter.

4.2.1 Ethereum Accounts

In Ethereum, an account is an object with a unique 20 byte address together with state transitions. Transitions refer to the data and value transferred between two accounts. There are two kinds of accounts namely, **Externally Owned Accounts (EOA)**, that work with private keys and **contract accounts**, that are generated when a contract code is uploaded into the blockchain. EOAs do not contain any code, and are meant for signing transactions to produce messages from these accounts. Contract accounts act on those messages which usually result in activating the contract code that these accounts contain. A contract is basically a software application that exists in the execution environment and is triggered whenever a message triggers its code. A contract account has its own contract memory that allows the persistent variables of the contract to be stored.

An account contains the following information. A transaction counter, that ensures a particular transaction takes place only once, balance of the account, contract code if any, and its own memory which usually houses contract specific data structures and the values stored in them. Ethereum uses Merkle trees to keep a track of all the EOAs and contract accounts.

Difference between Transactions and Messages

[Eth16] says that a transaction in Ethereum is a signed data package that contains a message to be sent from an EOA. Transactions contain the fields as described below:

- Transaction receiver
- Sending EOA's signature

- Value of ether to be transferred
- Data field
- STARTGAS value, the maximum number of possible computational steps in the transaction execution
- GASPRICE value, fees paid by the sending EOA for every computational step

The last two fields have been introduced in Ethereum transactions to prevent denial of service attacks. The reason they decided to set a limit on the number of computational steps for a transaction, was to avoid accidental infinite loops or resource wastage in the code. As discussed earlier, the unit used to measure fees per computation is called *gas*. Certain operations that involve more computations or require storing information as a part of the state involve more gas expenditure. This way the attacker will need to pay an appropriate amount of gas for every resource they utilize.

Messages and transactions are almost identical, except for the fact that messages are produced by contract accounts, and that these messages exist only within the execution environment. The fields that a message is comprised of are:

- Message sender (contract account)
- Message receiver
- Value of ether if needed
- Data field
- STARTGAS value

4.2.2 GHOST protocol

The main idea of Greedy Heaviest Observed Subtree (GHOST) [Eth16] was to **securely** cut down the consensus time for the bitcoin blockchain. This protocol tries to solve two problems. First, the problem concerning reduced security of the blockchain. The current process involves a high possibility of having stale blocks, and having a faster confirmation time would increase the stale rate much more. To deal with this issue, the calculation must involve taking the stale blocks into account when trying to calculate the **heaviest** chain. This means that the calculation of the chain into which maximum amount of total work was done, includes the stale descendants of the block's parent. Due to this, it is possible to increase the rate of transaction without compromising the security of the blockchain. In Ethereum, the stale blocks are referred to as *Uncles*.

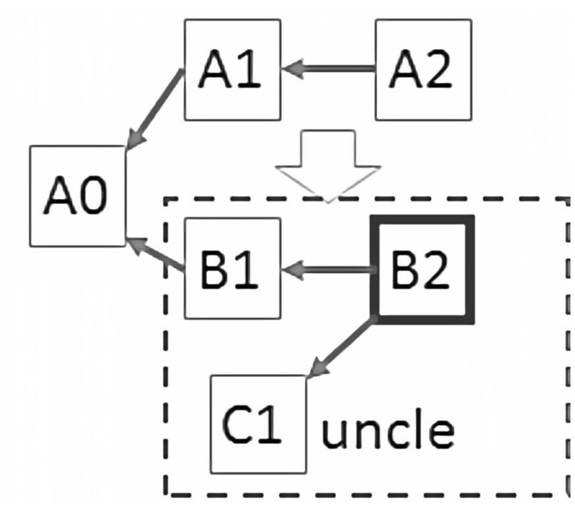


Figure 7: Ethereum GHOST protocol

Ethereum implement GHOST in a simplified way, wherein it goes only seven generations down. The way it has been defined in the Ethereum code is as follows.

1. A block should specify atleast one Parent and zero or more Uncles.
2. An Uncle of block A must have the following characteristics:
   - Should be different from all uncles included in previous blocks and all other uncles included in the same block.
   - Cannot be an ancestor of block A.
   - Should be a valid block header
   - Should be a direct child of the k^{th} generation ancestor of A, where

     `2 <= k <= 7.`
3. For every uncle U in block A, the miner of A gets an additional 3.125%, and the miner of U gets 93.75% of the standard mining reward.

According to [Eth16], Ethereum implements a limited version of the GHOST protocol due to two reasons. Firstly, an unlimited version would be too complex with all the calculations involving which Uncle block is valid. Secondly, the unlimited version of GHOST would make the miners greedy to mine the Uncle blocks, thereby making them vulnerable to mining on possibly the chain of a public attacker. By incentivising uncle mining, the Ethereum system acknowledges the energy spent on the generation of stale blocks.

4.2.3 Smart Property and Smart Contracts

Smart contract [Sza97] is a theoretical concept developed in 1994 by Nick Szabo to create self-enforcing and perpetual computer software aimed at replacing legal contracts. According to his description,

"*A smart contract is a computerized transaction protocol that executes the terms of a contract. The general objectives are to satisfy common contractual conditions (such as payment terms, liens, confidentiality, and even enforcement), minimize exceptions both malicious and accidental, and minimize the need for trusted intermediaries. Related economic goals include lowering fraud loss, arbitrations and enforcement costs, and other transaction costs.*"

In effect, it means that a smart contract is a distributed contract that can serve as an agreement between people and the blockchain without the requirement that both parties trust each other.

Smart Property [Sza97], is property controlled using smart contracts. This property could be a physical things such as a house or a box, or a non physical entity such as market shares or even permissions to a file server. The main function of a smart property is that it can be dealt with in a trustless way.

In case of Ethereum, smart contracts are objects that have their own Ethereum accounts. These objects contain programs that can store data, make decisions, communicate with other contracts via messages, or even send money to other accounts. These contracts are established in the blockchain by their owners, but their execution is taken care of by the Ethereum network. They exist in the blockchain forever. Contracts are programmed in a turing complete language called *Solidity.* It can be thought of as a database slot that one can query or alter by passing messages to the contract to access its functions managing that database.

Smart contract code execution

By definition, a contract is a written or a spoken agreement between two or more parties. But in the case of Ethereum, a smart contract is more like an application containing instructions for itself, that is activated every time a transaction is made to its corresponding Ethereum account. These instructions are written in a scripting language supported by the cryptocurrency system which lets the smart contract access desired transaction, message or other pertaining information on the blockchain. The scripting language in the case of Ethereum is a high level Turing complete, stack based and object oriented language that is executable on the Ethereum Virtual Machine (EVM). The EVM initially supported two languages, Solidity and Serpent. But Serpent is not officially supported anymore. Solidity is influenced by Javascript, C++ and Go languages, whereas Serpent is built to be very similar to the Python programming language.

During the block validation process, each miner tries to verify the portion of the smart contract code responsible for the transactions with respect to the current block [Eth16]. Figure 8 shows how a smart contract resides in the blockchain network.

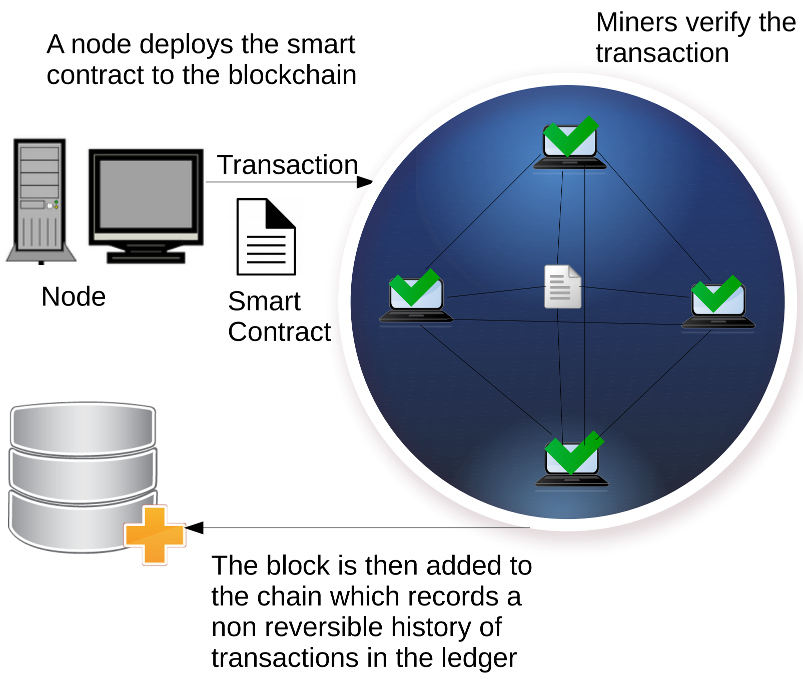

A node deploys the smart contract to the blockchain
Miners verify the transaction
Transaction
Node
Smart Contract
The block is then added to the chain which records a non reversible history of transactions in the ledger

Figure 8: Simplified view: Deployment of smart contracts in the blockchain

Gas

It is the coded logic in the smart contract that enables transfer of currency or data from one place to another, through *conditional transactions*. These conditional transactions in the blockchain require a certain amount of computational effort, depending on the type of operation, owing to the fact that the miner nodes in the network are the ones responsible for computing the logic and altering the state of the blockchain. In the Ethereum blockchain, the computational costs are determined by the unit called $\text{Gas}_{\text{price}}$ or *gas price*. $\text{Gas}_{\text{price}}$ is usually set by the developer, and if it is set within certain bounds, the transaction corresponding to that gas price may be accepted by the miners. Each operation requires a fixed amount of gas e.g., 3 units of gas for an ADD operation or 5 units of gas for a multiplication (MUL) operation. Table shows an example of some important operations along with the proportional gas units they consume.

Operation	**Gas units**
PUSH	3
ADD	3
MUL	5
SUB	3
OR	3
AND	3

Table 4: Gas consumption for some operations

Every transaction needs to be supplied with enough gas, also known as *startGas*, to support the computation costs and its storage. The residual gas is then given back to the user who initiated the transaction. The wallets usually calculate the amount of startGas to be supplied by themselves, without the user having to worry about it. The user can also calculate an approximate amount of gas required for the transaction. [Gav14] gives a detailed explanation of all the computations costs and how they can be calculated in Appendix G of their document. For example, to make a call to a smart contract on the blockchain, the calculation can be done in the following way.

$\text{Gas Price}_{/\text{transaction}} = \text{startGas} + (\text{operation}_{\text{gas}} * \text{Gas}_{\text{price}})$

In case the transaction runs out of gas before it is finished with all the operations, the transaction is reverted and the corresponding transaction fee is still paid to the miner.

Ethereum currency denominations

Ethereum has a metric system of currency denominations as units of ether, with one unit of *Wei* being the smallest unit. Each denomination has its own unique name, and some of them bear the names of prominent figures that have played a major role in the development of technology. For example, 1 babbage equates to 1000 Wei units.

4.2.4 Ethereum Mining

Ethereum uses a PoW system known as Ethash or Dagger-Hashimoto [DrB15], which requires a dataset known as a Directed Acyclic Graph (DAG) [Jen96] in order to verify the blocks. A DAG, is essentially a graph where a node can have multiple parents. Every full network node in the blockchain is required to generate a DAG file. The use of these directed acyclic graphs achieves *memory hard* computation, yet a validation that is *memory easy*. *Memory hardness* [But13] refers to the fact that in order to validate a block through PoW, a huge number of computations as well as a lot of memory are required. *Memory easy validation* means that lightweight clients are able to efficiently validate blocks. This subsection focuses on the specifications of this mining algorithm and the reason Dagger-Hashimoto was chosen for Ethereum.

As described in Section 3.2.1, in the case of Bitcoin, mining with PoW requires computation of SHA256 hashes in order to build consensus and create blocks in the blockchain. Many companies have built specialized hardware known as Application-Specific Integrated Circuits (ASICs) to compute these SHA256 hashes. These hardware serve no other purpose besides Bitcoin mining, and its dominance may lead to problems such as skewed distribution of cryptocurrency, or even a 51% attack. Memory hardness tries to solve this problem by making the memory a limiting factor, instead of CPU power. It can be argued that companies can now include terabytes of memory into their ASIC devices. But the advantage of having a memory limitation is that people who cannot afford ASIC devices, can add more memory cards to their existing hardware to improve performance. The subsequent paragraphs give a detailed description of the mining algorithm used by the current Ethereum blockchain system.

The general outline of the steps that the algorithm takes is as follows.

1. A *seed* is calculated for each block. A seed of a particular block, is a hash that is computed by scanning through all the parent blocks until the specific block. The algorithm below, illustrated using python code, is used to calculate the seed of a block.

   ```
   def get_seedhash(block):
       s = '\x00' * 32
       for i in range(block.number // EPOCH_LENGTH):
           s = serialize_hash(sha3_256(s))
       return s
   ```

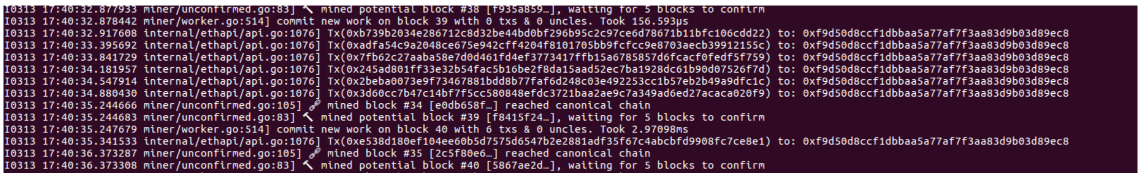
2. A *pseudorandom cache* is then computed from the seed obtained in the previous step. This cache is a 64 byte value of size 524288, and is computed by performing two passes of RandMemoHash algorithm [Ler14].

3. A *DAG* is then generated from the cache, which is stored by full network clients and mining nodes.

4. The DAG is used to mine the blockchain. It involves selection of random slices of the DAG dataset, combining it with the nonce value, and obtaining a hash. The Dagger Hashimoto algorithm specification explains how the DAG is generated.

The pseudorandom cache, computed in step 2, is stored by lightweight clients. It is used to verify desired blocks by regenerating the required pieces of DAG files from the cache. The size of the cache starts at approximately 16 MB, because the developers wanted to make it more resistant to ASIC hardware. The size of a DAG dataset was set to around 1 GB, in order to have a larger memory value than which the usual specialized memories are built. The cache size and the DAG size, both grow in a linear way [Eth15].

Figure 9 shows a snapshot of the mining console logs on the Go language implementation of Ethereum. The hammer symbols indicate a block being mined, and the lines beginning with 'Tx' depict a transaction bring recorded from one account to another.



```
I0313 17:40:32.877933 miner/unconfirmed.go:83] ⛏ mined potential block #38 [f935a859…], waiting for 5 blocks to confirm
I0313 17:40:32.878442 miner/worker.go:514] commit new work on block 39 with 0 txs & 0 uncles. Took 156.593µs
I0313 17:40:32.917608 internal/ethapi/api.go:1076] Tx(0xb739b2034e286712c8d32be44bd0bf296b95c2c97ce6d78671b11bfc106cdd22) to: 0xf9d50d8ccf1dbbaa5a77af7f3aa83d9b03d89ec8
I0313 17:40:33.395692 internal/ethapi/api.go:1076] Tx(0xadfa54c9a2048ce675e942cff4204f8101705bb9fcfcc9e8703aecb39912155c) to: 0xf9d50d8ccf1dbbaa5a77af7f3aa83d9b03d89ec8
I0313 17:40:33.841729 internal/ethapi/api.go:1076] Tx(0x7fb62c27aaba58e7d0d461fd4ef3773417ffb15a6785857d6fcacf0fedf5f759) to: 0xf9d50d8ccf1dbbaa5a77af7f3aa83d9b03d89ec8
I0313 17:40:34.181957 internal/ethapi/api.go:1076] Tx(0x245ad801ff33e32b54fac5b16be2f8da15aad52ec7ba1928dc61b90d07526f7d) to: 0xf9d50d8ccf1dbbaa5a77af7f3aa83d9b03d89ec8
I0313 17:40:34.547914 internal/ethapi/api.go:1076] Tx(0x2beba0073e9f73467881bdd8b77faf6d248c03e492253cc1b57eb2b49a9dfc1c) to: 0xf9d50d8ccf1dbbaa5a77af7f3aa83d9b03d89ec8
I0313 17:40:34.880430 internal/ethapi/api.go:1076] Tx(0x3d60cc7b47c14bf7f5cc580848efdc3721baa2ae9c7a349ad6ed27acaca020f9) to: 0xf9d50d8ccf1dbbaa5a77af7f3aa83d9b03d89ec8
I0313 17:40:35.244666 miner/unconfirmed.go:105] 🔗 mined block #34 [e0db658f…] reached canonical chain
I0313 17:40:35.244683 miner/unconfirmed.go:83] ⛏ mined potential block #39 [f8415f24…], waiting for 5 blocks to confirm
I0313 17:40:35.247679 miner/worker.go:514] commit new work on block 40 with 6 txs & 0 uncles. Took 2.97098ms
I0313 17:40:35.341533 internal/ethapi/api.go:1076] Tx(0xe538d180ef104ee60b5d7575d6547b2e2881adf35f67c4abcbfd9908fc7ce8e1) to: 0xf9d50d8ccf1dbbaa5a77af7f3aa83d9b03d89ec8
I0313 17:40:36.373287 miner/unconfirmed.go:105] 🔗 mined block #35 [2c5f80e6…] reached canonical chain
I0313 17:40:36.373308 miner/unconfirmed.go:83] ⛏ mined potential block #40 [5867ae2d…], waiting for 5 blocks to confirm
```

Figure 9: Snapshot of mining console on Go Ethereum

Dagger Hashimoto algorithm specifications

This algorithm is a combination of two different algorithms namely, *Dagger* and *Hashimoto*, invented by Vitalik Buterin and Thaddeus Dryja respectively.

Dagger Hashimoto uses a custom generated DAG data set, which gets updated at frequent intervals based on the blockchain metadata. Currently, the DAG is regenerated after the creation of every 30,000 blocks. The DAG is represented as a two dimensional array or a matrix with the dimensions of n * 16, where n is a large number greater than 16777186. The DAG has a depth of ten including the root level, and can have 2^{25}-1 values in total. For the first eight levels, every node in the DAG must have three parents that affect the value of the node. If the number of nodes at a given depth is N, then the number of nodes in the next level must be eight times larger than that i.e., 8N. For the ninth level, the value of each node in the DAG is dependent on sixteen parents and the number of nodes in the ninth level is two times that of the eighth level. The algorithm then pseudorandomly selects eight nodes, adds the nonce value to the block headers and computes a hash out of it. If the miner finds a nonce value less than 2^{256} divided by the difficulty amount, the PoW is solved and a block is generated.

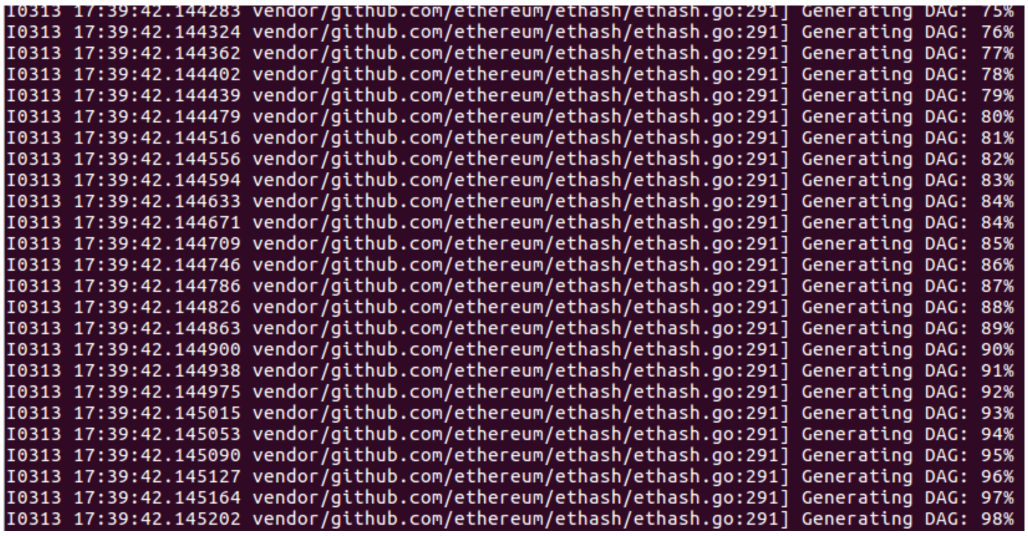
The following snippet, as defined in [But13], illustrates the algorithm. In the code below, D depicts the block header value, N the nonce value, L is the depth and spread(L) refers to the number of parents a node must have depending on the depth, and || is the concatenation operator. The objective of the code here, is to find the value of k, such that the value of the function eval(D,k) is less than 2^{256} divided by the difficulty value.

```
spread(L) = 16
if L == 9
else 3
```

```
node(D, xn, 0, 0) = D
node(D, xn, L, i) =
  with m = spread(L)
p[k] = SHA256(D || xn || L || i || k) mod 8 ^ (L - 1) for k in [0...m - 1]
SHA256(node(L - 1, p[0]) || node(L - 1, p[1])... || node(L - 1, p[m - 1]))

eval(D, N) =
  with xn = floor(n / 2 ^ 26)
p[k] = SHA256(D || xn || i || k) mod 8 ^ 8 * 2
for k in [0...3]
SHA256(node(D, xn, 9, p[0]) || node(D, xn, 9, p[1])... || node(D, xn, 9, p[3]))
```

Figure 10 shows a snapshot of DAG generation on the mining console logs of Go language implementation of Ethereum.



```
I0313 17:39:42.144283 vendor/github.com/ethereum/ethash/ethash.go:291] Generating DAG: 75%
I0313 17:39:42.144324 vendor/github.com/ethereum/ethash/ethash.go:291] Generating DAG: 76%
I0313 17:39:42.144362 vendor/github.com/ethereum/ethash/ethash.go:291] Generating DAG: 77%
I0313 17:39:42.144402 vendor/github.com/ethereum/ethash/ethash.go:291] Generating DAG: 78%
I0313 17:39:42.144439 vendor/github.com/ethereum/ethash/ethash.go:291] Generating DAG: 79%
I0313 17:39:42.144479 vendor/github.com/ethereum/ethash/ethash.go:291] Generating DAG: 80%
I0313 17:39:42.144516 vendor/github.com/ethereum/ethash/ethash.go:291] Generating DAG: 81%
I0313 17:39:42.144556 vendor/github.com/ethereum/ethash/ethash.go:291] Generating DAG: 82%
I0313 17:39:42.144594 vendor/github.com/ethereum/ethash/ethash.go:291] Generating DAG: 83%
I0313 17:39:42.144633 vendor/github.com/ethereum/ethash/ethash.go:291] Generating DAG: 84%
I0313 17:39:42.144671 vendor/github.com/ethereum/ethash/ethash.go:291] Generating DAG: 84%
I0313 17:39:42.144709 vendor/github.com/ethereum/ethash/ethash.go:291] Generating DAG: 85%
I0313 17:39:42.144746 vendor/github.com/ethereum/ethash/ethash.go:291] Generating DAG: 86%
I0313 17:39:42.144786 vendor/github.com/ethereum/ethash/ethash.go:291] Generating DAG: 87%
I0313 17:39:42.144826 vendor/github.com/ethereum/ethash/ethash.go:291] Generating DAG: 88%
I0313 17:39:42.144863 vendor/github.com/ethereum/ethash/ethash.go:291] Generating DAG: 89%
I0313 17:39:42.144900 vendor/github.com/ethereum/ethash/ethash.go:291] Generating DAG: 90%
I0313 17:39:42.144938 vendor/github.com/ethereum/ethash/ethash.go:291] Generating DAG: 91%
I0313 17:39:42.144975 vendor/github.com/ethereum/ethash/ethash.go:291] Generating DAG: 92%
I0313 17:39:42.145015 vendor/github.com/ethereum/ethash/ethash.go:291] Generating DAG: 93%
I0313 17:39:42.145053 vendor/github.com/ethereum/ethash/ethash.go:291] Generating DAG: 94%
I0313 17:39:42.145090 vendor/github.com/ethereum/ethash/ethash.go:291] Generating DAG: 95%
I0313 17:39:42.145127 vendor/github.com/ethereum/ethash/ethash.go:291] Generating DAG: 96%
I0313 17:39:42.145164 vendor/github.com/ethereum/ethash/ethash.go:291] Generating DAG: 97%
I0313 17:39:42.145202 vendor/github.com/ethereum/ethash/ethash.go:291] Generating DAG: 98%
```

Figure 10: Snapshot of DAG generation in mining console on Go Ethereum

4.3 Main differences between Bitcoin and Ethereum

While both Bitcoin and Ethereum are based on the same fundamental idea of cryptography and having a distributed ledger, there are distinct differences in their technicalities and the applications they are meant for. Both use a stack based languages, but Ethereum has the advantage of also having a turing complete programming language, as discussed earlier in Section 4.2. The consensus and block approval time in case of Ethereum is about 15 to 17 seconds, which is much lower than that of Bitcoin (10 minutes). The basic build of the proof of work algorithms that Ethereum uses is a memory hard hashing algorithm called Dagger-Hashimoto, while Bitcoin makes use of the SHA-256 hash algorithm. The economic models of the two are also slightly different. The Bitcoin miners have their block creation rewards halved every four years, whereas in case of Ethereum there is no such rule of reward reduction. The way the transaction fees are levied are also very different. Ethereum uses the concept of Gas based on the computational complexity of the operations performed, and has a maximum value per block. For Bitcoins, the transaction fees is limited by the block size.

However, from a general outlook, the applications and use cases Bitcoin and Ethereum were built for are very different. The idea of Bitcoin was to replace real money with cryptocurrencies without having to depend on central authorities to trust on, such as banks and governments. Thus, Bitcoin was meant to be a transaction channel and for the purpose of storing values. Ethereum, on the other hand, was meant to act as a platform to build decentralized applications and contracts using its own currency as a medium. The primary motivation for Ethereum was therefore, to be a facilitator for the developers to build their own DApps.

4.4 Other Implementations

In the big picture, it can be observed that there are two main methods a blockchain system can be built. The first being building a blockchain system by forking an existing system, and the second, building an independent system from scratch. Bitcoin and Ethereum are examples of independent systems built from scratch. Building a system from scratch is naturally pretty difficult to implement, despite some companies such as Namecoin having had reasonable success with it. The drawback such a system could have, would include communication issues between two or more applications having independent blockchains of their own. Systems that fork an existing independent blockchain system with alternative consensus rules are known as *Alt-coins* or alternative coins.

Using the first method described above, building a system on top of Bitcoin has the disadvantage of not being able to inherit the SPV system that Bitcoin uses. The current systems built on top of Bitcoin that have lightweight clients rely on a trusted server to provide them the blockchain data, which sort of defeats the purpose of having a trustless decentralised blockchain. There are also systems that are built on top of Ethereum, and target specific business needs.

4.4.1 Systems built on the Bitcoin blockchain

This section illustrates some of the blockchain implementations that were built as a fork of Bitcoin.

Namecoin

Namecoin [KCE15] is the first fork of Bitcoin and uses the same proof of work algorithm. The cryptocurrency system has the capability to store data in its own blockchain transaction database. The function of Namecoin can be best described as being a name register, or the equivalent of a DNS. They also have a domain name *.bit*, but it is independent of ICANN, the governing body of domain names.

In decentralised protocols such as Bitcoin and Ethereum, the accounts have a hash like identifier such as '1AW29dp2ZCbqW5CiBCrhQYtHagUWy'. Such identifiers, though are advantageous for the purpose of being obscure enough to not be impersonated, are difficult to interact with. Namecoin solves the problem by introducing the decentralised domain system and a first to file paradigm to prevent impersonation, where the first person who registers the name succeeds and the others fail.

Colored coins

Colored coins[ABH12], serves as a protocol to let people create their own cryptocurrencies, or even digital tokens on the Bitcoin blockchain. A new cryptocurrency can be issued by assigning a specific Bitcoin UTXO to a specific colour, and the users of that particular cryptocurrency would then maintain wallets containing UTXO outputs of a certain colour. They also have some special rules in case of mixed colour inputs and outputs. Simply put, color in this case is an attribute of a certain cryptocurrency, as well as its further applications. For example, certain colors could be meant for proving ownership of a car.

Metacoins

Metacoin[Bit16] is an extension on top of the Bitcoin blockchain and therefore does not inherit the SPV characteristics of the Bitcoin blockchain. Doing this provides a relatively easy to implement and low development cost process to create a consensus protocol with a possibility of having advanced features that cannot be implemented in the Bitcoin framework itself. For example, a small Bitcoin can be "labelled" as an ounce of gold, or a share in a company or even be counted as a score value in a video game.

Cryptonite - Mini Blockchain

Cryptonite is the first implementation of the lightweight *Mini-Blockchain* scheme[J.D14]. Their main intention of developing this cryptocurrency was to improve the scalability problem faced by Bitcoins. The mini blockchain scheme includes a self contained balance sheet system thereby eliminating the need to store transaction data perpetually. As a result, the maximum block size can be made larger. Even though the system has the advantage of being faster and less expensive, the drawbacks include a weakened security and less flexibility.

MultiChain

MultiChain [Gre15], a fork of Bitcoin, is a customizable and private blockchain service. Their main motivation behind building this software was to cater to large financial institutions that were not interested in using public blockchains. The multichain system also uses a randomised round robin method to add blocks onto the blockchain, and has a configurable mining process.

Rootstock

Rootstock[Ler15], an open source platform, is very similar to Ethereum in terms of creating smart contracts on a Turing complete smart platform, except that it utilizes the Bitcoin ecosystem to do so. The advantages to this platform are that it exists as a Bitcoin *sidechain*, and is backward compatible with the Ethereum virtual machine. This means that all Ethereum contracts can easily run on Rootstock. Their biggest advantage, however, is the fact that they can be merged mined with Bitcoin, thereby making it as secure. A **sidechain**[BE15] is a separate blockchain, whose assets can be transferred to and from the main blockchain, i.e. the Bitcoin blockchain in this context. This is a promising platform because it proposes Ethereum with a Bitcoin like security.

4.4.2 Systems built on the Ethereum blockchain

This section provides a brief description of some blockchain implementations forked from Ethereum.

Eris

Eris industries (now called Monax Industries) is a blockchain application platform that caters to financial institutions and helps them deploy private blockchains for their needs. They have forked Ethereum and also employ other technologies and wrapper tools that helped them become blockchain technology agnostic. They also provide service tools around Ethereum Virtual Environment.

Hydrachain

Hydrachain [EB15] is an open source platform built on top of Ethereum, and adds support for creating scalable permissioned (private or consortium) blockchains. Their main distinguishable features are the introduction of *accountable validators* and *instant finality* for private blockchains. Accountable validator refers to to a set of registered that propose and validate the order of transactions in the hydrachain distributed ledger. They use a byzantine fault tolerant consensus protocol that does not work with the proof of work algorithm. [CML99] and [Vuk15] explain the practical byzantine fault tolerance protocol that Hydrachain follows, and how it does not work with PoW. There needs to be a quorum by the validators that sign the blocks, before the block is added to the blockchain. This allows for the block time to be less than a second.

4.4.3 Independent blockchain systems

This section gives a brief description of Blockchain implementations built from scratch.

Hyperledger

Hyperledger[Hyp16] is a project hosted by the Linux Foundation as a cross industry collaborative project. The system was designed with the enterprise architecture in mind with customizable networking rules that help different consensus protocols operate. It borrows the UTXO and script based logic from Bitcoins and uses a byzantine fault tolerant consensus protocol instead of the proof of work algorithm. In general, it most likely is sort of an umbrella system that encompasses most possible use cases to suit the blockchain industry.

Lisk

Lisk [KB16] is a platform for decentralized applications and sidechains programmed in Javascript, and have a block time of approximately 10 seconds. They use DPoS as their consensus mechanism. They also have their own academy videos[Fer16] that teaches developers how to get acquainted with their system.

Their main idea is that every blockchain app, or the DApp equivalent of Ethereum, has its own sidechain, which helps with the scalability issues that the Bitcoin and Ethereum blockchains have been facing.

System Name	**Underlying System**	**Salient Features**
Namecoin	Bitcoin	Decentralised domain system.
Colored coins		A digital asset protocol that can customise the asset to suit specific needs.
Metacoins		A digital asset protocol that adds functionality to the Bitcoin protocol.
Cryptonite		A mini blockchain scheme with its own balance calculator.
Multichain		A private blockchain system for financial institutions.
Rootstock		A smart contract platform based on Bitcoins.
Eris	Ethereum	A smart contract platform coupled with service tools for ethereum virtual machine.
Hydrachain		A smart contract platform with accountable validators and instant finality for private blockchains.
Hyperleger	Independent	An umbrella blockchain system to facilitate maximum possible use cases for the industry.
Lisk		A decentralised application platform that makes use of sidechains for each decentralised app created.

Table 5: Overview: Some blockchain implementations discussed above

5 Management of Internet of Things Devices

Ericsson estimates the number of connected IoT devices to be upto 28 billion by 2021[4]. With the ever increasing array of communication protocols between IoT devices, there is a requirement for a transparent, yet highly secure and reliable IoT device management system. Other issues include tracking these billions of devices as well as storing the metadata they produce.

Since this thesis involves creating a device management system for constrained IoT devices, this chapter attempts to give an overview of various existing protocols and standards, and some implementations that deal with constrained devices and their management. Furthermore, the last section in this chapter attempts to introduce what it means to use blockchains with IoT and how it can be done.

5.1 IoT Protocols and Standards

The term Internet of Things (IoT) was coined by Kevin Ashton in 1999, while presenting at Proctor & Gamble. He proposed it as a link between radio frequency identification and the Internet[Ash09]. Semantically, it refers to identifiable networked objects or things, also called devices, virtual representations of which can be connected to each other over the internet. In other words, IoT refers to a wide variety of devices, including smart devices, systems and constrained devices, that interact and automate processes in a networked society.

Due to so much potential in the IoT field and the growing number of devices each year, solutions are being implemented in many areas, like supply chain, farming, healthcare, home automation to name a few. But with the development, people have come to realise the limitations that come from trying to make an overwhelmingly large amount of devices, belonging to different corporations and communities, communicate with each other.

Constrained Devices

Resource constrained devices or just constrained device, as mentioned in the IETF standard [BEK14], are devices that run on very limited processing, power and storage capabilities. Such networks display many constraints such as limited bandwidth

[4]Ericsson's prediction on their annual report: http://www.ericsson.com/res/investors/docs/2015/ericsson-annual-report-2015-en.pdf

or lossy channels. Continuous data retrieval from such devices is very challenging owing to the tight limits on available power, processing resources and memory. This thesis mainly makes use of CoAP as the application layer protocol, which we will discuss about in the subsequent sections.

5.2 Constrained Application Protocol (CoAP)

CoAP [SHB14] is a lightweight User Datagram Protocol (UDP) based standardised RESTful application layer protocol, for *extremely* resource constrained devices in machine to machine(M2M)[5] networks. It is designed to easily interface with HTTP for integration with the Web while meeting specialized requirements such as multicast support, very low overhead, and simplicity for constrained environments [SHB14]. It also widens the horizons in terms of solving M2M specific problems such as asynchronoous transfer, or reduction of message overhead, or resource discovery of constrained nodes. Common applications for CoAP include smart energy and smart building automation.

5.3 Constrained Objects Language and CoAP Management Interface

The combination of Constrained Objects Language(CoOl) [VPS16] and CoAP Management Interface (CoMI) [SBV17] is a CoAP based management protocol that uses Yet Another Next Generation (YANG) [Bjo10] modelling language to define its resources. Thus, CoMI and CoOL are adapted to low power, lossy constrained networked devices. Payloads for the information exchange are encoded using Concise Binary Object Representation (CBOR) [BoH16] data format, which facilitates extremely small code and message size.

The protocol is based on a client-server model. The client requests for the datastore resources and the event stream resources. The server is the provider of these resources. Figure 11 illustrates the protocol architecture. As per the standard, the implementers are free to choose the appropriate form of transport pertaining to the target applications.

[5]M2M communications is used for automated data transmission and measurement between devices

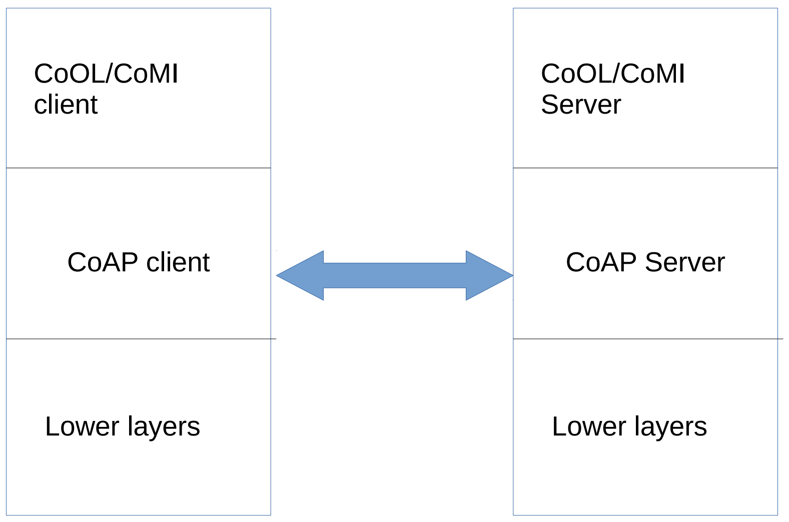


Figure 11: CoOL/CoMI architecture overview

5.4 Lightweight M2M (LWM2M)

OMA Lightweight M2M is an important protocol from Open Mobile Alliance (OMA), for the management of constrained devices. It usually runs on top of CoAP, and therefore is compatible with any device working with CoAP as its transport protocol. The main function of the LWM2M protocol, is to provide a set of interfaces that can monitor and manage constrained devices. The aim of this standard is to facilitate a fast deployable client server system serving M2M services.

LWM2M architecture defines three main parts, a *Bootstrap Server, a Client and a Server*. An LWM2M **Bootstrap Server** is a configuration server for the LWM2M client before it is connected to an LWM2M server. It essentially helps the server manage access control, keying and configuration of a client. LWM2M **Clients** are usually constrained devices, and contain many LWM2M objects and their corresponding resources. The task of an LWM2M **Server** is to manage the clients through management commands e.g. execute read, update or delete commands for the resources of a particular LWM2M client. The bootstrap server is supposed to configure the access controls to connect a server to a client.

The Open Mobile Alliance (OMA) specification [6] lists four logical interfaces for the

[6]OMA specifications: http://openmobilealliance.org/

server and the client to communicate with each other. They are as follows.

- **Bootstrap** The client sends a *request* message to the bootstrap server, and the server then accordingly performs a write and/or delete operation on the client, in order to register or remove LWM2M servers.
- **Client Registration** A LWM2M client registers to one or more LWM2M servers after it has been bootstrapped.
- **Device Management and Service Enablement** An LWM2M server can send various commands to the client in order to perform actions on the resources owned by those clients. The access control object, set during the bootstrapping process, determines the policies or the set of actions the server can take on a particular client's resources.
- **Information Reporting** Clients can report information in the form of notifications to the corresponding servers. This can be done due to the *Observe-notify* [Har15] feature from CoAP.

An LWM2M server can carry out one or more operations, in order to manage the clients and their corresponding resources. The resources of LWM2M clients are grouped into various objects. The format of these objects is specified by the IP Smart Objects (IPSO) Alliance [7].

Figure 12 depicts the architecture of LWM2M protocol.

[7]IPSO Alliance: http://www.ipso-alliance.org/

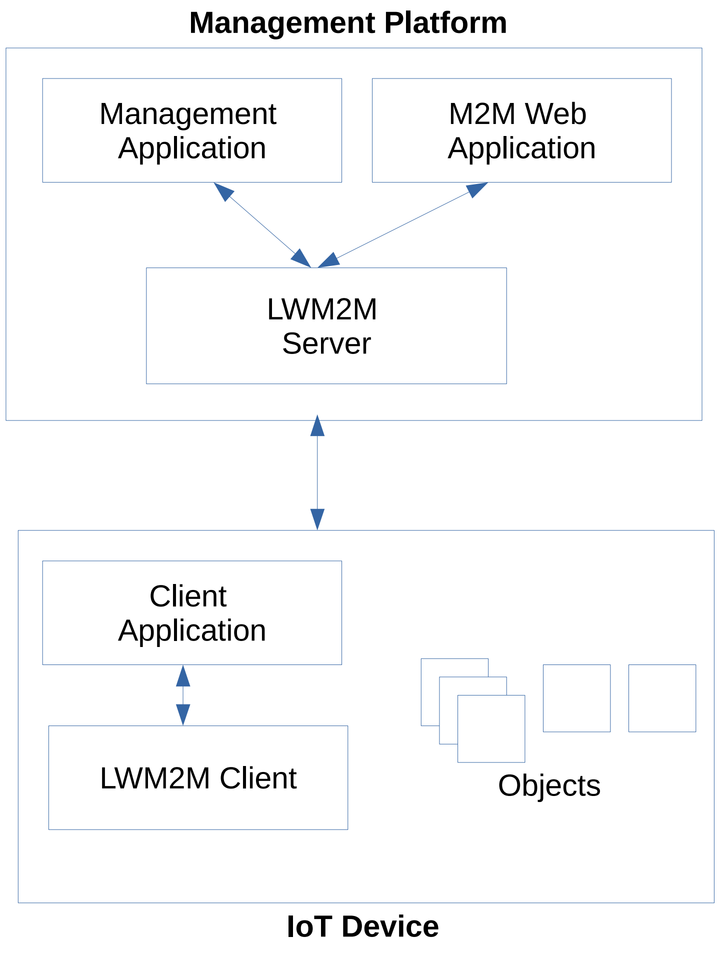


Figure 12: Architecture of Lightweight M2M

5.4.1 Device Management Implementations for IoT using LWM2M

Eclipse Leshan

Leshan[8] is an open source Java implementation of OMA LWM2M server and client. The project helps develop a customisable LWM2M client and server, and already comes with a demo bootstrap server, LWM2M server and LWM2M client.

[8]Leshan project: https://github.com/eclipse/leshan

Eclipse Wakaama

Wakaama [9] is a C language implementation of the OMA LWM2M server and clients. The project also provides some example demo applications that test the bootstrap server, LWM2M server and client capabilities of Wakaama.

5.5 IoT on Blockchains

From a generic perspective, it can be said that a blockchain is an immutable ledger of transactions. Through this, millions of IoT devices can be tracked thereby enabling secure co-ordination and communication between devices. The fact that blockchain is tamper proof and operates in a trustless environment, makes it extremely difficult to manipulate.

One way to use IoT on blockchains, would be to consider each IoT device as a blockchain node. Since the use of consensus algorithms in blockchains enable it to operate in a trustless environment, the IoT devices would not need to trust each other inherently. The devices can then co-ordinate with each other through a common network, despite having been produced by different manufacturers, or having incompatible softwares. If this can be done at a global level, then it would result in a web based user centric behaviour, as opposed to the current behaviour centred mainly around devices. But sometimes when IoT devices are constrained or incapable of storing a copy of blockchain, depending on their capacity, it is possible to either use them as lightweight clients, or have a higher capacity device as their blockchain representative node.

This kind of an arrangement could lead to several use cases that are hard to acheive at the current levels of interoperability between IoT devices. Some examples include blockchain acting as a "phone-book" to let machines find each other[10], creating a verifiable record of the usage of industrial medical devices[11] or a washing machine becoming a semi-autonomous device capable of managing its own supply of detergents and other parts[12].

[9]Wakaama project: https://github.com/eclipse/wakaama
[10]Implemented by onename: https://onename.com/
[11]Implemented by tierion: https://tierion.com/
[12]Prototype implemented by IBM

6 Design and Implementation of Proof of Concept

This chapter describes the small scale prototype that was developed for the management of constrained IoT devices. Section 6.1 describes why a certain implementation of Blockchain was chosen to implement the prototype. Section 6.2 describes the architecture and implementation of the system.

6.1 Rationale behind choice of technology

To be able to create a peer to peer distributed system in a trustless environment and still guarantee security, were of paramount significance for the prototype. Other factors such as ease of implementation through developer communities, and good documentation, were also considered while choosing the technology for the proof of work.

The creators of Bitcoin purposefully did not want to have loops in their scripting system in order to make it deterministic. That makes it easier to know exactly how a particular program ends. But for this project, it was important that we make use of loops so that we can store pertinent data in certain data structures and then iterate through the data. Therefore, Bitcoin was not chosen due to the absence of support for loops, and a very rudimentary scripting language[13]. Ethereum, on the other hand, uses a Turing complete language for development of decentralised applications.

Bitcoin also has relatively limited storage capabilities and is intended only as a cryptocurrency rather than being a decentralised application platform. The only way data can be transferred between two Bitcoin accounts at the moment, is through transactions. The data passed in such circumstances is public and therefore, is at a risk.

Another advantage of Ethereum that was taken into account, while choosing the technology to work with, was that the Bitcoin blockchain has a hard limit of 1 megabyte for a block size, whereas the Ethereum blockchain does not. However, Ethereum does have a limit to how much total gas can be spent per block, which is usually 1.2 times the exponential moving average. The main problem with having smaller blocks is that it tends to clog the network with more smaller packets. The gas limit on the Ethereum was more convenient to handle in comparison with the

[13]Bitcoin Script: https://en.bitcoin.it/wiki/Script

hard limit of 1 MB on Bitcoin blocks.

6.2 Architecture

The prototype that has been implemented for discussion in the thesis was designed to mitigate the issues associated with management of a large number of constrained IoT devices.

This part presents an overview of how the system is designed to work, high level architectural design and implementation of a blockchain based access management system for IoT devices. The basic application design is illustrated in Figure 13.

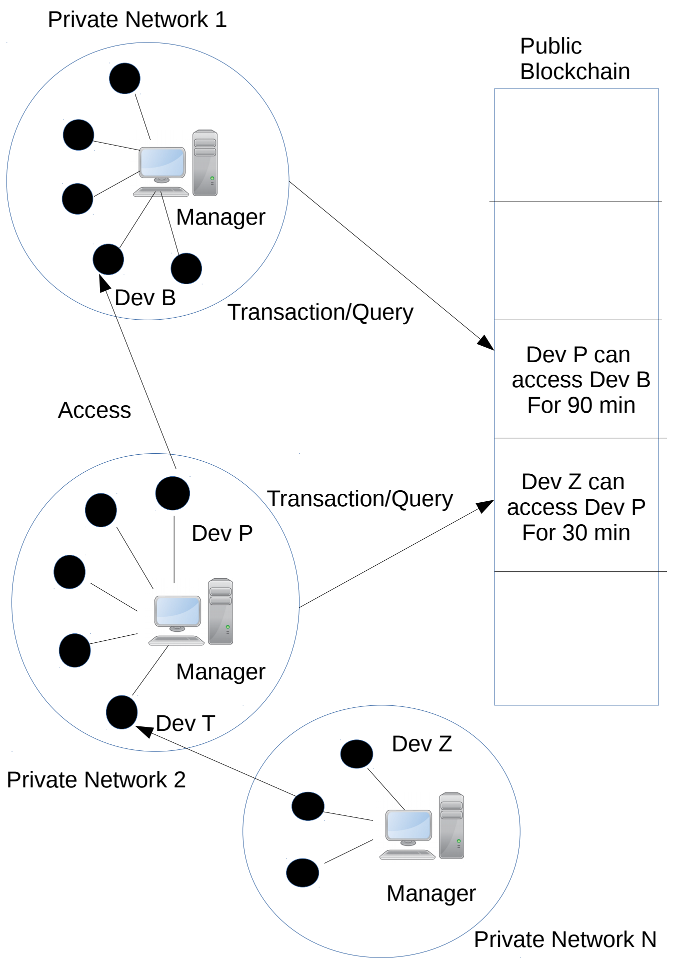


Figure 13: Overview of the system design

Since the IoT devices are largely constrained, they do not have the ability to store a copy of the blockchain in their system, thereby not participating in the blockchain network themselves. They instead, have an administrator node connected to the network, that we call *Managers*. The second set of actors, called *Clients* are nodes that do not have the rights to manage a certain device but want to query about their states. Figure 14 presents the use case diagram showing the roles played by the actors involved in the system.

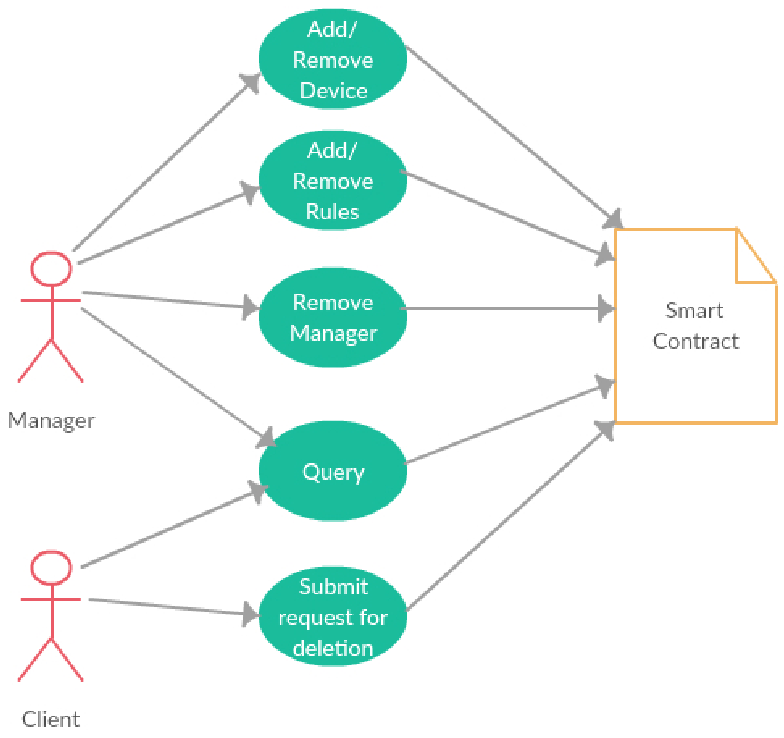


Figure 14: Use case diagram for the actors

The system makes use of the Ethereum blockchain with a custom genesis block. Ethereum nodes support a node discovery feature, also called the discovery protocol. In this protocol, the nodes continuously try to find peer nodes to connect to, until the criteria of the number of peers required, as specified by the *–maxpeers* flag, is met. Another way for the peers to find each other, is by manually adding the identity of the corresponding peer nodes. In the latter case, the nodes that are manually connected are called *Static* nodes. In this system, we use the static connection to avoid unpredictable situations.

Figure 13 illustrates different managers managing their own sets of constrained devices, operating under their respective private networks and firewall configurations. Before registering the constrained devices to the blockchain, the manager adds itself as a *manager* on the blockchain, by broadcasting a registration *transaction*. Upon verification of this transaction by the miners, it is added to the blockchain. The constrained devices then send an identity registration message to their respective managers, and in turn are registered in the blockchain by the manager by the way of broadcasting a registration transaction. Now as shown in Figure 13, constrained device Dev P, from Private Network 2 needs to access certain resources of Dev B from Private Network 1. Dev B's manager sends a transaction to the blockchain announcing the rule "Dev P can access Dev B for 90 minutes". The miners check the transaction, and a rule is established on the blockchain. Dev P can now access Dev B.

6.2.1 Data flow in the system

The proof of concept created to address the characteristics of our system consists of four separate components namely, the contract creator, the client, the manager and the Ethereum smart contract. The functions of each component have been broadly illustrated in Table 6. Figure 11 illustrates the block diagram showing how the different components interact.

Application	**Function**
Contract creator	an API that compiles and deploys the smart contract to the blockchain
Client	queries the smart contract and submits requests for the Manager to approve
Manager	registers and deregisters device identity, adds and deletes rules corresponding to the device, and queries the smart contract
Smart Contract	includes the logic to store and remove all the data and permissions based on the role.

Table 6: Functions of the components

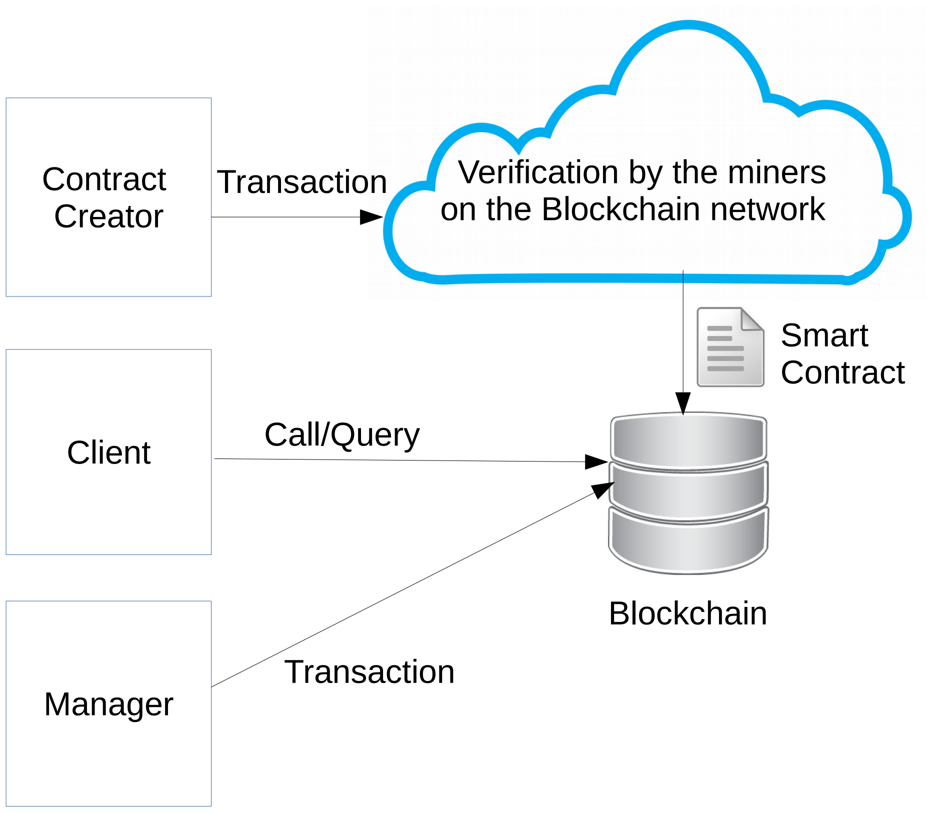


Figure 15: Overview of how the different components interact

6.2.2 Component: Contract Creator

This application consists of a back-end written in NodeJS[14]. It can be hosted on a webserver, or locally on a node that participates in the blockchain network. JavaScript was chosen because the application was intended to be an application programming interface (API) that can then be integrated with client. The main motivation behind developing this application was to automate the smart contract creation process.

The application takes the smart contract code as an input, compiles it and deploys it to the blockchain network. This deployment is broadcast as a *transaction* in the blockchain. The smart contract then goes through the validation process on the network and when correctly verified, gets stored into the blockchain forever. The detailed description of the functions used in the contract creator application have

[14]NodeJS: A server side JavaScript based framework. https://nodejs.org/en/

been described in *Appendix 1*.

6.2.3 Component: Manager

Managers are the main actors in the system. They have the ability to make changes to the data stored on the smart contract memory. Depending on the permissions, a manager can perform the following operations on the smart contract.

- Register and de-register (if needed) the identity of a constrained device to the blockchain
- Add and remove rules corresponding to the devices they manage, to the blockchain
- Query the blockchain

These operations are explained in detail in the smart contract subsection.

A manager is responsible for a certain set of devices, that it alone can register to the blockchain. It also adds access rules corresponding to the devices it manages, such as which constrained device can access a specific resource in a device it manages and for how long, as illustrated in Figure 13.

All managers host a personal copy of the blockchain i.e., they act as full nodes. They can store it in any kind of network accessible storage, privately hosted, on the cloud, decentralised storage such as Swarm[15] or IPFS[16] or other types of storage. For the sake of simplicity, it has been presumed that all managers also act as miners in the system. This ensures the safety of the system in case of a malicious manager, as will be discussed later in Section 7.4.1.

6.2.4 Component: Client

Clients include the managers that do not have access to a certain device, but want to query the system about the manager of a specific constrained device, or want to check whether one constrained device can access another one. The clients only have sort of a read-only access and do not have the permission to modify the system on

[15]Swarm: A decentralised content distribution platform. More information at http://swarm-gateways.net/bzz:/theswarm.eth/

[16]IPFS: A peer to peer distributed web storage system. More information at https://ipfs.io/

behalf of devices they do not manage. They can request for a rule to be deleted or a manager to be deleted, which is approved by the manager.

The front end or the client, shown in Figure 16, uses a JavaScript library called React[17] and is needed to trigger the JavaScript back-end.

The back-end is a JavaScript file representing the client application in the proof of concept. The client can be hosted on a webserver, or on a decentralised storage such as Swarm or IPFS, or locally on a server that participates in the blockchain network. It may be used by all the participants on the network to communicate with the smart contract, on successful contract creation and deployment by the *Contract Creator*. The client interacts with the smart contract memory through the *Web3* library of JavaScript and can do the following things based on the permissions.

- Query if a device or a rule exists in the blockchain
- Query if a device has a certain manager
- Submit a request to delete a rule or manager corresponding to a device

[17]React: A JavaScript library for building user interfaces. https://facebook.github.io/react/

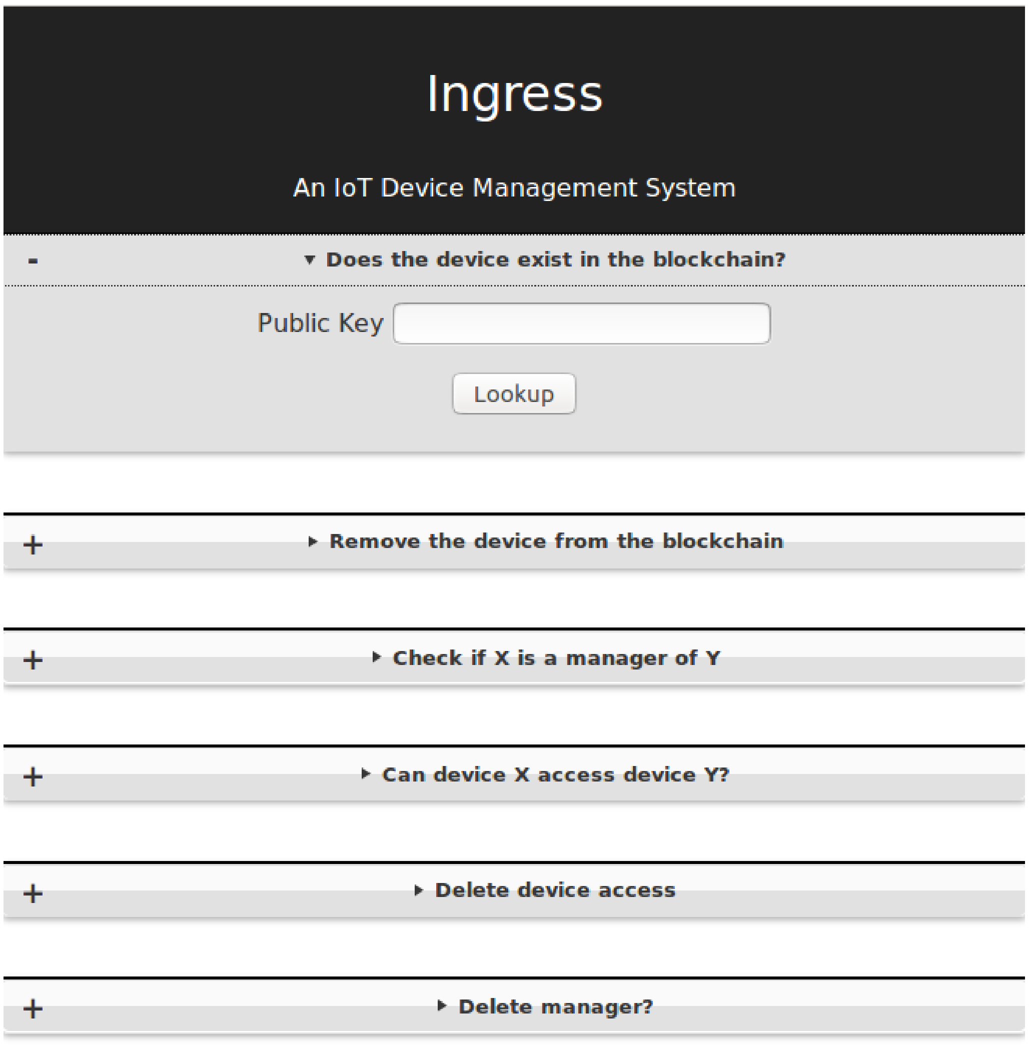

Ingress
An IoT Device Management System
- ▾ Does the device exist in the blockchain?
Public Key
Lookup
+ ▸ Remove the device from the blockchain
+ ▸ Check if X is a manager of Y
+ ▸ Can device X access device Y?
+ ▸ Delete device access
+ ▸ Delete manager?

Figure 16: The front end of the Client

A description of the libraries used by the Client JavaScript can be found in the *Appendix 3*.

6.2.5 Component: Smart Contract

As discussed earlier in Section 4.2.3, a smart contract is a distributed application that is a part of the blockchain, and is therefore hosted on all the nodes connected to the blockchain network. The smart contract has been coded in Solidity programming language. The Solidity files end with a ".sol" extension and need to be converted into the bytecode format for it to be understandable by the Ethereum Virtual Machine.

The Contract Creator component, described in Section 6.2.2, takes care of converting the solidity code into a format that can be read by the EVM and deploys it into the Ethereum blockchain.

On deployment, the smart contract is assigned an Ethereum contract address. If any node in the Ethereum network wants to communicate with the smart contract, they do so by passing messages to its address. The messages could be of the form of a *Transaction* or a *Call*. Transaction messages are sent when the nodes require to store information pertinent to identities or access information of a specific device. A call is made when nodes want to query the state of a device at a given time. The following paragraphs briefly explain some of the nuances of the Solidity programming language and the functions of the smart contract. Figure 17 shows the general architecture of the smart contract implemented for the prototype.

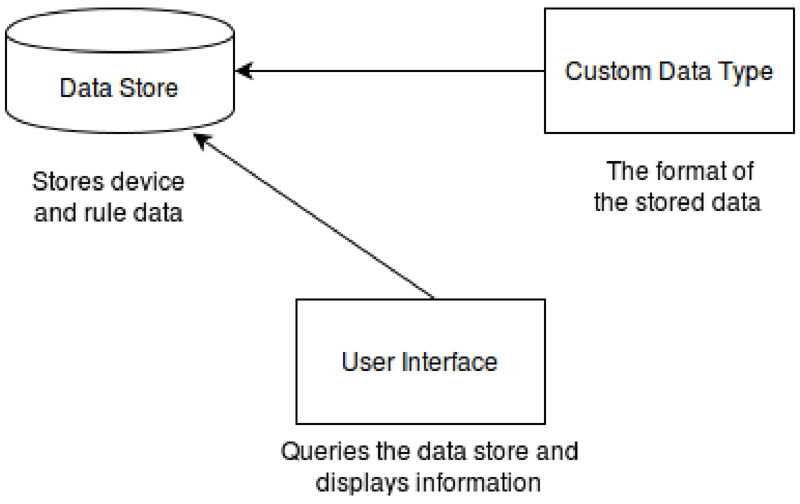

Data Store
Custom Data Type
Stores device and rule data
The format of the stored data
User Interface
Queries the data store and displays information

Figure 17: Simplified architecture of the smart contract

The implemented smart contract consists of a various functions corresponding to certain features. The Manager component has access to all the features in the smart contract, provided the actions performed by the manager are on behalf of the devices it manages. The features have been explained in the following paragraphs.

The custom data type refers to the 'Struct' data type defined in the solidity language. The struct data type facilitates grouping of different data types together in contiguous memory locations, thereby combining them to form a unified data type. In this case, the constrained device information, the manager information, and the rule details are stored in three different structs. When a registration transaction is made, the registration information goes through certain checks based on the logic of the smart contract, and is stored in the Data Store. Solidity uses a non iterable data structure called *Mappings* to store data. Mappings are data types that are declared as

```
mapping(Key_type=>Value_type)
```

Here the Key_type can be any low level data type in solidity. A Key_type cannot include a mapping, a struct, a contract, a dynamically sized array, or an enum. But the Value_type can include everything including other mappings. These Mappings may be described as hashtables, except for the fact that the key type is not stored in the mapping itself, rather the key type's hash is used to look up the value.

The smart contract has several functions, a description of which can be found in the *Appendix 2*. The subsequent sections provides an explanation of the main tasks implemented by the smart contract.

Device Identity Registration and De-registration

The identity registration operations register a management node or a constrained device into the system. This operation requires a public-key encryption system to verify the identity of the nodes before adding them into the blockchain. On verification of identities, the manager adds the devices it manages to the system and itself as its manager, which in turn generates a transaction, thereby storing the information to the blockchain upon verification by the miners. Both the operations consume a certain amount of *gas* and a transaction fee for the miners.

If needed, the manager can also de-register itself or a device registered to it. It may be noted however, that a manager can only de-register itself when it has no corresponding constrained devices to manage.

Adding and Revoking Rules

The rule operations facilitate adding or removing some secure rules into the blockchain. These rules involve granting or denying the access rights between two constrained devices. The manager adds or removes the rules corresponding to the devices it manages. These rules specify what resource a particular devices has access to, what type of access it has and for how long. Typically, the access types vary between read, write and both. A previously added rule can also be modified and updated.

The revocation of rules is also done by the manager, for the constrained devices it manages. Both addition and revocation of rules generate a transaction charging a certain amount of *gas* and transaction fee, which is then stored in the blockchain upon verification by the miners. A depiction of how a rule is added to the smart contract can be seen in Figure 18.

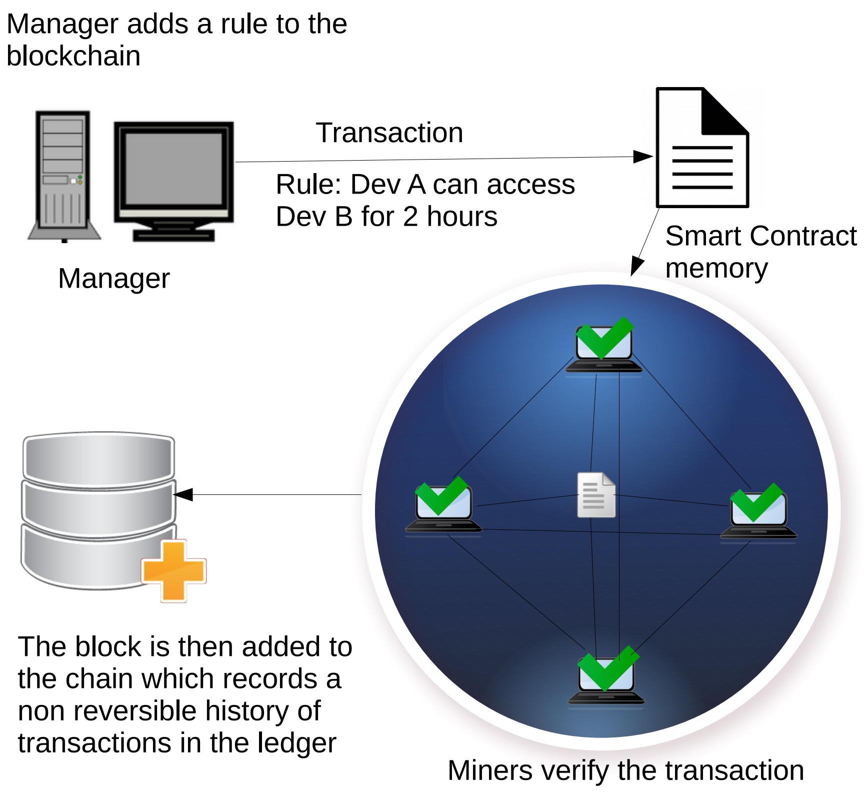

Manager adds a rule to the blockchain
Transaction
Rule: Dev A can access Dev B for 2 hours
Manager
Smart Contract memory
The block is then added to the chain which records a non reversible history of transactions in the ledger
Miners verify the transaction

Figure 18: Overview: How a rule is added to the smart contract

Querying the Blockchain

This operation queries whether one constrained device has access to certain resources in another device. Any node connected to the blockchain network can access this information in a read only manner. This operation is only meant to query the access information, and does not send out a transaction every time a client queries. This functionality in Ethereum is called *Call*. A Call request invokes a function in a smart contract and gets a return value. This operation therefore, does not require any *gas* or any sort of transaction fee, since it does not need to be mined.

7 Evaluation

This chapter presents the evaluation of the implemented proof of concept system. The evaluation primarily refers to the assessment of the system to better understand its capabilities, and to determine whether the system meets the expected requirements.

This chapter is organised as follows. Section 7.1 introduces some important software testing methodologies, used during and after development of the system. Section 7.2 briefly describes the test environment and provides information on the hardware capabilities and network conditions used for the development and evaluation. Section 7.3 lists the metrics and the result of the evaluation experiments. Section 7.4 analyses the security aspects of the system, and Section 6.5 briefly talks about the limitations of Ethereum Blockchains in general, as well as the proof of concept system.

7.1 Test Methodologies

Testing Methodologies are approaches used to test products to ensure they function as expected. There are many different testing methodologies depending on the stage of development. They can be broadly classified into *Functional* and *Non functional* testing methodologies.

7.1.1 Functional Testing

Functional Testing of a system refers to a kind of *black box testing* meant to examine the system's external workings, according to business requirements. Black box testing is the method of testing wherein the internal workings of the system are not required to be known. It typically involves identifying the features the system is expected to perform, and determining whether the outputs match the expectations. This thesis uses two main kinds of functional testing namely, system testing and acceptance testing. *System testing* includes testing the entire system for errors and bugs, and *Acceptance testing* involves making sure that all the functionalities work according to expectations.

To conduct a functional test on the smart contract, a development and testing tool called Truffle[18] was used. It has a built in smart contract compilation system coupled with an automated contract testing framework that made the unit testing relatively convenient.

7.1.2 Non Functional Testing

Non functional testing requires that the system be tested against defined technical qualities some of which include *Performance*, *Usability*, *Security* and *Vulnerability* testing. This chapter mainly focuses on the non functional aspects of testing since it is of more importance, in this thesis, to understand the internal workings of the system. In the subsequent sections, there will be more discussion about the non functional testing methods and an explanation of the results obtained as a result of these non functional tests.

7.2 Test Environment

7.2.1 Hardware

The test environment consists of a host machine and an Ethernet connection. The host machine is a server running a 64-bit Ubuntu Xenial 16.04.1 LTS operating system. The processor has 8 multithreaded cores and an Intel Core i7-4800MQ processor with a maximum clock speed of 3 GHz, and a RAM of 16 GB. Each client or miner is run as a separate virtual *Docker*[19] container on the host. A 240 GB Solid State Drive (SSD) was used to prevent storage disk's bottleneck issues. A 1 GbE connection has been used to evaluate the proof of concept. The network conditions are kept constant, and no external tools are used to modify the network latency or packet loss, as evaluation over different network situations is not included in the scope of the thesis.

7.2.2 Docker Containers

An open source software called Docker has been utilised to simulate a real world distributed system. Docker containers are a lightweight way to package a node with all the necessary runtime dependencies, and isolate it from the underlying

[18]Truffle github: https://github.com/ConsenSys/truffle
[19]Docker: https://www.docker.com/

infrastructure, i.e. the Linux kernel. Therefore, docker containers guarantee that the nodes always run in the same way.

Docker was chosen over virtual machines, because of its lightweight nature and faster boot time due to the isolation. It also allows a very effective memory and CPU resource allocation system that was very useful for the evaluation of the proof of concept.

7.2.3 Testbed configuration

This section explains an overview of the testbed used to evaluate the performance of the blockchain based system developed to manage access configurations for IoT devices.

The system uses a private Ethereum blockchain for evaluation purposes. There are two main types of nodes used in the configuration namely, a *bootnode* and *managers*. As discussed earlier, Ethereum nodes support a node discovery feature, wherein they look for other peers on the network to connect to. In this case, a node that acts as a connection facilitator for the rest of the network, called a bootnode, has been used to achieve the peer connection. In practice, the main public Ethereum network is also served by three bootnodes hard-coded into the Ethereum client. The test configuration also includes six miner nodes, also called managers. The purpose of the managers has already been discussed in Section 6.

7.3 Latency and Throughput Evaluation

Theoretically, Ethereum does not have any upper limit on the number of transactions that a block can accommodate. When a miner tries to fit in transactions worth more than the block gas limit, the block gas limit gradually adjusts itself. In practice, however, there are many factors that influence the throughput and the latency of a blockchain network. For example, clients that are big machines with plenty of bandwidth and processing power will have a higher threshold for throughput whereas smaller devices will have a lower throughput. This evaluation attempts to exhibit how the proof of concept system behaves when the CPU processing and memory are varied.

Evaluation Methodology and Results

It is imperative to understand the state of the system before, during and after the evaluation is conducted, since the results depend on it to a great extent. The evaluation set-up involved setting up six miners and a bootnode on Docker containers. The role of the bootnode was to aid in peer discovery for the miners. The bootnode dynamically generates an Ethereum address, which the miners connect to, in order to find each other. The official Ethereum image *ethereum/client-go*[20] was used on each Docker container.

The initial bandwidth between each pair of Docker containers was examined, to rule out those bottlenecks. The bandwidth, measured using the Linux tool called *iperf*[21], averaged out to 14.6 Gbits per second. The memory and CPU consumption of the physical machine as well as every container before and during the tests were also compared. A tool called *sysstat*[22] was used to measure the performances of each Docker container and the physical host at every stage. To ensure that unnecessary processes don't interfere with the evaluations, they were terminated.

After the private blockchain was connected, the CPU quota and the memory values of each Docker container was changed to take the readings. The difficulty level for block creation was initially kept low to see if it would adjust gradually. 60 transactions of the same size and type were pushed to the blockchain each time. This was done to keep the transaction variables constant, and to check the amount of time all the nodes would take to process the transactions and reach a consensus. Due to the sheer number of steps involved in getting the system up and running, bash and javascript scripts were used to automate the process.

[20]Ethereum Docker image: https://hub.docker.com/r/ethereum/client-go/

[21]iperf:https://iperf.fr/

[22]sysstat: System performance tool for Linux. For more information: https://github.com/sysstat/sysstat

		Time per 60 transactions (seconds)					
CPU Quota (%)	**RAM(GB)**	**Run1**	**Run2**	**Run3**	**Run4**	**Run5**	**Run6**
100	**16**	31	36	37	36	40	38
	8	73	69	85	71	69	75
	4	77	75	87	85	77	79
	2	150	144	157	139	149	132
	1	184	205	192	199	198	192
50	**16**	63	55	62	73	75	64
	8	97	86	87	75	102	81
	4	105	110	100	107	120	102
	2	183	163	182	171	178	174
	1	245	219	229	235	234	231

Table 7: Relationship between System Resources and Block Creation Time

Table 7 shows the readings obtained after the process described above was performed. The main purpose of making these readings was to observe the existence of a pattern and to try to visualize the identified pattern.

As can be seen in Table 7, varying the CPU levels makes a bigger difference than a 50% variation of the memory in the case of block creation time. Each *Run* in the table refers to the sample data collected every time an iteration was made. 6 samples were collected for every CPU Quota and memory configuration, and a total of 60 samples were collected. These were averaged out for every 60 transactions and a graph was plotted. Figure 19 provides an illustration of the readings found from the evaluation.

Each Docker container was allocated a specified amount of memory and CPU resources, and a set of 6 readings observed for each resource configuration. However, a point to be noted here is that the total memory capacity of the Server or the physical machine, is 16 GB. The RAM therefore, indicates a maximum amount that the particular Docker container can use, and there exists a possibility of memory contention on the machine. Since this was just an experimental set-up, the aforementioned limitations were recorded as constraints to the arrangement.

As previously mentioned in Section 4.2.4, mining in Ethereum is a memory as well as a CPU intensive process. The generation of the DAG dataset, in itself, takes up around 1 GB of memory and continues to increase linearly with the number of blocks in the blockchain. Therefore, in this case, the order in which the experiment was done also affects the amount of memory consumed.

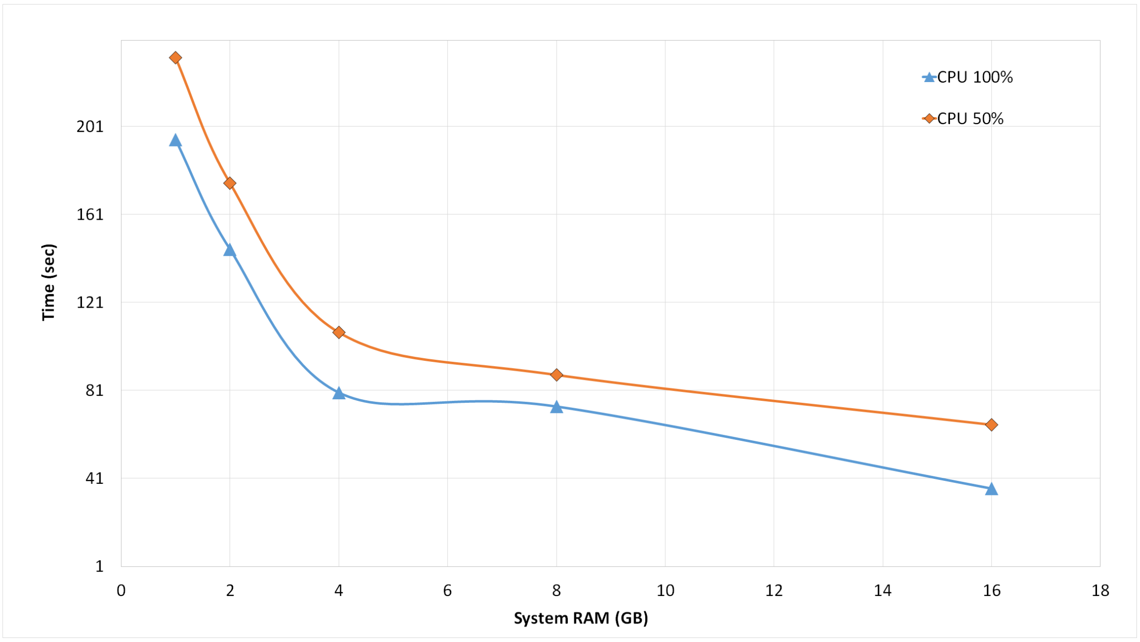


Figure 19: Effect of System resources on block creation time

It can be observed, from both Table 7 and Figure 19, that the time to create a block gradually increases when the resources are limited. The lowest configuration that was tested here had a memory of 1 GB with a 50% of CPU Quota of the original physical machine. One point to be noted here, is that every Docker container was run on one CPU core every time. This limited the resources to an even greater extent. For example, in this case, the lowest resource configuration device could be a mid range Android phone or a tablet. Thus, a simple mobile device could potentially be the miner in this instance. However, as the size of the blockchain increases, it would be more and more difficult for a mobile device to be able to store it. A solution to that could be to offload the data onto a cloud server. As the number of miners grow in the blockchain, the difficulty level for a block to be generated would increase as well. The increase in difficulty level would, in turn, lead to more investment of computing resources of the device.

7.4 Security Analysis

This section analyses some of the weaknesses associated with the security of blockchains in general as well as the proof of concept system.

7.4.1 Malicious Clients

A client that does not follow the rules of a given blockchain network is no more considered a peer by all other clients, and ends up making another parallel blockchain with its own rules. A more malicious client that is able to pretend to follow rules of the blockchain system might possibly be able to act without getting noticed.

In this case, there is a possibility that a manager, who also happens to be a miner, may go offline out of the blue, drop transactions, make wrong transactions, or even decline that a transaction ever happened. One way the system can circumvent this kind of behaviour is by using some more financial incentive for the managers to submit the correct information in all situations. As a miner, even if a manager refuses to verify transactions or declines that a transaction ever happened, the other mining nodes make up for it. The transaction can only be excluded if a majority, i.e. greater than 51 % of the miners decline to include it. In general, the managers are assumed to be parties that the constrained devices trust, presumably the owners of those devices. Therefore, there should be no reason for the managers to want to sabotage the devices. The main advantage of using blockchains, as far as device management is concerned, is to ensure the data integrity. This makes sure that the data already in the system has not been modified - neither intentionally nor by accident.

7.4.2 Malicious External Smart Contracts

In the case of smart contracts, any code that makes calls to an external smart contract is a potential security risk which has to be evaluated very carefully, specially if it includes some payment. The following paragraphs explain some known attacks that can happen while making use of external contracts.

Denial of Service Attack The depth of the stack meant for making calls through the Ethereum Virtual Machine is limited to 1024. If an attacker continues making recursive calls until the stack depth limit, they may be able to take control of the smart contract code. This is called as the *Call Depth Attack*[CoF17]. The attacker can also manipulate the amount of gas needed for a transaction and at the very least can make the gas required more than the block gas limit, thereby blocking any transaction from happening at all.

Reentrancy One of the major security hazards of invoking an external contract from any given smart contract S, is that the external smart contract can make changes to its data without S having any idea. In this attack, functions of S are called over and over even before the first function call is finished. This may cause the different invocations of the same function to interact with each other in destructive ways [CoF17]. A way to tackle this problem is to treat the function that invokes the untrusted smart contract as an untrusted function itself. Another way to approach this would be to use a *mutex* to lock a state in the function which can be unlocked by the owner of the lock alone.

7.4.3 Sybil Attack

Many distributed systems do not have any form of identity management other than having accounts. Due to this reason, any actor is capable of creating multiple accounts. A Sybil Attack [Dou02] manifests when one actor can manage to act as multiple entities at the same time. An attacker can try to fill the network with the clients it controls. In such a case, the attacker can refuse to relay certain blocks or only relay the ones created by the attacker.

A known solution to this attack is using a resource based counting mechanism, i.e., keeping a track of the accounts created per resource. Another approach to solving this can be to require the participants of the network to have a stake of a limited resource, such as cryptocurrency, so that they do not get any benefit out of pretending to be multiple entities in parallel. Manual authorization of each identity could also be a way, although tedious and inconvenient.

7.4.4 Timestamp Hacking

Each node maintains an internal counter representing the network time based on the median time of its peers. This information is exchanged between the nodes during peer connection. However, if the difference between the median time and the system time is greater than 70 minutes, the network counter time is set as the system time.

An attacker could potentially affect the network time if it chooses to, by connecting multiple peers and reporting the time wrongly. This can be fixed by changing the network time calculation methodology done by the nodes[BiF17].

7.5 Limitations of the system

The following paragraphs analyse some of the limitations of the proof of concept implementation.

7.5.1 Block creation time

One disadvantage of the implementation is the amount of time it can take to create a block. As per Ethereum implementation, it may take up to 12 seconds to create a block in a public blockchain. Every time a transaction takes place in the blockchain, it goes into the queue to be picked up by a miner. This means waiting for a miner to collect transactions and then generate a new block.

The block creation time issue is hard to avoid, given the nature of the Ethereum blockchain. This could pose a problem for the IoT devices in case of urgent access requirements. If the transaction difficulty is set low enough, the block creation time become relatively short. However, low difficulty could lead to multiple security issues as well as generation of more orphan and stale blocks.

7.5.2 Changes to the system

The transactions work based on the logic provided in the smart contract. If a change is to be made to the logic, or the way transactions are handled, the entire system has to be somehow set up from scratch. This would cost a lot of gas, not to mention other overheads such as the clients having to adapt to a new system. Another problem that arises from this is the access data getting lost, due to the smart contract system being set up all over again.

To remedy this problem, distributed databases such as Swarm[23] or IPFS[24] can be made use of. This would mean that the actual data would be stored in these distributed databases instead of the smart contract memory. These distributed

[23]Swarm: A decentralised content distribution platform. More information at http://swarm-gateways.net/bzz:/theswarm.eth/

[24]IPFS: A peer to peer distributed web storage system. More information at https://ipfs.io/

databases generate a hash based on the data stored in them. This hash, in turn, would be saved in the blockchain, thereby making it immutable.

Therefore, in other words, if there any changes are required to be made to the system, there can be two possible approaches. Either a new smart contract needs to be written with the new logic or, a new smart contract that manipulates the existing smart contract needs to be created.

7.5.3 Timestamp Dependence

The system uses *Block number* and *average block time* to schedule the expiry of a specific rule. This is not definite as the block times may change with different releases of Ethereum.

8 Discussion

The aim of the thesis was to investigate how to securely manage IoT devices in a more autonomous and distributed way, without having to trust a central entity. The questions the thesis attempts to answer are as follows.

1. *Is it possible to implement IoT device management, specifically in constrained devices, on blockchains, on a global level?*

   It has been found from the observations and experiments conducted during the course of this thesis, that Blockchain Technology has the potential to be utilised to authenticate device identity, protect and store data, handle access to devices, and process transactions in the operation of those devices, in a distributed way. The distributed replication model of blockchains facilitates accessing and supplying IoT information for customers as well as organisations, without requiring a centralized management server. The blockchain can also be used to securely make payments for services or use of device resources given pre-agreed conditions are tracked.

2. *What advantages does the use of blockchain bring to IoT?*

   Equal stakes Blockchain technology has already proven its worth in the financial industry through the use of cryptocurrencies such as bitcoins. This concept can be adapted to fit the needs of IoT networks, thereby allowing billions of devices to talk to each other over the same network without the need for additional resources. Blockchain also takes into account the issue of ownership and authority between various vendors by ensuring everybody involved has equal stakes.

   Transparency, Privacy, Non-repudation, Integrity The main advantage of using blockchains in IoT, is that it is public and transparent i.e., anybody participating in the blockchain network will have the chance to see all the transactions taking place and the blocks. At the same time, the complete transaction data is not visible. The involved parties in the transaction remain private since they transact under their respective accounts, and their personal information is protected by their private keys. This combination of non-repudiation and confidentiality helps improve the privacy aspect of blockchain transactions. Blockchain technology also incorporates hashing, which helps check the integrity of the information stored in the blockchain.

Decentralization, Trustless As blockchain is decentralized, there is no single central entity involved in approving transactions or set conditions for certain transactions to be accepted. This means that it is a highly trustless network, and requires consensus among all the participants in the network to accept transactions.

Immutable Most importantly, it is secure from the point of view that the data once entered cannot be changed, and may act as the single source of truth. This property of immutability is a major advantage of the blockchain technology. There is a possibility however, for a malicious node to try to change the data or deny that a transaction happened. But in such a case, the malicious miner would just be left behind, or desynchronise from the network and in turn would end up running its own tiny blockchain network which would be of no use to it.

Circular Economy Another advantage the use of blockchain can bring to IoT can be to help build a so called circular economy[25], where device resources can be shared instead of purchasing and disposing.

Data Tracking The ability to track the data exchange history corresponding to every single IoT device from the ledger of transactions, is also an important implication and is highly advantageous if used judiciously.

3. *What are the caveats and challenges of using blockchains for management processes in constrained devices, specially considered at a global perspective, and how can they be mitigated?*

   There are some shortcomings to using blockchains for IoT, but the solutions for them are either actively being worked on, or have already been solved at the time of writing the thesis.

[25]A circular economy is an economic model which focuses on reusing materials and value.

Storage Scalability One of them, is the issue of *Scalability*. There are different factors that could come under the umbrella term of scalability. It includes aspects such as bootstrap time, transaction costs, throughput, latency time [CDE16]. *Bootstrap time*, refers to the time taken by a new full node to completely synchronize with the blockchain. At present, Bitcoin has more than 100 GBs worth of data that new nodes might need to download and process in order to synchronize, and depending on the hardware specifications it takes an average of *four days* to do so. *Latency* refers to the maximum time taken to confirm a transaction, and *maximum throughput* refers to the maximum rate at which the confirmation of transactions can be done on the blockchain. *Transaction costs* comprise of the cost of **resources consumed** in order to have a confirmed transaction. This includes the digital currency paid as a transaction fee, as well as the physical currency paid for the operational costs of the hardware that acts as a node, bandwidth of the network and storage costs.

Energy consumption When a cryptocurrency is in its nascent stage, the resources required to mine the currency do not amount to much. But as the difficulty level of mining the cryptocurrency increases, the miners need to resort to much more powerful hardware, and as we know, the more operational efficiency a hardware has, more energy it consumes. With the growth in mining farms all around the world, blockchain mining has very little regard for sustainability.

Mitigation Strategies for Scalability

SPV nodes As discussed earlier, SPV nodes download a copy of the headers of all blocks instead of the entire blockchain. This means that the storage requirements scale linearly with time, the start time being when Bitcoins were created, in this case [Sat08].

Block Pruning As mentioned earlier in Section 2.1.6, the blockchain grows larger every 10 minutes. *Block pruning* removes transactions that are no longer needed to verify if the blockchain is in a consistent state, and hence, allows full nodes to free up storage space [Sat08]. In terms of scalability, pruned nodes scale in the same way as the number of UTXO, or unspent transaction outputs. This means that the storage will scale O(n) where n is the number of users instead of number of transactions.

Sharding The main idea of *sharding*[But17], is to split the memory address spaces of all the blockchain accounts into subspaces. For example, a sharding method can insert the accounts with addresses beginning with 0x00 into one shard, and addresses beginning with 0x01 into another one, and so on. Essentially, each shard has its own transaction history and the effects of state transitions corresponding to a particular shard is limited to that shard only. The effect of a transaction taking place in a particular shard though, will depend on the events taking place in the blockchain as a whole. Figure 20 explains sharding with a simple example.

Let us consider that account A, in shard X wants to transfer 100 digital coins to account B, in shard Y. In this case, the transaction taking place in shard Y depends on the events that took place previously in the other shard X. The 'debit' transaction in shard X destroys coins in that shard, whereas the 'credit' transaction taking place in shard Y creates coins in that shard, thereby showing how the legitimate transaction will take place.

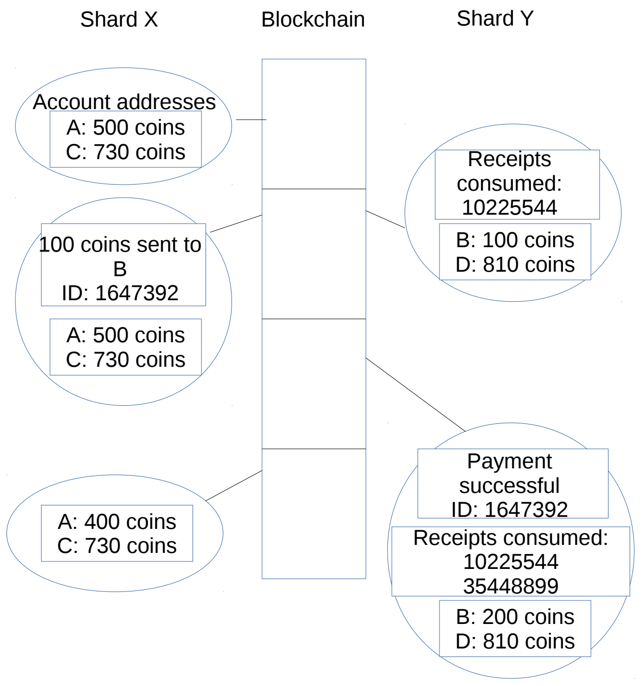


Figure 20: Example: Sharding

Sharding is made use of commonly in distributed systems such as MongoDB, MySQL and BigDB.

Scalable Blockchain Protocols

- **BitcoinNG**[EGS16] The main problem BitcoinNG aims to solve is the improvement in transaction throughput and latencies in the blockchain, at the same time reducing the transaction confirmation to a few seconds. This protocol improves performance by decoupling Bitcoin's blockchain operations into two different planes namely, serialising transactions and electing a leader. The inventors of the protocol, divide time into epochs. A chosen leader is responsible for serialising state machine transitions, in every epoch. The leader is also responsible for creating new blocks in order to facilitate state propagation. Therefore, there are two kinds of blocks in this protocol, i.e., a key blocks, that elect the leader, and microblocks, that record the transactions.

  BitcoinNG exhibits that the scalability of blockchain protocols can be improved until the only limiting factors are the individual node processing power, limiting throughput, and network diameter, limiting the consensus latency.

- **Lightning Network**[JpT16] The lightning network tries to address the issue of network scalability for blockchains. It uses the concept of *micro-payments* through payment channels for Bitcoin blockchains. Micropayments mean making small payments and receiving instant confirmations. The micropayments methodology allows two parties to open a payment channel for multiple payments and the actual transactions are recorded in the blockchain only after the channel closes. Without going into low level technical details of the protocol, it is necessary to bear in mind that this protocol mainly works with a minimal level of trust between two parties and keeps track of the channel time and whether the signatures for the transactions are properly done. The worst case scenario in this protocol is that one party keeps the funds of the other party locked for approximately a day. When the disadvantaged party opens a new payment channel, a refund is initiated since the other party had disappeared.

- **Bigchain DB** [MMM16] is a blockchain database that claims to write one million transactions per second. All Bigchain DB nodes connect to a single ReThinkDB[26] cluster. However, the flaw with the implementation is the failure to have independent storages. If the distributed database is attacked or if the blockchain has one malicious node that tries to delete the table, the entire system might be jeopardised. The other drawback to this blockchain database is the low performance of the interaction between the database and nodes connected to the blockchain.

[26]ReThinkDB is a distributed, NoSQL database. More information at https://www.rethinkdb.com/

9 Conclusion

The purpose of this thesis has been to study the blockchain system, and analyse if it is useful to employ the technology for identity and access management of constrained devices. A working proof of concept of an access management system was designed and implemented successfully on the Ethereum blockchain in order to assess the technology. This was done through extensive research and consideration of the choice of technology. The evaluation of the implementation proved that a blockchain based device management system may bring some advantage over the existing device management systems. Throughout the thesis, the advantages of using Blockchains have been revisited multiple times.

The evaluation also revealed some limitations of the proof of concept system implemented, most of which were issues related to blockchain technology in general. The time it takes to generate a block, in particular, is a big disadvantage to a real time IoT setup, as the readings indicated. The issues of scalability, energy consumption, cost and block generation time were therefore examined in detail. It is essential for these issues to have a solution in order to have an even widespread adoption of the blockchain system.

To conclude the thesis, despite the blockchain technology still being in early stages of development, it has many advantages that can be leveraged in a wide array of application domains. If the current challenges can be overcome, it can potentially change the way a lot of systems operate in the world.

10 Future Work

In continuation with the current research, it would be interesting to see how the system would perform with physical nodes and real IoT devices. The current system uses *Docker* containers to simulate a distributed blockchain environment, and just uses identification information of the constrained devices, since the idea was to first test out whether such a system is possible in the first place. An analysis of the data could determine the real world implications of the system to a certain extent. Therefore, an implementation of a full fledged hardware solution would be the most plausible next step.

In addition, it would be a good idea to include the suggested improvements for the system and develop a complete decentralized autonomous organization. This would lead to discovery of many more issues and challenges relevant to the system, in comparison with the controlled environment.

References

ABC16 Atzei, N., Bartoletti, M. and Cimoli, T., A survey of attacks on ethereum smart contracts. Technical Report, Cryptology ePrint Archive: Report 2016/1007, https://eprint. iacr. org/2016/1007, 2016.

ABH12 Assia, Y., Buterin, V. and Lior Hakim, Meni Rosenfeld, R. L., *Colored Coins whitepaper.* 2012. URL `goo.gl/1cqgcS`.

AbW03 Aboba, B. and J.Wood, *Authentication, Authorization and Accounting (AAA) Transport Profile.* 2003. URL `https://tools.ietf.org/html/rfc3539`.

And14 Antonopoulos, A. M., *Mastering Bitcoin.* O'Reilly Media, 2014.

AVS12 A. Sehgal, V. Perelman, S. K. J. S., *Management of resource constrained devices in the internet of things*, volume 50. 2012.

Ash09 Ashton, K., *That internet of things things.* RFiD Journal 22, 2009.

BE15 Bot, B. and Ethereum, *Enabling Blockchain Innovations with Pegged Sidechains.* 2015. URL `https://blockstream.com/sidechains.pdf`.

Bit16 BitFury, G., *Digital Assets on Public Blockchains Whitepaper.* 2016. URL `http://bitfury.com/content/5-white-papers-research/bitfury-digital_assets_on_public_blockchains-1.pdf`.

Bjo10 Bjorklund, M., *YANG - A Data Modeling Language for the Network Configuration Protocol (NETCONF).* 2010. URL `https://tools.ietf.org/html/rfc6020`.

But13 Buterin, V., Dagger: A memory-hard to compute, memory-easy to verify scrypt alternative. Technical Report, 2013. URL `http://www.hashcash.org/papers/dagger.html`.

But16 Buterin, V., *Ethereum 2.0 Mauve Paper.* 2016. URL `http://vitalik.ca/files/mauve_paper.html`.

But17 Buterin, V., *On Sharding Blockchains.* 2017. URL `https://github.com/ethereum/wiki/wiki/Sharding-FAQ`.

BEK14 C. Bormann, M. Ersue, A. K., *Terminology for Constrained-Node Networks*. 2014. URL `https://tools.ietf.org/html/rfc7228`.

BoH16 C. Bormann, P. H., *Concise Binary Object Representation (CBOR)*. 2016. URL `https://tools.ietf.org/html/rfc7049`.

CML99 Castro, M. and Liskov, B., *Practical Byzantine Fault Tolerance*. Proceedings of the Third Symposium on Operating Systems Design and Implementation, 1999.

DrB15 Dryja, T. and Buterin, V., Dagger-hashimoto protocol specifications. Technical Report, 2015. URL `https://github.com/ethereum/wiki/blob/master/Dagger-Hashimoto.md`.

Dou02 Douceur, J. R., The sybil attack. IPTPS '01, 2002, URL `http://dl.acm.org/citation.cfm?id=646334.687813`.

SBV17 der Stok, P. V., Bierman, A., Veillette, M. and Pelov, A., Coap management interface. Technical Report, 2017. URL `https://tools.ietf.org/html/draft-vanderstok-core-comi-11`.

EB15 Ethereum and Bot, B., *Hydrachain open source code*. 2015. URL `https://github.com/HydraChain/hydrachain`.

EGS16 Eyal Ittay, Gencer Adem Efe, S. E. G., Bitcoin-ng: A scalable blockchain protocol. *Proceedings of the 13th Usenix Conference on Networked Systems Design and Implementation*, NSDI'16. USENIX Association, 2016, pages 45–59, URL `http://dl.acm.org/citation.cfm?id=2930611.2930615`.

Fer16 Fernandez, J. O., *Lisk Academy*. 2016. URL `https://academy.lisk.io/`.

FeA11 Fette, I. and Melnikov, A., *The WebSocket Protocol*. 2011. URL `https://tools.ietf.org/html/rfc6455`.

Eth15 Foundation, E., Ethash design rationale. Technical Report, 2015. URL `https://github.com/ethereum/wiki/wiki/Ethash-Design-Rationale`.

Eth16 Foundation, E., *Ethereum white paper - a next-generation smart contract and decentralized application platform.* 2016. URL `https://github.com/ethereum/wiki/wiki/White-Paper`.

Hyp16 Foundation, L., *Hyperledger Whitepaper.* 2016. URL `https://github.com/hyperledger/hyperledger/wiki/Whitepaper-WG`.

BiF17 Foundation, B., *Bitcoin Weaknesses.* 2017. URL `https://en.bitcoin.it/wiki/Weaknesses`.

CoF17 Foundation, C., *Ethereum Safety.* 2017. URL `https://github.com/ConsenSys/smart-contract-best-practices`.

Gav14 Gavin, W., Ethereum: A secure decentralised generalised transaction ledger., 2014. `http://gavwood.com/paper.pdf`

Gre15 Greenspan, G., *MultiChain Private Blockchain White Paper.* 2015. URL `http://www.multichain.com/download/MultiChain-White-Paper.pdf`.

Har15 Hartke, K., *Observing Resources in the Constrained Application Protocol (CoAP).* 2015. URL `https://tools.ietf.org/html/rfc6120`.

KCE15 Harry, K., Miles, C. and Ellenbogen Paul, Joseph Bonneau, A. N., *An empirical study of Namecoin and lessons for decentralized namespace design.* 2015. URL `http://randomwalker.info/publications/namespaces.pdf`.

CFS90 J. Case, M. Fedor, M. S. J. D., *A Simple Network Management Protocol (SNMP).* 1990. URL `https://tools.ietf.org/html/rfc1157`.

J.D14 J.D.Bruce, *The Mini-Blockchain Scheme.* 2014. URL `http://cryptonite.info/files/mbc-scheme-rev2.pdf`.

Jen96 Jensen, F. V., *An introduction to Bayesian networks*, volume 210. UCL press London, 1996.

Jim16 Jimenez, J., Open mobile alliance-lightweight m2m specifications. Technical Report, 2016. URL `https://tools.ietf.org/html/draft-jimenez-t2trg-coap-functionality-lwm2m-00`. RFC.

KB16 Kordek, M. and Beddows, O., *Lisk Whitepaper.* 2016. URL `https://lisk.io/whitepaper`.

CDE16 Kyle Croman, Christian Decker, I. E. A. E. G., On Scaling Decentralized Blockchains, 2016. `http://fc16.ifca.ai/bitcoin/papers/CDE+16.pdf`

Ler14 Lerner, S., Strict memory hashing functions. Technical Report, 2014. URL `http://www.hashcash.org/papers/memohash.pdf`.

Ler15 Lerner, S. D., Rootstock: Bitcoin powered smart contracts. Technical Report, 2015. URL `http://www.the-blockchain.com/docs/Rootstock-WhitePaper-Overview.pdf`.

ToA15 *Leshan Implementation.* 2015. URL `https://github.com/eclipse/leshan`.

LRM81 Leslie Lamport, Robert Shostak, M. P., The Byzantine Generals Problem , 1981. `http://research.microsoft.com/en-us/um/people/lamport/pubs/byz.pdf`

Jpm14 Morgan, J., *Advanced Message Queuing Protocol (AMQP) standard.* 2014. URL `http://www.iso.org/iso/home/store/catalogue_tc/catalogue_detail.htm?csnumber=64955`.

VPS16 M. Veillette, A. Pelov, A. S. R. T. A. M., *Constrained Objects Language.* 2013. URL `https://tools.ietf.org/html/rfc7049`.

Joh07 O'Hara, J., *Toward a commodity enterprise middleware.* 2007. URL `http://oldwww.acm.org/acmqueue/digital/Queuevol5no4_May2007.pdf`.

Org15 Org, B., Delegated proof-of-stake consensus. Technical Report, 2015. URL `https://bitshares.org/technology/delegated-proof-of-stake-consensus/`.

Pau13 Paul, D., Beyond mqtt: A cisco view on iot protocols. Technical Report, 2013. URL `http://blogs.cisco.com/digital/beyond-mqtt-a-cisco-view-on-iot-protocols`.

JpT16 Poon, J. and Dryja, T., The bitcoin lightning network: Scalable off-chain instant payments. Technical Report, 2016. URL `https://lightning.network/`. draft.

Sat08 Satoshi, N., *Bitcoin: A peer-to-peer electronic cash system*. 2008. URL `http://bitcoin.org/bitcoin.pdf`.

RSS94 Sandhu, R. S. and Samarati, P., *Access control: principle and practice*, volume 32. 1994.

Ste14 Stewart, I., *Slimcoin*. 2014. URL `http://www.slimcoin.club/whitepaper.pdf`.

Sza97 Szabo, N., *The Idea of Smart Contracts*. 1997. URL `http://szabo.best.vwh.net/smart_contracts_idea.html`.

Tel94 Teller, A., Turing completeness in the language of genetic programming with indexed memory. *Evolutionary Computation, 1994. IEEE World Congress on Computational Intelligence., Proceedings of the First IEEE Conference on*. IEEE, 1994, pages 136–141.

MMM16 Trent McConaghy, Rodolphe Marques, A. M., *BigchainDB Whitepaper*. 2016. URL `https://www.bigchaindb.com/whitepaper/`.

Vuk15 Vukoli, M., *The Quest for Scalable Blockchain Fabric: Proof-of-Work vs. BFT Replication*. 2015. URL `http://www.vukolic.com/iNetSec_2015.pdf`.

VaK15 V. Vaishnavi, B. K., Design research in information systems. Technical Report, 2015. URL `http://desrist.org/design-research-in-information-systems/`.

SHB14 Z. Shelby, K. Hartke, C. B., *The Constrained Application Protocol (CoAP)*. 2014. URL `https://tools.ietf.org/html/rfc7252`.

Appendix 1. Contract Creator function description

Compile contract

Input: Smart contract code in Solidity
Output: Compiled version of the smart contract

Description: This function uses the solidity compiler binding library for javascript. The smart contract code needs to be stored into a variable, without any line breaks, for this function to work.

Calculate gas

Input: The full transaction along with the data to be passed
Output: The units of gas required to run the transaction

Description: This function uses the *estimateGas* method from the external *Web3* javascript library, to determine the approximate amount of gas one needs to supply for the contract upload transaction to be successful.

Deployment format

Input: The compiled smart contract code
Output: Ready to deploy smart contract

Description: This function determines the *ABI*[27] of the smart contract, and encapsulates the compiled code into error handling blocks. It also takes care of other things such as having an unlocked user account for the deployment transaction, and checking if all the parameters required to upload the contract have been taken care of. It outputs the deployable version of the compiled smart contract code.

[27]ABI: Application Binary Interface is a description of all the functions included in the solidity smart contract and how to use those functions.

Deploy

Input: Ready to deploy smart contract
Output: Smart contract deployed on the blockchain

Description: This function deploys the smart contract that the previous function produced as an output, and outputs the smart contract address if successfully deployed.

Appendix 2. Smart Contract function description

Manager Authentication Check

Input: Manager's Ethereum address
Output: True or False

Description: It is a constant function that verifies whether the manager updating rules for a certain device in the smart contract memory is the one responsible for that particular device. It returns a Boolean value.

Query Manager

Input: Manager's Ethereum address, Device's Ethereum address
Output: True or False

Description: This function takes the Manager and the constrained device addresses as inputs, and queries whether a certain Manager is responsible for a certain device or not. It returns a Boolean value.

Query Rule

Input: Manager's Ethereum address, Ethereum addresses of the two communicating devices
Output: True or False

Description: This function takes the Manager, and two constrained device addresses respectively as arguments. It queries whether a certain constrained device can access a particular constrained device or not. It returns a Boolean value.

Add Manager

Input: Manager Ethereum address
Output: True or False

Description: This function adds a manager in the smart contract memory on the blockchain. This function executes when a manager broadcasts an identitiy registration message, essentially making a call to the add manager function to add itself to the blockchain. It stores the manager address as a key value in a mapping, and returns a boolean value confirming if the add was successful. It also broadcasts a message on the blockchain claiming that the manager with the given address has been added.

Add Device

Input: Ethereum addresses of the Manager and Device
Output: True or False

Description: The manager of a constrained devices calls this function to add the particular device. It takes the manager address and the constrained device address as arguments, and adds the constrained device address in the value array corresponding to the key value mapped pair with the manager address as the key value. It then broadcasts a message over the blockchain indicating that the device corresponding to the specific manager has been added.

If the manager needs to add more devices corresponding to it, those addresses go into subsequent array indices in the value array of the key value pair.

Add Rules

Input: Ethereum addresses of the Manager and two communicating devices, Rule information
Output: True or False

Description: This function takes the manager's address, the constrained device's address that the manager manages, the new device's address that can now have access to the former device, the resources it can have access to, for how long the access is valid and what kind of access rights the latter can have over the former's specified resources. Access rights usually range between none, read and write indicated by numerical values, and the expiration time is specified by the number of blocks after which the access rights are expected to expire. The number of blocks is an indication of the time, in case of blockchains. After the data is stored in the smart contract memory, a message is broadcast to the blockchain indicating that a new rule has been added with the relevant details.

Remove Device

Input: Ethereum addresses of the Manager and Device
Output: True or False

Description: This function takes the device address and the manager address as inputs and updates the value corresponding to the key value of manager address to zero. Then a message is broadcast over the network specifying the details of the device that has been de-registered from a particular manager on the blockchain.

Remove Rules

Input: Ethereum addresses of the Manager and the two communicating devices
Output: True or False

Description: This function takes the constrained device addresses and the manager address as inputs and updates the value corresponding to the key value of constrained device address to zero. Then a message is broadcast over the network specifying the details of the rule that has been removed.

Remove Manager

Input: Ethereum address of the Manager
Output: True or False

Description: This function takes the manager address as an argument. It first checks for any constrained device addresses corresponding to the manager as the key value. If there are none, it goes ahead with setting the corresponding value to null, and returns a true boolean value. Otherwise, an error is returned. This function also broadcasts a message over the network specifying the details of the manager that has been de-registered from the blockchain.

Execute Smart Contract Later

Input: Expiration time specified as a rule information
Output: True or False

Description: This function takes the expiration time of a rule as the input. It then includes an internal logic to execute a transaction to the smart contract after the given expiration time, to update the rule description. A trusted external contract, called Ethereum Alarm Clock [28], is also made use of to schedule the contract calling after the contract has expired.

Kill Smart Contract

Input: Ethereum address of the contract creator
Output: True or False

Description: Only the creator of the smart contract has access to this function. This *suicide* function was implemented in the prototype in the event of a bug or an unwanted behaviour that needs to be removed.

[28]Ethereum Alarm Clock: http://www.ethereum-alarm-clock.com/

Appendix 3. Client libraries used

React

React is a javascript library meant to create interactive user interfaces. The code of this library is mainly maintained by Facebook. It was used in the project to create the Client user interface. https://facebook.github.io/react/

Web3

Web3 is a javascript library that communicates through RPC calls on a local Ethereum node, and basically exposes the RPC layer to communicate with Ethereum directly. https://github.com/ethereum/wiki/wiki/JavaScript-API

Solc

Solc javascript library binds javascript code to Solidity compiler, thereby making it possible to compile the smart contract code and deploy it. It uses Nodejs internally to do so, and more information about it can be found at https://github.com/ethereum/solc-js.

Lodash

Lodash is a javascript utility library that contains many inbuilt high performance functions such as iterating through a javascript object, searching in javascript objects, using properties from different javascript objects to form a new object, etc. https://lodash.com/