KING'S College LONDON

University of London

Smart Home Inventory Management Using Private Blockchain and Purchase Order Contract

Final Project Report

Author: Jae Seok An

Supervisor: Martin Chapman

Student ID: 1447414

April 18, 2017

Abstract

Twenty-eight billion Internet of Things (IoT) devices are forecast by 2022 [Ericsson, 2016], which certainly will generate a large amount of network traffic. Nowadays, a cloud-based centralised database model is prevalent as an infrastructure for IoT. However, it has a few problems as an ideal infrastructure for IoT: 1) bottlenecks can occur 2) a single point of failure can down the entire system, and 3) it requires intermediaries. Private blockchain can decentralise the network, and it will allow safe transactions without any intermediaries. Furthermore, private blockchain would be a bridge towards a new paradigm of the sharing economy with public blockchain. This project presents designs and implementation of smart home inventory management system using private blockchain. The report evaluates the effectiveness of different smart contract designs. In addition, the usefulness of private blockchain for the IoT network, as an alternative to the centralised cloud model, is explored.

Contents

Chapter 1

Introduction

1.1 Motivation

A typical family home is predicted to be equipped with more than 500 Internet of Things (IoT) devices by 2022 [Meulen and Rivera, 2014]. This would provide an excellent infrastructure to build intelligent systems that could improve the way we live. Jarvis is a virtual assistant, designed by Mark Zukerberg.1 Jarvis surprised many people by demonstrating what AI can do with home IoT devices. The automated home inventory system is one of the examples of home intelligent systems; it refers to the system that is able to purchase items needed in home automatically. This system can save us time, and allow us to spend more time with friends and family. It can be especially helpful to seniors living alone, as it might be difficult for them to go out often, as well as the vast majority may not be tech savvy to conduct online shopping.

Most similar home intelligence systems are based on a centralised cloud storage database; however, as a number of devices in home are rapidly increasing, the security and efficiency of the current model are being questioned. Secureness of a database is important in inventory management as it involves financial transactions. [Conoscenti et al., 2016] and [Unknown, 2017] provide the reasons why a blockchain database is more secure and could be useful for the IoT network. Data will be stored in a block, which has a cryptographic signature of previous blocks and a timestamp, making it extremely difficult to modify data compared to a normal centralised database. On the other hand, the blockchain database model is still not widely accepted because its effectiveness is yet to be proven.

Assuming that there is a home assistant technology that can also control a laundry machine, as it is connected to a cloud server. This laundry machine sends the level of fabric softener, which is stored inside the machine, to the cloud server. The server, upon detecting that the fabric softener is running low, contacts a few suppliers to find an available supplier. Then, the server will use the bank transaction service to make a payment to a supplier. There are two limitations in this model. Firstly, there is an intermediary, such as a bank, to make the transaction. There will be the transaction fees, and the record of the transaction will be known to a third party, which is the bank in this case. Secondly, there is a single point of failure. If the cloud server stops, the system will be stuck and it will also be difficult to recover data in the event it is lost by the cloud server.

Using blockchain as a database, the whole process could be processed without any intermediaries, which will shorten the processing time. The data will be kept within two stakeholders, which means the system owner will have full control of data generated from their IoT devices. There will be no transaction cost to intermediaries. As the blockchain database will be distributed, there is no single point of failure. It will remove bottlenecks occurring on the centralised server. Data will be kept more securely in case of disaster. This system also benefits the suppliers as they do not need to verify the order when they use the blockchain backed system. In the long run, this blockchain backed system could yield many benefits such as saving energy consumption, saving significant costs in verification, fraud, data ownership, transparent economy and so on.

1.2 Objective

The main objective is developing a prototype of a private blockchain backed smart home inventory management system. This system will demonstrate how the private blockchain system could be utilised in smart home to overcome the limitations of a centralised cloud database. This system aims to show how it can work without any intermediary, as well as show methods how to improve the single point of failures in the current database system.

This prototype aims to show the core function of the inventory management system, which is purchasing the items automatically. The system should be secure enough to be used in the real-world and should be scalable to be useful.

Another objective is showing how this system could further develop to blueprint a new paradigm of the economy using blockchain.

1.3 Scope

The project will focus on tackling the trickiest part in the inventory management, which is purchasing the items securely. The project will not implement the system that updates the inventory level among the IoT devices.

When purchasing the items, there are at least three stakeholders, namely the supplier, buyer and deliverer. This project will not design a system that the deliverer can participate in. However, this project is aware that the deliverer will need to be taken into consideration to further develop the system, so it will be briefly considered in the design.

In order to have no intermediary for this system, it is possible to implement the cryptocurrency in the system. However, it is not a viable idea at this stage, simply because this cryptocurrency will not carry much value if not many suppliers and homeowners use it. If it does not carry value as currency, this cryptocurrency should be exchanged for real cash and intermediary will be required for a smooth exchange. Using bitcoin is one of the solutions for this. Alternatively, a smart contract could send stored bank information to the bank directly to send money to suppliers. Since this logic is stored inside the smart contract, it will be secured from fraud, as long as suppliers are verified. The bank will receive minimal information about the transaction and this method also opens the possibility of using bitcoin-like solutions in the new paradigm of a cryptocurrency economy.

Chapter 2

Background

2.1 Smart Home Inventory Management

The Internet of Things (IoT) paradigm refers to the world where many objects around us are connected to other objects through the internet network [Gubbi et al., 2013]. According to [Gubbi et al., 2013], the emergence of IoT will impact four application domains: (1) Personal and Home, (2) Enterprise, (3) Utilities and (4) Mobile. This report will focus on Personal and Home domain in IoT.

The Smart home is a concept wherein objects in a home are connected digitally. Originally, smart home technology controlled environmental systems are in a home such as lighting and heating [Ricquebourg et al., 2006], but now it includes almost every object in a home including the microwave, laundry and almost everything that is used in a home. Smart home technology could be used to monitor energy consumption, lifestyle, inventory and security. Use cases of smart home technology could be divided into two in terms of involving intermediaries. Simply turning on and off the lighting system using a controlled device does not require any intermediary to be involved. Activities, like monitoring energy consumption or analysing life pattern could be achieved by intermediaries by sending data to the service provider or within a home using the analytic software without transferring information to intermediaries. On the other hand, an Inventory management system, which tracks inventory level, sends purchase orders and manages deliveries could not be achieved without involving intermediaries [Lesonsky, 1998]. It is because purchasing stocks require an authorised financial organisation and involves security risk, because purchasing inventory requires information to be transferred outside of the home and financial transaction.

Research [Ricquebourg et al., 2006] concluded that there are two necessary components to facilitate smart home technology; a network infrastructure for home IoT to communicate and software architecture to utilise the data obtained. [Soliman et al., 2013] and [Ye and Huang, 2011] described how the cloud-based network infrastructure could be incorporated with IoT to advance smart home technology. Research [Dorri et al., 2017] suggested using blockchain as a mean to communicate information for the smart home.

2.2 Blockchain Technology

In 2005, Szabo described an idea of a cyber-currency that could overcome limitations of real currency, which has been under the influence of counterfeiting or theft [Szabo, 1998*a*]. Bitcoin is a cyber-currency wherein digital signatures provide validity of its value [Nakamoto, 2008]. It is a completely distributed and digitalised version of cash [Nakamoto, 2008]. Transaction of cash is one of the basic requirements for currency and a common problem in the transaction is double spending, which could occur when there is no way to completely track every transaction of the currency. When a trusted third party is involved, the double spending problem could be easily solved. However, it means that some party must have an authority of currency, and drawbacks show a lack of fairness and effectiveness. One of the ways to solve this issue without involving the third party is using a timestamp server [Nakamoto, 2008]. Whenever a transaction occurs, transaction information with a timestamp will be broadcasted to all participants in a network [Nakamoto, 2008]. Then, the participants in the network will agree to the transaction, a process known as proof-of-work [Nakamoto, 2008]. Therefore, double spending could be prevented in a completely decentralised network without a third party. Blockchain is timestamp hash-based proof-of-work block with a cryptographic device providing its own validity. It is a database, which a blocks store data. It is shared by everyone, so there is the absence of centralised authority with absolute power to validate the network. This type of blockchain is referred to a public blockchain.

2.3 Public and Private Blockchains

Distributed, secured and transactive currency could be useful for the smart home inventory management system. A Distributed network could provide effectiveness, which means an IoT device could make transactions without a central server securely. Foremost, it will provide transparency and fairness in inventory transaction. However, there are critical limitations in implementing public blockchain in the smart home inventory management system.

First of all, the owner of the smart home system will not have the privacy of their inventory data. Secondly, proof of work requires a heavy amount of computation. At present, transaction cost for a public blockchain is about $0.01 for one transaction [Buterin, 2015]. However, the transaction cost could be reduced up to $0.0003 with scalability technology [Buterin, 2015]; yet this is currently uncertain. Lastly, a public blockchain will prevent double spending and secured transaction but it will not guarantee on the successful transaction; for instance, a failed transaction that an attacker could disguise as a supplier for inventory and take money without sending actual inventories. Hence, using public blockchain network as a smart home network is not ideal in a typical home where transparency is not required. Nevertheless, there are exceptional cases when the importance transparency and fairness outweigh the limitations mentioned earlier. For instance, the government office may want to publicise their inventory data in order to provide transparency and give equal chances for every supplier to meet the standard. In such a case, using public blockchain might be encouraged.

Private blockchain has an access control layer [Monax, n.d.*b*], so only a permitted node could access a copy of the private blockchain. There is a central node, which could control the permission and the central node monitor's write-permissions [Buterin, 2015]. Read permission could be fully public or restricted [Buterin, 2015]. Private blockchain could overcome limitations that public blockchain faces.

Firstly, only a small number of validator nodes are involved in validating private blockchain, whereas every node is involved in the public blockchain. Therefore, it is much easier to scale the network because it is less costly and has a lower latency when exchanging data. Secondly, unpermitted users cannot access the private blockchain, so inventory data will only be shared among stakeholders. The smart home network owner will have the authority to control permission level to different stakeholders. Thirdly, private blockchain is generally more secure because only trusted nodes can join the network. Although security could be threatened when the attacker sneaks into a node with accessibility, private blockchain network could minimise damage from attackers by removing the hacked node from the network. Lastly, private blockchain also offers more technical options such as changing the rule of smart contract or editing data in the blockchain. This flexibility is essential in smart home inventory management as there are various scenarios where changing of logic or database is required to debug system issues. For example, if the laundry sensor failed to report that it is running out of stock, there should be a way for the house owner to manually write data in a database. Hence, private blockchain is more suitable for a smart home inventory management network due to these reasons.

2.4 Private Blockchain and Centralised database

Databases that are being used most widely, both SQL and no-SQL type databases, are contained within one entity [Thompson, 2016]. Despite the fact that centralised databases with cloud network are currently being adopted in IoT, there are two major drawbacks to adopting this in smart home inventory management system [Greenspan, 2008].

Firstly, a third party will be involved all the time using the centralised database, whether it is a provider of cloud or smart home service. There will be a maintenance fee for a service provider to maintain the system. Although smart home's data will be secured, the information needs to be shared with a third party. On the other hand, the private blockchain database does not need to be maintained unless business logic needs to be changed, and all the information generated by home IoT devices will be shared only among verified suppliers and the smart home owner. Direct communication between the supplier and the smart home could lead to high efficiency; yet, this needs to be discussed in depth to be concluded.

Secondly, a risk of single point of failure is another major drawback. A failure of accessing the centralised database system will influence all smart home owners using that database system. In case of hacking or disaster, all the data may be lost eternally or will be expensive to recover [Greenspan, 2008]. However, private blockchain solves this issue as it is robust in fault tolerance [Greenspan, 2008]. There is no single point of failure as every node has a copy of the database. If some nodes are down for a while, it will always be able to receive the most recent version of the database when it is connected to the private blockchain network [Greenspan, 2008].

The centralised database has strengths in performance. When a transaction is executed, one single record will be written, whereas a transaction will be recorded in every node in a private blockchain network. This record needs to be agreed upon by every node in the network, which makes private blockchain to scale.

2.5 Smart Contract

Szabo's paper *Secure Property Titles with Owner Authority* describes three requirements for the ideal database to manage digitalised assets.

(1) Servers should not be able to forge transfers (2) Servers should not be able to block transfers to or from politically incorrect parties (3) Current owner, Alice, should be able to

transfer her title to only a single relying counterparty (double spending problem). [Szabo, 1998*b*]

Bitcoin is one on the applications fulfilling the above properties using blockchain. Bitcoin's transaction is open to everyone, but nobody has control over it. It is safe from a double spending problem. A Bitcoin transaction is a contract. This contract indicates changing of states according to predefined rules. A sends X btc to B. It will reduce X btc from A's account, and will increase X btc in B's account.

A Smart contract consists of digitalised codes that represent a contract and is self-executed [Tapscott and Tapscott, 2016]. Foremost, a smart contract satisfies the above three properties. There are many platforms inspired by bitcoin's transaction and are providing smart contract functionality that is backed by blockchain.

A Smart contract is a critical element for purchasing stocks in smart home inventory management system. The rules to purchase stocks will be self-executed when the system receives information that an inventory is running out. In any environment, the logic of smart contract will always work as it is deployed in the blockchain [Monax, n.d.*e*].

2.6 Eris

Eris is a platform consisting of Eris-db and four modules [*Eris(Monax)*, 2017]. Eris-db is a private blockchain network, which could be created by the Eris platform. It has a permissioned layer. Module Chains module helps to create public and private keys to access blockchain with a different level of permissions. Eris uses Tendermint consensus protocol [Monax, n.d.*c*], which is a Byzantine fault-tolerant [Monax, n.d.*a*]. This consensus protocol uses the proof-of-stake system. The participants in the network could have different numbers of stakes. In order to update the Eris-db, more than 66.6

Eris is chosen to build the system as it is highly optimised in logic and transaction, and is permissioned [Monax, n.d.*b*]. Eris provides various tools to develop a system backed by private blockchain and smart contract. Eris-db can be accessed through JavaScript application [*JavaScript*, n.d.]. Eris platform also provides the library that allow the JavaScript application to interact with contract that is written in Solidity [*Solidity*, n.d.]. Therefore, Eris platform it will allow me to focus on developing a prototype and evaluate its effectiveness, usefulness and secureness.

2.7 Related Work

[Conoscenti et al., 2016] illustrates 18 use cases of blockchain and 4 of them are blockchain in IoT. The author of this research aims to utilise the blockchain and P2P approaches in the IoT network without centralised authority or a third party [Conoscenti et al., 2016]. My report was originally inspired from [Conoscenti et al., 2016] as it suggests that a decentralised IoT network has benefits of integrity, anonymity and adaptability.

[Zhang and Wen, 2015] describes the redesigned IoT Electric Business Model using bitcoin blockchain. [Wöner and Bomhard, 2005] suggests the bitcoin protocol is exchanging sensor data with IoT.

[Unknown, 2017] discusses on security and performance of a decentralised IoT network in comparison to a centralised network. My project agreed on the claim from the paper that Blockchain network has better security than centralised network.

I have not identified any papers that deal with IoT inventory management use cases using private blockchain.

Chapter 3

Requirements & Specification

3.1 Domain Concepts

The following concepts will be used throughout the report.

Inventory Management System is a prototype of a private blockchain backed system, which is developed for this project. The system in this report refers to the Inventory Management System.

An Agent refers to a node that is registered in the system. An agent is identified with an address, which is a 20 byte String. Each agent has a pair of public and private keys. There are five types of agents in this system; (1) Administrator (2) Device (3) Supplier (4) Developer and (5) Deliverer. Devices' agents are completely automated. They will function without input from users, and they are only executed with pre-defined rules. Although there are five agents in the system, and this system will support more agent types, the core agents for this system are the administrator, the device and the supplier agents. Hence, developer and deliverer agents may not be mentioned directly in some of the requirements and specifications.

A User refers to a human or a group of humans using this system. Homeowner and supplier are the main users of this system. There will be one homeowner user in the system, and a homeowner owns one administrator agent and more than one device agents. Suppliers represent either an organisation or an individual who has been verified by the homeowner to supply inventory. The Developer and deliverer are also possible users for this system. All the users will need to receive permission from the homeowner to be in this system. The concept of the user will be used in the user story to describe what the real life user would like to do.

Eris-db is the private blockchain backed network that is used in this system. It is also functioning as a database in this system. Eris-db has the blockchain and every agent in this system will keep a copy of this blockchain. Eris-db also refers to this copy of blockchain in this report.

PO Contract is a name of the smart contract that exists in the Eris-db. PO is an abbreviation for Purchase Order. It is also a digitised representation of a legal purchase order contract.

Item code is a unique identifier of the item name in this system. For instance, FS12 could represent the fabric softener from a certain brand with a certain size.

3.2 User Stories

∗ As a homeowner, I want my home inventory to be automatically managed.

∗ As a homeowner, I do not want any intermediary in purchasing items.

∗ As a homeowner, I want this system to be decentralised so that the system is more robust than the centralised cloud system.

∗ As a homeowner, I want to keep my inventory data safe.

∗ As a homeowner, I want to have the data ownership of the inventory data created from this system.

∗ As a homeowner, I want to add or remove supplier and device in the system.

∗ As a homeowner, I want to set purchase orders, which indicate an item code, the quantity of the item and a price, so that I do not need to purchase them manually.

∗ As a homeowner, I want to update the purchase orders that I created.

∗ As a supplier, I want to sell as many as items as possible, as long as it is profitable.

∗ As a supplier, I want to set the price I am willing to pay for an item so that I do not sell the unprofitable items.

∗ As a homeowner and a supplier, I want to know when the purchase order is completed or failed.

3.3 Functional Requirements

The functional requirements are the functions of the system to satisfy the needs of the users.

· The system has different permission levels to access Eris-db and smart contracts for different agents.

· Homeowners can create all types of agents in the system.

· Homeowners can remove any agents, except the administrative agent, from the system.

· Homeowners can create a purchase order, which indicates an item code, the quantity of the item and the maximum price one is willing to pay for the items.

· Homeowners can set the rules on when the device agent can open and close the purchase order

· A device agent opens and closes the purchase order according to the rules that are set by the homeowners.

· A device agent can obtain inventory data from its sensor

· A device agent can share the inventory data to other devices

· Suppliers should be notified when the purchase order is opened

· Suppliers can respond to purchase orders automatically

· Suppliers can set the price they are willing to offer for the corresponding purchase order

· Purchase orders should have legal power

· Purchase orders can be re-opened

· Once a purchase order is opened, the quantity and the price cannot be updated.

· The logic of purchase order cannot be changed.

3.4 Non-Functional Requirements

The following non-functional requirements are necessary for this system to achieve the objective of this project, which is developing the decentralised system that can replace the centralised database and function without any intermediary.

Purchase orders should be secure. Purchase orders need to be safe to use for both parties. In the perspective of the homeowner, purchase orders can be unsafe when an unauthorised user can access or use this system. This project assumes a purchase order is safe for the homeowner when the supplier agents cannot call the expected method from the device agents or the administrator agent.

It is unsafe for the supplier when the purchase order cannot guarantee to give promised value to the suppliers. This can be solved if the PO contract can have the legal power, because this contract will give the right for suppliers to receive the money from the homeowner.

The system should be decentralised. There are different degrees to how decentralised the system is. For instance, in the fully distributed system, all agents in the system must have the exact same level of permission and a number of bonds. A fully distributed system is inefficient because the proof-of-work system require heavy amount of computation to verify the data. Also, it does not provide privacy to the homeowner, as the home data could be seen by anyone. Therefore, the system should be decentralised to reduce bottlenecks and heavy network traffic, but the system will not be fully distributed.

The system should be scalable and usable. The scalability and the usability are important aspects for this application to be used in the real world. These two aspects are mainly depending on the design of the system.

3.5 Specification

The system should be running all the time on the blockchain network.	A computer running administrator node should run Eris platform all the time.

The system has different permission levels to access Eris-db for different agents.	It is called Chain tooling to initialise different types of agents to the Eris-db. Two files need to be set before booting the system. Firstly, permission levels of the different types of agents need to be defined in the *account-types* folder in the Eris platform. In this system, the permission level of the administrator, developer, device, supplier and the deliverer needs to be defined. Secondly, create a file indicating the number of agents for each type in the *chain-types* folder. The file, *Genesis.json*, indicates how the blockchain will be initiated, and the newly designed chain should be indicated in this file.
Homeowners can create all types of agents in the system.	In order to add a new agent, the homeowner should call the *GenerateAccount(accountType)* method, which generates an account and returns private and public keys.
Homeowners can remove any agent, with the exception of the administrative agent, in the system.	In order to add a new agent, the homeowner should call the *RemoveAccount(address)* method, which generates account and returns private and public keys.
Homeowners can create a purchase order, which indicates an item code, the quantity of an item and the maximum price they are willing to pay for the items.	Homeowner agents should run an application, create the purchase order contract with parameters of item code, the quantity of an item and the maximum price. For instance, *createPurchaseOrder(itemcode, quantity, price).*

Homeowners can set the rules on when the device agent should open and close the purchase order.	Homeowners can use the *setRule(itemCode, minInvLevel, time)* method in a JavaScript application to set the rules for the device agent. When the device has a number of items that is lower than *minInvLevel*, the device agent will call the *openPurchaseOrder(item code)* method.
A device agent opens and closes the purchase order according to the rules that are set by the homeowners.	Device can call the *openPurchaseOrder(item code)* method in the smart contract. Once this method is called, suppliers will be notified and will be able to set the supplier price. Then, the device can call *closePurchaseOrder(item code)*. The smart contract will contact a bank to make a transaction, and the purchase order will be closed if the transaction is completed.
A device agent can obtain inventory data from its sensor	The device should read a text file, which represents the current data information. This information should contain an array of item code and the quantity of the item.
A device agent can share the inventory data to other devices	There should be a smart contract that saves the current inventory information. The device can read and write those data.
Suppliers should be notified when the purchase order is opened	The openPurchaseOrder method should invoke the event *POopen(item code, quantity, maxPrice)*
Suppliers can respond to purchase orders automatically	The supplier should subscribe to *POopen* event. When the supplier receives the purchase order that matches the supplierâĂŹs availability, the supplier can send the *setSupplierPrice(item code, price)* method to the smart contract.

Suppliers can set the price they are willing to offer for the corresponding purchase order.	The supplier can set the item code and price variables prior to the *POopen* event happen.
Purchase orders can be re-opened	In order to re-open the purchase order, the purchase order need to be closed first.When the purchase order is closed, it should not be removed. This contract can be re-opened anytime.
Once the purchase order is opened, the quantity of an item and the price cannot be updated.	There should be an instance variable in the smart contract representing the status of the purchase order. If it is opened, the device or the administrator cannot update the price using the *createPurchaseOrder* method.
The logic of purchase order cannot be changed.	All the logic that is related to the purchase order must be kept inside the smart contract. The logic as follows: *The administrator creates the purchase order. The device opens the purchase order. The supplier responds to the purchase order by sending the price for the purchase order. If the money is successfully transferred to the supplier, the purchase order will be closed.*
The Purchase order should be secure to use for the homeownerâĂŹs	*setPermission* method, which can only be called from the administrative agent, set permissions to access methods in smart contract to all agents in the system.
The system should be decentralised.	Eris-db is a distributed database system. Therefore, every agent will have a copy of Eris-db. The validation process needs to be distributed. More than 66.6% of a node must be presented on the network for the system to be working. This can be done by distributing the stake of bonds among the devices network.
Purchase orders should have legal power	Dual integration should be done.

Table 3.1: Specification table

Table 3.2: Permission specification for the smart contract

Permissions	Aministrator	Device	Supplier
setPermission	Available	NotAvailable	NotAvailable
createPurchaseOrder	Available	NotAvailable	NotAvailable
open/closePurchaseOrder	Available	Available	NotAvailable
setSupplierPrice	Available	NotAvailable	Available

Permission specification for PO manager methods is shown in Table 3.2.

// Chapter 4

Design

4.1 Design Overview

The blockchain backed smart home inventory management system has two benefits over the centralised database system:

· it reduces dependence on intermediary and

· is decentralised.

First of all, the system functions with minimum level of dependency on intermediaries. The device should contact the supplier directly, and vice versa. The device should manage inventory themselves. The system will use smart contract to make the purchase order happens without any intermediaries. Upon the agreement on the purchase order from the device and the supplier, a PO contract will be triggered to complete the transaction. Secondly, in order to decentralise the system, the bond of out any intermediaries, as it can purchase inventory without financial organisation, such as a bank.

Therefore, transaction fee will be significantly reduced, and the homeowner will keep the data within stakeholders, which are homeowners themselves and suppliers. Secondly, this system is decentralised as each device and supplier will keep a copy of the database. Since each home can transact to suppliers without the centralised organisation, a bottleneck on the central server is less likely to happen. Also, data will be safely stored in case of disaster. The design focus should be on taking these advantages of a blockchain backed system. Those benefits are action-driven benefits. Those benefits are created when the system takes action. Therefore, following the design part is categorised into different actions.

The system consists of three actions. The first phase is the stakeholder agreement. Homeowners and suppliers agree on a purchase order. This phase includes updating the list of suppliers, devices and items. The second phase is inventory managementing. At this phase, devices update their inventories and share it with other devices to determine when they need to purchase items. The last phase is inventory transaction. Devices broadcast purchase orders to suppliers and the transaction then occurs according to a predefined rule.

The stakeholder agreement and the inventory managementing phases are not expected to benefit from using a private blockchain instead of a centralised database. The third action, which involves a financial transaction, benefits from using the private blockchain backed system. This will be the focus of this system.

The design part will discuss various alternative solutions. This is as a result of a lack of previous related work. This report opens the possibilities of alternative solutions and suggests how it could be considered in different home inventory management scenarios.

4.2 System Structure

Eris-db is the private blockchain backed system in the Eris platform. Eris-db can be accessed through JavaScript applications. All agent has a copy of Eris-db. Eris-db stores contracts. An agent communicates to another agent through Eris-db, and the only way to access Eris-db is through contract. Therefore, agents communicates with other agents through contracts. Node-JS applications are used for communication between non-contract agents and contract agents. There are three types of Node-JS applications in this system, which are inventory control, system management, and suppliers applications.

The inventory control application will also update current inventory, and the centralised database system will be used.

4.3 Chain Design

Account type determines different levels of permission to access Eris-db. This needs to be well designed before running the blockchain as it will decide permissions of agents. Poorly designed chain could destroy the system, or even worse, it could lead to bankruptcy of the system's owner as the system contain financial information. Eris-db requires above 66.6% of bonded stake to be presented to create the next block. The system cannot be trusted if the homeowner has more than 66.6% of bonded stake. When the suppliers have more than 66.6% of bonded stake, the system could be unstable. It is because if some of the suppliers are offline, the system will be halted. Therefore, it is ideal if the suppliers and the homeowner share their bonds in the ratio of 50:50. In that case, none of the parties will have absolute power. The homeowner would require at least 32% of suppliers to co-operate with the system. This system can become safer if more suppliers join the system.

The homeowner will have 50% of the bonded stake. Some blockchain systems put all the bonded stake to the centralised cloud system that runs eris-db server for IoT devices. However, it is considered to be less robust as it leads to a single point of failure. Distributing bonded stake among the device agents will make the system more stable because it can still function even when the administrator agent is not working.

A developer account is needed when the smarthome owner needs support from external developers. The developer account will have all the permissions apart from the root permission. The root permission will only be granted to the administrator agent.

Suppliers and devices will have same level of permissions, but will be recorded with different type of accounts. This will allow administrator account to distinguish between suppliers and devices, so it can grant different permissions to access smart contracts.

Although number of bonded stake will be spread evenly for the devices. This can be changed depending on the network stability of the devices.

4.4 Smart Contract

Once contract is deployed into the blockchain, it cannot be modified. Therefore, it is critical to design a contract that is scalable. The contracts deployed on the blockchain will be visible to anyone who has access to the blockchain. Although it is possible to increase anonymity of deployed contracts using projects such as Hawk, it does not guarantee the perfect anonymity, or require unrealistically high cost computation [Buterin, 2016]. It is more practical to design a secure smart contract that does not have any loophole. The first subsections shows a design of basic PO contract and typical smart contract problems. With the analysis of weaknesses of the basic smart contract, advanced smart contract is designed. Three limitations of the basic contract are found, which are:

· lack of scalability

· security

· and efficiency.

In order for a smart contract to call another contract, it should have the source code in the ABI folder.

4.4.1 Basic PO Contract

Purchase order must contain item code, number of units, and price per unit. The basic contract can be written as *Device A purchases N item with quantity X with price K from supplier B.* This very basic purchase order can be used in smart home inventory management, yet it will face a few problems.

* Scalability issue: Smart home owner would like to update item(X), item quantity(N), or price(K).

* Efficiency issue: Smarthome owner would like minimise costs by sending contracts to multiple suppliers, not only to a selected supplier(B).

* Security issue: Device or supplier can trigger this contract without supplierâĂŹs agreement, and vice versa.

4.4.2 Advanced PO contract

Solution to scalability issue - PO manager The issue is that the basic PO contract will have fixed values for an item code, the quantity of the item and a price. However, users would like to create the purchase orders with different variables.

One solution is creating the contract that takes those three variables inside the constructor. So the users can create the contracts with an item code, the quantity of the item and a price. When a contract is created, they will have an address, and can be managed at the application level. It seems this solution could solve the scalability issue, yet it will not be scalable when this contract will contain complex logics. The PO contract will be accessed by at least three different agents, or could more than that if the system considers deliverer or other stakeholders in the future. Those three agents need to have different level of permissions to change the variables in the PO contract. There should be methods for the three agents to set and get the variables in the PO contract. Those methods should have different permission levels for different agents. There are two approaches to manage the permission level of different agents. First is managing the permission level in the smart contract, and another is managing them at the application level. The later method is more secure, which will be explained in the solution to security issue in the later part. If the system would like to manage the permission level in the smart contract, the permission level would need to be reset whenever a new contract is created. This is not a good idea, because same logic is being repeated again, which could have been simplified using the another solution based on the Factory Design Pattern.

In the second design solution, PO manager contract will create and manage the PO contract. PO manager is similar to the factory method in the Factory Design Pattern in Object Oriented Programming. PO manager will set permissions to all the users, and this contract will create PO contracts with flexible variables. PO manager will also manage the addresses of PO contracts using a map variable. The key of the map is the item code, and the value is the address of the PO contract.

Solution to efficiency issue The logic of the basic contract is simple, but it might be not efficient enough in the real world. The smart contract should try to reflect the real life situation. In case of purchasing inventory, the homeowner would likely visit a few stores before deciding to buy. There will be a situation when the supplier does not have the item. However, the supplier is assigned to the one specific entity in the basic contract. This can be extended to multiple suppliers, and this will allow user to make the purchase order more efficiently. This report suggests two ways to implement the logic to choose suppliers among the multiple suppliers that will increase efficiency and will reflect a more real behavior of the homeowners.

Method A is to keep all the verified suppliers' addresses at the application level, and iterate those addresses. Then, it will send the PO contract to the first one in the list, and if the supplier is unavailable, it will send it to the second supplier in the list. **Method B** is keeping all the inventory data in the purchase order, and allow the suppliers to access this contract. The purchase order will be available when the inventory level is low, and so one of the suppliers can take the purchase order. **Method C** is implementing the auction. When the inventory level is low, suppliers will be notified. Then, the suppliers can send their price to that opened purchase order. The device can close the auction after certain time period, then, the supplier with the lowest bid will take the purchase order.

Method A : *Smart home sends a signal to supplier S1 to purchase item A of N units with pre-agreed price X, if S1 is available, S1 will be paid X, and will need to send item A of N units. If S1 is unavailable, smart home sends signal to S2 and repeat the same process until SN, which is the last supplier in the supplier list.*

Method C : *Smart home starts an auction of item A of N units with the maximum price X, then suppliers sends input of X2. The supplier with the lowest input X2 will be paid X2 and will need to give item A of N units.*

The advantage of method A is that this contract represents the homeowner's priority to the suppliers. The homeowner can put the most favorable supplier on the first list. The homeowner may be more favorable towards supplier X because that supplier is capable of taking the social responsibility, or the homeowner just simply has a brand loyalty on supplier X. The advantage of method B is that the process is simple. Both the homeowner and the suppliers know the price of the deal. The advantage of method C is that the homeowner is likely to get the most cost efficient deal through the auction system. In addition, it's generally a reasonable decision to buy the cheapest offer amongst the same items.

The **method C** is chosen because it reflects the best real life situation amongst all others.

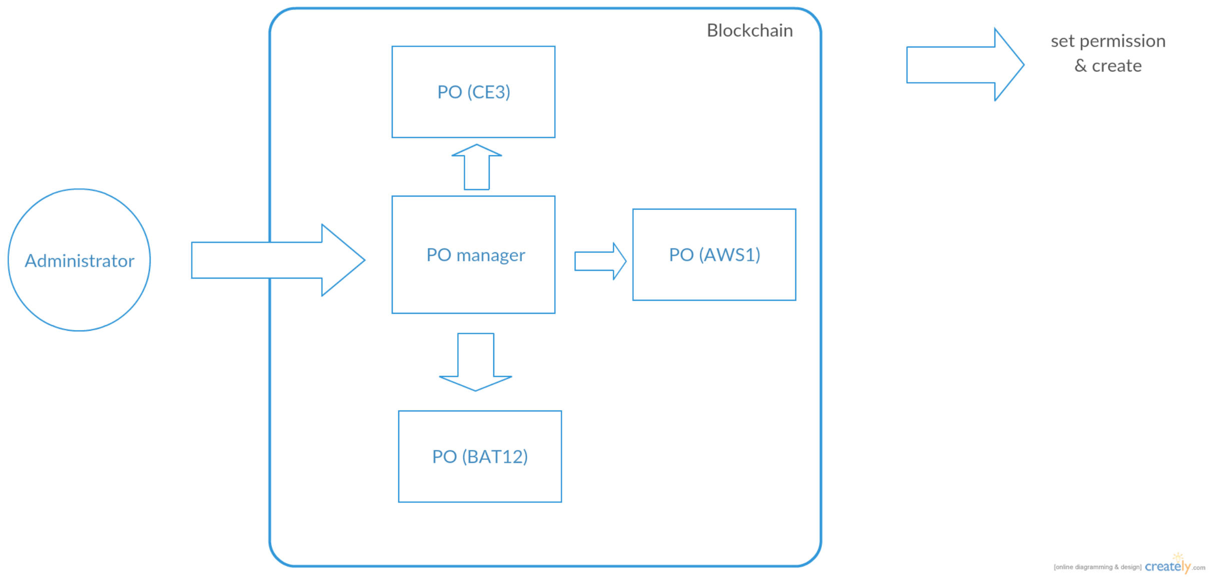
Solution to security issue - *setPermission* The PO contract could be hacked in two cases. One case is that one of the agent is being hacked, and the hacker, uses that agent to access and destroy the whole system. Another case is that one of the agent in the system abuses the system.

Setting the price should be the only database write method that the supplier can call, so it is generally safe even if the system is hacked by outside hacker. For instance, when the supplier agent is hacked, the worst thing that the supplier can do is setting the lowest price on the opened purchase orders. Even if the hacker hacked this system, the hacker will not receive the money from the transaction, unless the hacker also hacked the supplierâĂŹs bank account. If the administrator or the device agents are hacked, the system could be at risk. Still, there is no way that the hacker can directly benefit from this hacking.

However, the true security concern is arisen from the internal users. The suppliers and the homeowner should be able to fully trust this system. The suppliers can generally trust this system because the suppliers only send the items after they receive the cash they are promised to receive. However, if the opened PO contract is updated, the supplier may get a different result. For example, the supplier responds to the PO with 30 GBP for 10 units, but the owner can suddenly change the units to 30. There should be an instance variable that indicates the status of the PO. This will make sure the PO cannot be updated when the PO is opened.

The homeowner would like to make sure that suppliers cannot change any variable apart from the supplier's price. The homeowner do not want the suppliers to change the inventory data of the system nor other variables that are set in the PO, such as the item code or the price in the PO. Therefore, the homeowner should be able to set the permission level for the agents in the system to access the method in the PO. The first step to secure the PO contracts is making sure that PO manager contract is safe. There will be many PO contracts, but there will be only one PO manager contract. When the PO manager is created, it should set the owner of the contract as the administrator. This can be done by setting an instance address variable to the administrator address in the constructor. Since the constructor is only called once, therefore, once the address variable is set correctly, it cannot be modified. Then, there should be the *setPermission* method, which sets the permission level of all the agents. This method should only be accessible by the owner.

Figure 4.1: Dependency and permission relationship



Now, every method in the PO manager will be safe. However, the suppliers can still access to the PO contracts directly. Therefore, PO manager contract should send the address of itself (PO manager address) to PO so that PO can call PO manager to set the permission for the methods in PO.

Homeowner can set permission level of accounts, so it is possible that the homeowner creates fake supplier accounts to lower the price intentionally. However, repeatedly lowering price can be easily recognised by suppliers as the suppliers will have access to the transaction history, and will be able to recognise the transaction with the fake supplier.

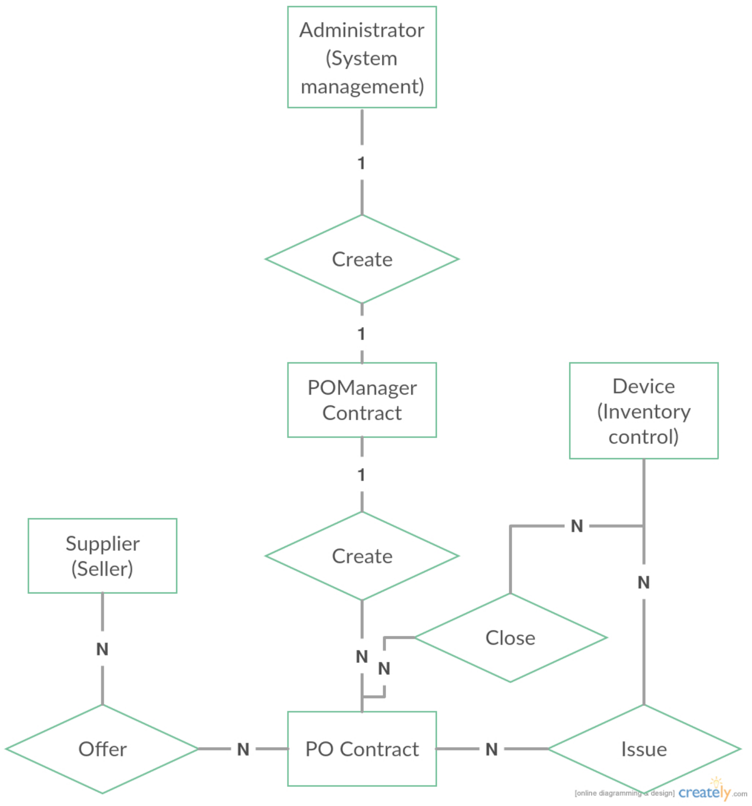
4.5 Applications

Applications are written in JavaScript to use Eris-db JavaScript libraries. These JavaScript applications communicate with the PO manager and PO contracts in Eris-db. These applications are running by different agents.

4.5.1 Entity Relationship Diagram

Figure 4.2 is the entity relationship diagram representing the relationship of agents around PO contract. The PO contract cannot be executed without any of the three entities, which are administrator, supplier, and device.

Figure 4.2: Entity relationship diagram


Administrator (System management)
1
Create
1
POManager Contract
1
Create
N
Device (Inventory control)
N
N
Close
N
Supplier (Seller)
N
Offer
N
PO Contract
N
Issue
[online diagramming & design] creately.com

4.5.2 System Management Application (Administrator)

The administrator creates the PO manager contract. Then, the administrator creates PO contracts that are corresponding to the items that need to be purchased through the system. PO contract will be identified by its item code. Therefore, the item code must be unique. The administrator will also set the two variables which are the maximum price for the item and quantity of the item to purchase in one order.

4.5.3 Inventory Control Application (Device)

When the inventory control application detects that the inventory level is low, the application will call the PO contract with the unique item code to open the PO. It is important to limit the supplier ability to open the contract. The way to prevent supplier's attack is discussed in 4.4, Smart contract section.

4.5.4 Seller Application (Supplier)

Once the device agent opens the contract, the seller application will be notified. Then, the seller application will respond to the PO contract, and will send the two variables: availability and the price. If the supplier sends the availability as true and the price is the lowest among other available suppliers, the supplier will need to start delivery. Upon the completion of passing the items to the shipping company, the supplier will receive the money that the customer had sent.

4.6 Stakeholder Agreement

Stakeholder agreement is the first step to initialise the smart home inventory management system. Once the initial stakeholder agreement is done, it can be updated later by an administrative agent.

Homeowner and suppliers are the stakeholders of this system. Homeowners are represented as devices in this system. This action is consisted of two actions:

∗ Controlling Eris-db permissions of stakeholders

∗ Controlling PO contract permissions of stakeholders.

Both of the actions must be done by the homeowner through an administrator agent. Generating or removing accounts in the system is a very critical function, so it is important to restrict this permission to suppliers.

4.7 Current Inventory Updating

When it comes to current inventory updating, recording inventory level in Eris-db might not be a good option. Firstly, it does not involve a third party when using a centralised database.

Secondly, current inventory data information may be useless in case all the devices are lost due to a disaster. Therefore, two benefits of using blockchain backed system are not valid in this case. Nevertheless, as all agents have a copy of Eris-db, using it as a current inventory information database may not charge extra cost or lower performance. Hence, this report is designed for both systems.

Centralised Database IoT devices send censored data to the administrator agent. Then, the administrator agent selects one device for one item, and sends back the updated quantity of item to selected devices.

Peer to peer update One of the effective way to handle inventory data is through peer to peer communication. Inventory data will be obtained from device's sensor, and will be stored in the device memory. Devices with same inventory will form a network, and they will select a leader to communicate with supplier. Then, all the devices will report their data to the leader device.

Smart contract storage as a database Deploying a contract that could store current inventory data could be the third option. This contract should track the quantity of items among the devices. However, this can be very inefficient as inventory updating could happen very frequently, and calling smart contract to write those data in eris-db would take too much overhead memories and computation.

It came to the conclusion that the first option is the most suitable solution at this stage for following reasons. This database will only be used within the home, there is less security concern. In addition, the centralised database is useful if the smarthome owner would like to utilise the data, since accumulated data could be used for various advanced technologies such as image data for Computer Vision and the voice data for Natural Language Processing.

4.8 Post Inventory Purchase

There are two aspects that must be considered into the system in order to achieve the fully functioning inventory management system. First is shipping the purchased inventory from the suppliers to the home. Second is the way to transact money.

4.8.1 Shipping

Standard purchase order should clearly indicate on shipping method. Few solutions are available.

The PO contract can be designed to lead to another shipping contract when the PO is closed. The shipping contract will select the most suitable shipping company for the product according to the logic indicated in the contract.

Alternatively, shipping companies can join the PO contract with different permission, and the PO contract can only be closed when at least one of the deliverer signed the contract, which means changing a variable in the smart contract to indicate that the deliverer can ship the products.

4.8.2 Payment

Banks can be involved in the PO contract as a stakeholder. Then, the PO can be designed to check from the bank if the homeowner is available to make the transaction. Then, the bank will call the PO contract when the transaction between the homeowner and the supplier is completed. The PO contract can be closed, when the transaction is successfully occurred.

Alternatively, when the transaction is confirmed, the PO will send a user's information to the bank according to the pre-defined rule in the PO. The PO will get the user's information from the administrator agent, and the PO can only be closed, when the bank transaction is successfully completed.

4.9 Dual Legal Engineering

Should have specification of item and transaction. This contract is as a natural language legal document, which represents how the logic supposed to work.

Chapter 5

Implementation

The design part describes how the software should be designed to achieve the requirements of the users, and this part shows how the designed system is implemented in the codes. This report also provides snippets of codes to help provide an explanation.

5.1 Agile Software Development

Agile methodology was chosen for a few reasons. Agile methodology will provide the opportunity to see how it works from the start, and I can make amendments throughout the development process. Secondly, there is no definite specification for developing the system for blockchain. Therefore, the requirements for the project could change as my knowledge level evolves. Lastly, Eris platform is currently under active development, and new features are updated every month. That is why it is better to leave the room for changes.

Each iteration includes full development cycle starting from requirements analysis to implementation to testing.

- First iteration: Development of a default permissioned Eris-db, a smart contract that stores a single variable, and a JavaScript application calls the smart contract.
- Second iteration: Development of PO contract and the system management application.
- Third iteration: Development of the Chains.
- Fourth iteration: Development of PO manager contract, and development of the inventory control and the seller applications.

Table 5.1: Chain setting

Permissions	Aministrator	Developer	Device	Supplier
Number of accounts	1	1	10	15
Number of bonded stake per account	5	0	1	1
Create_account	Available	Not Available	NotAvailable	NotAvailable
Send/Call contrat	Available	Available	Available	Available
setPermission (Contract)	Available	Not Available	Not Available	Not Available

5.2 Chain

Based on the design, the chain has been implemented in this system as follows (Table5.1). Number of bonds are spread out among the device agents who will ensure they can function when the administrator agent is not on the network. Supplier and shipping will have exact same permissions to the blockchain system, yet, they will have different permissions to smart contract, which will be discussed in the next section.

5.3 Setting the Permission To PO Manager

Three different contracts were implemented in the system; PO manager, PO, and Inventory contracts.

Admin instance variable will be initialised as the address of the administrator agent. This variable is initialised only once when it is created.

```
// POManager.sol
 function POManager(){
        admin = msg.sender;
    }
```

Following snippets is a modifier function. It will shorten the codes, and improve readability.

```
// POManager.sol
     modifier onlyAdmin {
        if (msg.sender == admin) // this ensures that only the owner can access the
             function
             _
    }
```

Then, the administrator agent sets the permissions for the users. *setPermission* method takes the inputs from JavaScript. The administrator agent calls *setPermission* method, and iterate through all the addresses of the agent.

```
// POManager.sol
mapping (address => uint) perms;
    function setPermission(address user, uint perm) onlyAdmin{
            perms[user] = perm;
    }
```

The *setPerms* is a JavaScript method written in the system management application. *accountData* is a map data. The name of agents are the keys in this map, and the addresses, and the public and private key are the values. Since it contains the private keys of the agents, this variable must not be used in the production.

The *setPerms* method iterate through the *accountData* map. Account name always contains account type in Eris. For instance, device accounts will have the following names *device001 device002*. Therefore, the type of the account could be identified using the regular expression.

```
// SystemManagement.js
var accountData = require('./accounts.json');
function setPerms(){
    for (var key in accountData)
    {
        var device = /device/
        var supplier = /supplier/
        var developer = /developer/
        var curAddress = accountData[key].address;
        if(device.test(key)==true){
            setPermission(curAddress,2);
        }
        if(supplier.test(key)==true){
            setPermission(curAddress,1);
        }
        if(developer.test(key)==true){
            setPermission(curAddress,0);
        }
        accountList.push(curAddress);
    }
}
```

Afterwards, the device, supplier, and developer agents will have different permissions to access the PO manager contract. Hence, it is safe to assume that all the methods in the PO manager can only be accessed by the agents with appropriate permissions.

5.4 Setting the Permissions To PO

Blockchain is transparent, and it means that any agents in the network can see the address of newly created PO contracts to directly access them. Setting the permissions to PO is quite challenging, as the permissions need to be set through the PO manager contract. This can be done in the following order: First of all, the PO manager should pass the address of itself to the newly created PO contracts. It is because the PO contracts need to get the list of agents' permissions from the PO manager. *this* variable in the Solidity refers to address of itself. *PO(cur).setPOMAddress(this);* This line needs to be executed when the PO manager creates a PO contract.

```
// POManager contract
   address POMAddress;
   function setPOMAddress(address q){
      POMAddress = q;
   }
```

Then, this modifier method in the PO contract will ensure the permission level.

```
// PO contract
modifier onlyDevice {
      uint permL = POManager(POMAddress).getPermission(msg.sender);
       if(permL==2)
          _
   }
```

5.5 Setting Up PO Contract

PO manager creates PO contracts. *createPO* method takes item code, price, and the quantity of items as a parameter. The address of newly created PO will be saved as a value in the instance map *nameReg*, and the key of this map is the item code. When the administrator creates a contract with the item code that is already existing, the event will be fired to warn the administrator.

```
// POManager contract
    mapping (bytes32 => address) nameReg; // This must not be exposed.
    event itemAlreadyExist(bytes32);

   function createPO (bytes32 itemcode, uint price, uint orderQuantity) onlyAdmin {
        address cur = nameReg[itemcode] ;
        if(cur != 0x0){
            address newDeal = new PO(itemcode);
            nameReg[itemcode] = newDeal ;
            PO(cur).setPOMAddress(this);
            PO(cur).setMaxPrice(price);
            PO(cur).setQuantity(orderQuantity);
        }else{
            itemAlreadyExist(itemcode);
        }
    }
```

The PO contract can also be removed.

```
// PO contract
function remove() onlyDevice{
        if(msg.sender == admin){
            selfdestruct(admin);
        }
    }
```

5.6 Purchasing Inventory

In order to purchase an item *battery*, the PO contract with the item code *battery* needs to be signed by both the suppliers and the device. The signing process is as follows:

1. The administrator creates the *battery* contract.
2. The device opens the *battery* contract.
3. The suppliers set the price of the *battery* contract.

4. The device closes the *battery* contract.

5. The transaction will be carried out.

The device calls the *POOpen* method which will change the *isOpen* variable to 1 in the PO contract. *isOpen* variable represents the status of the PO contract. It is 0 when the PO is first initialised. The integer type is used, instead of the boolean type, since the PO contract might need to be implemented with a different level of status in the future.

```
// POManager contract


  function POOpen(bytes32 itemcode) onlyDevice{
                PO(nameReg[itemcode]).open();
     }
    }
```

```
// PO contract
PO {
 function open() onlyDevice {
       notifyOpen(Itemcode,maxPrice,quantity);
       isOpen = 1 ;
    }
}
```

The suppliers can subscribe to the *notifyOpen* event of the PO. Eris also supports filter method, and this can help the suppliers to listen to the events they want. However, the event subscription method was not implemented in the prototype, as event method was not working due to an issue. This issue will be mentioned again in the later part of the implementation.

Instead, the following interface methods were implemented. It clearly describes how the suppliers can listen to the events, and set the price in the PO contracts.

```
// Seller.js


var supplierItems = {}; // item name and the quantity of item
var itemPrice = {}; // itemname (key) -> price
var subscribedEvent = {}; // itemcode (key) -> event address (value)


function getEventDemo(){
   // subscribe the events of the item that the supplier currently have.
   for (var key in supplierItems)
    {
        // subscribedEvent[key] = subscribeEvent(key);

    }
}

function listenToEventDemo(){
   // subscribe the events of the item that the supplier currently have.
   for (var key in subscribedEvent)
    {   var isOpen = subscribedEvent[key];

        if(isOpen == true) {
           //price
           setSupplierPrice(key,itemPrice[key]);
        }

    }
}
```

This is the *responePO* method, which was implemented in the PO manager contract.

```
// POManager contract

function responsePO(bytes32 itemcode, uint price) onlySupplier {
       PO(nameReg[itemcode]).setSupplierPrice(price,msg.sender);
    }
```

The *responsePO* method will call *setSupplierPrice* method in the PO contract. Only the address of the caller with the lowest price will be saved in the PO contract. The following is the implementation of the auction system.

```
// PO contract

    address selectedSupplier;
    uint lowestBid;
    function setSupplierPrice(uint price, address sender){
        if(price<lowestBid && price<maxPrice){
            selectedSupplier = sender;
            lowestBid = price;
        }
    }
```

The device can close the PO contract by calling the following method. The close method handles the exception case when the *selectedSupplier* is not assigned. This will happen when none of the suppliers respond to the PO with the price lower than the *maxPrice*.

```
// PO contract
function close() onlyDevice {
        if(selectedSupplier != 0x0){
        purchaseConfirmed(selectedSupplier,lowestBid);
        //remove(); // remove the deal after transaction
        }
        else{
        purchaseFailed();
        }
    }
```

5.7 Automating the Purchasing Process

In order to automate the purchasing processes for the homeowner, which are described in 5.6, two types of data are required. First is the current inventory information, and this should be obtained from the sensor of the device. Second is the conditions and how to open the purchase order. For example: *The system should have more than 15 units of battery. When it is lower than that, the system should make a purchase order of 30 units of the item. The maximum price willing to pay for this item is 3 GBP per unit, and the purchase order should be opened for 20 minutes.*

The statement can be written as *createInventory(battery, 15, 20, 30, 3).*

```
// Inventory contract


 mapping (bytes32 => Inv) invMap;
```

```
    function createInventory(bytes32 itemcode,uint minQuant, uint opentime, uint
        orderQuant, uint price) onlyAdmin {
    uint totalprice = q*p;
    invMap[ic] = Inv(itemcode,minQuant,opentime,orderQuant,price,totalprice);
}
```

Struct data structure was implemented to contain the data above. This struct data type will be saved into a map, and the key of this map will be the item code.

```
// Inventory contract
struct Inv{
    bytes32 Itemcode;
    uint minQuant;
    uint PO_time;
    uint Quantity;
    uint Price;
    uint TotalPrice;
    }
```

Then, the device can get the conditions with the âĂIJgetInvRuleâĂİ method.

```
// InventoryControl.js
var inventoryMap = {};
function getInvRule(itemcode) {
  invContract.getInv(itemcode,function(error, result){
    if (error) { throw error }
    console.log("Getting the deal " + result );
    inventoryMap[itemcode] = result[1].toString();
  });
}
```

5.8 Limitation

There are 3 functional requirements that were not successfully implemented, because some of the features were not working in the Eris platform at the time [15th of April].

- The homeowner can create all type of agents in the system.

- The homeowner can remove any agents with the exception of the administrative agent, in the system.

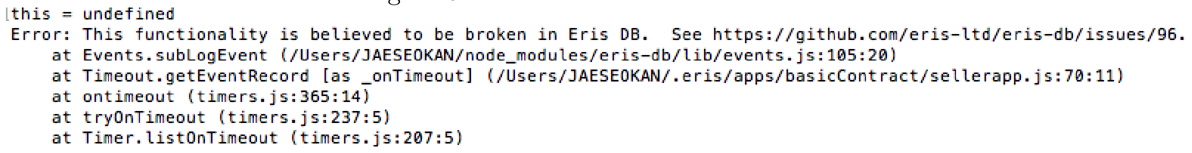
Generating and removing the agents through Javascript application has not been implemented in Eris platform yet. This feature is expected to be implemented in the language in the future [*Generate account*, n.d.*a*]. For this reason, these two functions were only designed in the system managing application, but not implemented.

```
// SystemManagemet.js

function generateAccount(accountType) {
  purchaseContract.generateAccount(accountType,function(error, publicKey, privateKey){
    if (error) { throw error }
    console.log("Generating the account with a public key of " + publicKey);
  });
}
```

Smart contract can write a message on the log through firing an event. The suppliers are supposed to listen to the *POopen* event, so that the suppliers can know that the PO contract is open. This functional was not able to be implemented, as the event methods in the Eris was not available at the time of development. [1]

Figure 5.1: Event methods issue



```
this = undefined
Error: This functionality is believed to be broken in Eris DB.  See https://github.com/eris-ltd/eris-db/issues/96.
    at Events.subLogEvent (/Users/JAESEOKAN/node_modules/eris-db/lib/events.js:105:20)
    at Timeout.getEventRecord [as _onTimeout] (/Users/JAESEOKAN/.eris/apps/basicContract/sellerapp.js:70:11)
    at ontimeout (timers.js:365:14)
    at tryOnTimeout (timers.js:237:5)
    at Timer.listOnTimeout (timers.js:207:5)
```

Eris is currently not supporting deploying the Eris platofrm in IoT operating systems yet. It will be available in ARM devices in the future. [2]

[1]https://github.com/hyperledger/burrow/issues/96
[2]https://github.com/monax/cli/issues/1088

Chapter 6

Professional and Ethical Issues

This project did not require any sensitive personal data. The project was built on the open source projects, and it is important to aware the responsibility of using the open source. Especially, I should be aware of licensing and plagarism issues.

6.1 Plagarism & Licensing

The system was implemented using various open-source projects. Licensing of those projects were thoroughly examined before using them. The system is built on Eris[1], which is an open platform. As a part of the system, Tendermint [2], Etereum Project [3], Node-JS [4], and many other open source libraries were used.

All the resources that I have referred for this report is cited accordingly.

6.2 British Computing Society

British Computing Society provides standardised guidelines for computing professionals.[5] [6] All guidelines was considered respectfully throughout the project. During the development, the open source seemed to have few issues. I took the responsibility by raising the issues. [7]

[1]https://monax.io/
[2]https://tendermint.com/
[3]https://www.ethereum.org/
[4]https://nodejs.org/en/
[5]http://cgi.csc.liv.ac.uk/ comp39x/2008-09/BCS_CoC.pdf
[6]http://www.bcs.org/upload/pdf/cop.pdf
[7]https://github.com/monax/legacy-db.js/issues/49

Chapter 7

Testing

In the agile software development life cycle, testing needs to be done in every iteration. Testing was done promptly after the implementation in every iteration. After each implementation, different functionalities have been tested to ensure this prototype had developed as expected. Another purpose of testing is to ensure the knowledge that I have acquired in the paper and the documentation are correct.

7.1 Unit Testing During the Agile Implementation

7.1.1 First iteration

The goal of the first implementation is to write anything in the permissioned blockchain through a smart contract. Two criteria was used to test this. Agent A had a full permissions to access the blockchain. Agent B did not have a permission to write on the blockchain.

- Agent A writes an integer variable by calling a smart contract, and checks if using the agent A and B if it is correctly written.
- Agent B writes an integer variable by calling a smart contract that agent A used, and check if using A and B if agent B did not make any changes.

7.1.2 Second iteration

The second iteration was done to ensure the methods that need to be implemented in PO contract are functionable. Agent A with a full permission was used, as validity of permission has been confirmed after the first iteration. Agent B also has a full permission.

- Agent A writes integer variables and characters that could represent the purchase order in the inventory management system.
- Agent A updates the variables in the contract.
- Agent A sets the permissions to update the variables in the contract, and limits this method to the agent A itself. The permissions of the agent B were tested.

7.1.3 Third iteration

The third iteration was done to test the functionality of Eris Chains.

- Creates the chains with supplier, device, and administrator.
- Use system management application to recognise different types of agents.

7.1.4 Fourth iteration

In order to test, the following criterias were used:

- Administrator agent creating multiple contracts containing different variables
- 3 suppliers input different price, and return the address of the supplier called with the lowest price.
- Check every method in both PO and PO manager, and if those methods can be called by the agents without the permissions. The detail of this is mentioned below.

7.2 Unit Testing

The table 7.1 and 7.2 are the record from the unit testing. In addition to this, *setPermission* method was tested. Only administrator agent was able to run the *setPermission* method.

7.3 Scenario testing

Scenario test is software testing based on the scenarios. The result of the scenario test is very useful for evaluation. The aim of the project is evaluating the usefulness of the private blockchain, and this scenario test will show if this system could function without a third party.

Table 7.1: Unit test : createPO

Itemcode	**Price input**	**Quantity input**	**Price expected**	**Quantity expected**	**Test result**
T1	1	1	1	1	pass
T2	2.5	2.5	2	2	pass
T3	0	10	Contract should not be created		pass
T4	-1	0	Contract should not be created		pass

Table 7.2: Unit test : setSupplierPrice

Itemcode	**Price input**	**First supplier input**	**Second supplier input**	**Expected output**	**Test result**
T5	10	8	5	5	pass
T6	10	5	8	5	pass

Scenario was carefully designed to test the usefulness. The purpose of this test is to test the logic, so the minimum number of agents and variables were used. The system was tested with 1 homeowner, 1 device, and 2 supplier agents. The 4 agents executed their applications separately on the same virtual machine.

The homeowner sets the battery purchase order, and sets the rule for device to manage the inventory. *createPO(battery,10,5)* and *setInvRule(battery,20,24,5,10)).* Both of the suppliers, A and B, have enough amount of inventory, and they are willing to offer 5 units of battery with 5 gbp and 7gbp respectively.

The supplier agents were executed manually to respond to the PO, as the subscribeEvents methods are not working due to the platform issue. The test was done using the POManagerTest contract, which is designed to test logics. Following is the result from the scenario test.

- The device opened and closed the PO contract.
- Transaction happened between the device and the supplier A.
- The A battery PO should be closed after the transaction.

From this result, the report can conclude that the core functionality of the system is working.

Chapter 8

Evaluation

This section evaluates the functional requirements by comparing it to the results obtained from testing. Then, it will evaluate the non-functional requirements. From the evaluation of these two, this report will determine to what degree this system achieve the main objective. This report shows two alternative designs for the PO contract, and one alternative design for the managing current inventory data. All functional requirements could be satisfied using the alternative designs. However, alternative designs would fulfill different non-functional requirements.

8.1 Functional Requirements

Functional requirements are the core functions for the system, and these could be evaluated through testing. The following functional requirements were successfully implemented and tested.

The system has different permission levels to access Eris-db and smart contracts for different agents.

- This requirement was satisfied using the Eris Chains module. It was tested through getPermission method in PO manager. It was also tested in the scenario testing.

Homeowners can create a purchase order, which indicates an item code, the quantity of the item and the maximum price one is willing to pay for the items.

- This function was implemented in the PO manager contract, namely *createPO*. This method can be accessed through createDeal method in the system management application. This method was unit-tested, and the result of this is described in the table 7.1.

Homeowners can set the rules on when the device agent can open and close the purchase order.

- This function was implemented in the Inventory contract, namely createInventory.

A device agent opens and closes the purchase order according to the rules that are set by the homeowners.

- This function was implemented in the PO manager contract, namely *POOpen* and *POClose*. This method can be accessed through *openPurchaseOrder* and *closePurchaseOrder* in the inventory control application. *checkCondition* method in the inventory control application will compare the inventory level with the conditions, and *openPurchaseOrder* method will be called automatically.

A device agent can obtain inventory data from its sensor

- This function was implemented in the inventory control application, namely *getInventoryData*. This method gets the inventory data.

A device agent can share the inventory data to other devices.

- JSON format file was used as a database in this system. Although, it could show how the current inventory data could be managed, it needs to be implemented with centralised database, such as SQL, in the production.

Suppliers can respond to purchase orders automatically.

- This was implemented in the seller application. The supplier would keep the available inventory list with the quantity and the price.

Suppliers can set the price they are willing to offer for the corresponding purchase order.

- This function was implemented in the PO manager contract, namely *responsePO*. This method can be accessed through setSupplierPice in the seller application. This method was unit-tested, and the result of this is described in the table 7.2.

Purchase orders should have legal power.

- This could be achieved through dual integration.

Purchase orders can be re-opened.

- This function was implemented in the PO contract. The integer variable named isOpen will indicate the status of the PO contract. PO contract will be set as *close* when it is first created, and will be changed to *open* when the device opens the PO. It will be reverted to *close* after the transaction. In this way, PO contract does not need to be created again, which would have taken up the memory in the blocks.

Once a purchase order is opened, the quantity and the price cannot be updated.

- The isOpen variable indicates the status of the PO contract, and the *setQuantity* and the *setMaxPrice* methods in the PO contract cannot be modified when the current status of the PO is open.

The logic of purchase order cannot be changed.

- *Device A purchases N item quantity X with price K from supplier B.*The purchase order logics are strictly saved as codes in the PO contract. This cannot be modified in any case.

Interface of the following 3 requirements were designed in the applications. Although these requirements were not implemented in the applications, those applications could be evaluated through checking the expecting effect of the functions.

- Homeowners can create all types of agents in the system.
- Homeowners can remove any agents, except the administrative agent, from the system.
- Suppliers should be notified when the purchase order is opened.

Functional requirements represent the user stories. Therefore, the successful implementation of the functional requirements mean that the user could use this system as they want. All the functional requirements were developed, and most of them were implemented. Hence, the system is a prototype showing the inventory management system logics.

8.2 Non Functional Requirements

Security, scalability and usability of the system are heavily affected from the design of the smart contract.

Security Eris-db, which is the database of this system, does not allow unauthorised users to access this database. Though the authorised users can damage the systems, the system is designed to minimise the effect. The PO and PO manager contracts are possibly vulnerable, and are evaluated below.

- The PO manager contract has a permission layer, and this permission can only be set by the administrator node. This permission layer keeps away unauthorised agents to access the smart contract method.
- The permission layer of the PO contract will be set by the PO manager contract.
- While the PO contract is opened, the variables cannot be changed.
- Any agents in the system can see the PO and PO manager contracts.
- The homeowners can possibly set a fake supplier agent to manipulate the price, but this can be seen by the suppliers as the blocks are transparent.
- The system cannot function if more than 66.6% of suppliers are not on the system.
- The PO contract can be legally binding through dual engineering techniques, hence, the suppliers and the homeowners can claim their rights according to the PO.
- Overall, it is more secure than the centralised database system, which can replace the current electronic or paper purchase order.

Scalability

- The PO contract can represent every item code through Bytes32 type variable, which could represent much more than billions of different combinations of items.
- The PO contract can represent the price of the items with integer values.
- The PO contract can represent different units of items.
- The system is designed to use the centralised database for managing the current inventory. It can be scaled easily in comparison to using the blockchain.

Usability

- The auction system mimics the economic activity in real market place.
- Alternative design of smart contract *(method A)* could be more useful to some of the homeowners.
- It takes average in X time and expect to cost less than Y, which can be used in normal smart home.

The system should be decentralised.

- A copy of Eris-db will be distributed to every agent in the system.
- Bonded stakes will be distributed among the suppliers and the devices. The homeowner will have 50% of the bonded stakes, and one supplier will have (50n)%. The validation ability is decentralised, but it is not perfectly distributed.
- Bonded stakes will be distributed among the devices.
- The purchase transaction occurs directly between a supplier and a device.
- Since the homeowner can set the permissions in the system, the system is partially under the control of the homeowner. However, the homeowner does not have the authority without the consensus of the suppliers.
- The system can be even more decentralised using the blockchain or peer-to-peer communication in the managing current inventory information. However, it will significantly decrease the efficiency and scalability. The alternative solutions cannot share large data, and will cost significantly more. In conclusion, this system is more decentralised comparing to the centralised database system. Nevertheless, the homeowner has more power than other stakeholders.

8.3 Aims and Objectives

The purposes of developing the private blockchain-backed smart systems are as follows:

Reducing the dependence on intermediaries The current cloud inventory management system would require the following intermediaries:

- The inventory management system service provider
- Intermediaries between the suppliers and the service provider. (this number can vary)
- A bank (the service provider will have the homeowner's bank information)

On the other hand, this prototype would require the following intermediaries:

- A bank (can be further replaced to cryptocurrencies)

It is clear that this system is highly independent. It is because of the direct communication between the suppliers and the devices. Less number of intermediaries will be involved in this system, and the dependency could be further reduced by integrating technology such as Bitcoin or supplier verification system.

Shorten processing time and verification cost The current cloud inventory management system would take the following processes. This assumes that the inventory management system has the lists of the suppliers, and can make purchases orders to them.

1. The hub inside the home collect the inventory data and send the data to the inventory management system service provider.
2. The service provider verifies the inventory data.
3. When the inventory level is low, the service provider sends purchase orders to the suppliers.
4. The suppliers and the homeowner signs to the purchase orders.
5. The service provider contacts the bank to transact money to the intermediary, and the intermediary transacts money to the supplier.

On the other hand, this system would take the following processes:

1. The devices send the purchase orders to the suppliers when the inventory level is low.
2. Upon the agreement of the suppliers, the system will contact the bank to transact money to the suppliers.

Hence, this system reduced the inventory purchasing processes using the PO contract.

Data ownership The owner of this system will have a full control of data from the IoT sensors. However, the inventory purchasing history will be recorded in the system, and will be shared amongst the suppliers. This can be further shared with other stakeholders, such as a shipping company.

Bottleneck One of the most significant bottleneck in the centralised system occurs to the inventory management service provider. It is because a countless number of smart homeowners will be connected to one inventory management service provider. This system decentralised this communication.

No single point of failure The most frequent single point of failure of the centralised system is the inventory management service provider. This system clearly overcomes this limitation.

Although the total number of single point of failures are reduced in the system, there are still a few single point of failures. The administrator agent could control all the permissions to the system. It means that when the homeowner loses the administrator agent keys, the system will become useless. In addition, the IoT devices will need the cloud service to be connected, and the system will fail if that cloud service is stopped.

Chapter 9

Conclusion and Future Work

The project aims to develop the prototype of private blockchain-backed system for the smart home inventory management. This prototype demonstrates the benefits of the private blockchain system over the cloud centralised system. This project also shows trade-offs of various smart contract and IoT network designs.

9.1 Influences on the Field

The PO contract demonstrates an original design of smart contract. The PO manager contracts sets up the permission levels to access methods in the PO contract. Multiple stakeholders are participating and expressing their offers on the PO contract by setting the variables. The system specifically shows a use case of private blockchain in smart home inventory management. This system could be further expanded into inventory management in other organisations such as factories, shops and libraries. For example,

- The factory owner A would like to send the purchase orders to multiple companies.
- The companies can easily join the smart contract by changing the variables with their own address.

The evaluation sections shows that the practical private blockchain system can be developed. This system can overcome the limitations of current centralised database model. This system will have less intermediaries, will have less bottlenecks, and will be more robust. Further benefits of system is also observed. The system owner can have the data ownership and the transaction process will be shorten with significantly less cost. Hence, this report presents the usefulness of the private blockchain for the IoT network.

9.2 Future Work

Disintermediation Two essential stakeholders need to be involved in this system in order to set up the automated inventory management system without any intermediaries. The deliverer is needed to be considered in the future work. Moreover, the system would become more secure as more diverse stakeholders join this system. It is because of the proof-of-stake system where the participants own the power to validate the data. However, the smart contracts must be designed even more robustly. The bank needs to be presented in the contract to confirm that the homeowner is capable of executing the purchase order, and to transact the money according to the smart contract. Alternatively, the cryptocurrency can be used. Bitcoin is one of the good examples. The system could also have its own cryptocurrency.

Privacy All transaction history will be recorded in the system, and every participant can access to this data. There will be even more private data in the real purchase order contracts, such as the home address. There are different approaches to this, such as HAWK or Coinjoin.

Big data The data from the IoT sensors could be utilised in a great degree with the Machine Learning technology. This data could be further used to smartly decide prices when purchasing items.

Testing The design suggested in this project has only been tested in a limited environment. The testing should be done in various operating systems, including the one designed for the IoT devices.

References

Ammous, Saifedan and Hisham [2016], 'Blockchain technology: What is it good for?', *SSRN* .

Buterin, V. [2014], 'Ethereum White Paper', `https://github.com/ethereum/wiki/wiki/White-Paper`. [Online; accessed 2017-03-10].

Buterin, V. [2015], 'On Public and Private Blockchains', `https://blog.ethereum.org/2015/08/07/on-public-and-private-blockchains/`. [Online; accessed 2017-03-10].

Buterin, V. [2016], 'Privacy on the Blockchain', `https://blog.ethereum.org/2016/01/15/privacy-on-the-blockchain/`. [Online; accessed 2017-03-20].

Conoscenti, M., Vetrò, A. and Martin, J. [2016], 'Blockchain for the internet of things: a systematic literature review', *IEEE* .

Dorri, A., Kanhere, S., Jurdak, R. and Gauravaram, P. [2017], 'Blockchain for iot security and privacy: The case study of a smart home', *IEEE PERCOM WORKSHOP ON SECURITY PRIVACY AND TRUST IN THE INTERNET OF THING* .

Ericsson [2016], 'Internet of Things forecast', `https://www.ericsson.com/res/docs/2016/ericsson-mobility-report-2016.pdf`. [Online; accessed 2017-03-25].

Eris(Monax) [2017], `https://monax.io/`.

Generate account [n.d.*a*], `https://github.com/monax/cli/issues/1008`. [Online; accessed 2017-03-27].

Generate account [n.d.*b*], `https://github.com/hyperledger/burrow/issues/968`. [Online; accessed 2017-03-27].

Greenspan, G. [2008], 'Blockchains vs centralized databases', `http://www.multichain.com/blog/2016/03/blockchains-vs-centralized-databases`. [Online; accessed 2017-03-14].

Gubbi, J., Buyya, R., Marusic, S. and Palaniswami, M. [2013], ‘Internet of things (iot): A vision, architectural elements, and future directions’, *Future Generation Computer Systems* .

Heath, A. [2016], ‘Jarvis’, `http://uk.businessinsider.com/how-mark-zuckerberg-personal-smart-home-assistant-jarvis-works-2016-12`. [Online; accessed 2017-03-20].

JavaScript [n.d.], `https://www.javascript.com/`.

Lesonsky, R. [1998], ‘Tracking Inventory’, `https://www.entrepreneur.com/article/21852`. [Online; accessed 2017-02-10].

Meulen, R. and Rivera, J. [2014], ‘Gartner Special Report Examines Trends in Digital Technologies’, `http://www.gartner.com/newsroom/id/2839717`. [Online; accessed 2017-03-12].

Monax [n.d.*a*], ‘Documentation — Blockchain Client’, `https://monax.io/docs/documentation/db/`. [Online; accessed 2017-03-02].

Monax [n.d.*b*], ‘Explainer — Blockchains’, `https://monax.io/explainers/blockchains/`. [Online; accessed 2017-03-12].

Monax [n.d.*c*], ‘Explainer — Chain Making’, `https://monax.io/docs/chain-making/`. [Online; accessed 2017-03-12].

Monax [n.d.*d*], ‘Explainer — Permissioned Blockchains’, `https://monax.io/explainers/permissioned{\_}blockchains/`. [Online; accessed 2017-03-12].

Monax [n.d.*e*], ‘Explainer — Smart Contracts’, `https://monax.io/explainers/smart{\_}contracts/`. [Online; accessed 2017-03-12].

Nakamoto, S. [2008], ‘Bitcoin: A Peer-to-Peer Electronic Cash System’, `https://bitcoin.org/bitcoin.pdf`. [Online; accessed 2017-03-10].

Pilkington, M. [2015], ‘Blockchain technology: Principles and applications. research handbook on digital transformations’, *SSRN Available at SSRN: https://ssrn.com/abstract=2662660* .

Ricquebourg, V., Menga, D., Marhic, B., Dealahoche, L. and Logé, C. [2006], ‘The smart home concept : our immediate future’, *IEEE* .

Solidity [n.d.], `https://solidity.readthedocs.io/en/develop/`.

Soliman, M., Abiodun, T., Hamouda, T., Zhou, J. and Lung, C. H. [2013], Smart home: Integrating internet of things with web services and cloud computing, *in* '2013 IEEE 5th International Conference on Cloud Computing Technology and Science', Vol. 2, pp. 317–320.

Szabo, N. [1998*a*], 'Bit Gold', `http://nakamotoinstitute.org/bit-gold/`. [Online; accessed 2017-03-01].

Szabo, N. [1998*b*], 'Secure Property Titles with Owner Authority', `http://nakamotoinstitute.org/secure-property-titles/`. [Online; accessed 2017-03-01].

Tapscott, D. and Tapscott, A. [2016], 'The blockchain revolution: How the technology behind bitcoin is changing money, business, and the world', *ISBN 978-0670069972* .

Thompson, C. [2016], 'Introduction to a Private Blockchain', `http://www.blockchaindailynews.com/Private-Blockchain-or-Database-Whats-the-Difference{\_}a24596.html`. [Online; accessed 2017-03-15].

Unknown [2017], 'Blockchain as a P2P protocol for the Internet of Things', `http://www.the-blockchain.com/docs/Blockchain{\%}20as{\%}20a{\%}20P2P{\%}20protocol{\%}20for{\%}20the{\%}20Internet{\%}20of{\%}20Things.pdf`. [Online; accessed 2017-03-12].

Wöner, D. and Bomhard, T. V. [2005], 'When your sensor earns money: Exchanging data for cash with bitcoin', `http://cocoa.ethz.ch/downloads/2014/08/1834{\_}sample.pd`. [Online; accessed 2017-03-01].

Ye, X. and Huang, J. [2011], A framework for cloud-based smart home, *in* 'Proceedings of 2011 International Conference on Computer Science and Network Technology', Vol. 2, pp. 894–897.

Zhang, Y. and Wen, J. [2015], An iot electric business model based on the protocol of bitcoin, *in* '2015 18th International Conference on Intelligence in Next Generation Networks', pp. 184–191.