KING'S College LONDON

University of London

6CCS3PRJ

Blockchain-based Access Point Channel Allocation Implementation

Final Project Report

Author: Vlad-Stefan Mititelu

Supervisor: Dr. Mischa Dohler

Student ID: 1739372

July 24, 2020

Abstract

With increasing usage of WiFi technology throughout the last two decades, the 2.4 GHz unlicensed band has seen more inefficient usage of the limited number of channels available. Out of these, only channels 1, 6 and 11 are non-overlapping. The paper looks to implement a blockchain-based system i.e. a DApp for automatic channel allocation for the managed a group of three Access Points in an Extended Area Set. The implementation process is debated with multiple approaches considered. The evaluation of the chosen implementation is further scrutinised and tested. Future work is considered about expanding functionalities to other aspects of Access Point management. The conclusion looks to argue the use fullness of a Blockchain Decentralised Application approach for managing access points.

Originality Avowal

I verify that I am the sole author of this report, except where explicitly stated to the contrary. I grant the right to King's College London to make paper and electronic copies of the submitted work for purposes of marking, plagiarism detection and archival, and to upload a copy of the work to Turnitin or another trusted plagiarism detection service. I confirm this report does not exceed 25,000 words.

Vlad-Stefan Mititelu

July 24, 2020

Acknowledgements

I would like to thank my project supervisor, Dr. Mischa Dohler for his help in providing me the resources necessary to complete the project, for his quick responses and for the opportunity to work on the chosen project.

Contents

Abbreviations

- **WLAN**: Wireless Local Area Network
- **ESS**: Extended Service Set
- **AP**: Access Point
- **P2P**: Peer-to-peer
- **PoA**: Proof-of-Authority
- **PoW**: Proof-of-Work
- **PoS**: Proof-of-Stake
- **TPS**: Transactions per second
- **EVM**: Ethereum Virtual Machine
- **pBFT**: Practical Byzantine Fault Tolerance
- **DCA**: Dynamic Channel Assignment
- **ISM**: Industrial, Scientific and Medical
- **CSMA/CA**: Carrier Sense Multiple Access/Collision Avoidance
- **DLT**: Distributed Ledger Technology
- (SINR): Signal-to-Interference-and-Noise Ratio
- **IEEE**: Institute of Electrical and Electronics Engineers
- **EIP**: Ethereum Improvement Proposals
- **RF**: Radio Frequency
- **DSSS**: Direct Sequence Spread Spectrum

Chapter 1

Introduction

1.1 Motivation and Background

Blockchain is one of the most enthused about technology trends. Blockchain became a popular trend following the release of the white paper "Bitcoin: A Peer to Peer Electronic Cash System" by anonymous author Satoshi Nakamoto in 2008. After the success of the Bitcoin cash system, the focus shifted on to the potential of the technology outside of the world of cryptocurrencies. Organizations have found several use cases for the technology. According to a 2019 survey by Statista [1], the top three most popular uses for the technology are data validation, data access/sharing and identity protection. The project looks to implement a system that would solve a problem from the field of WiFi Access by exploring the usability of the decentralized and programmable nature of the blockchain technology.

IEEE 802.11 WLAN is the family of standards that specify the media access control (MAC) and physical layer (PHY) protocols for the implementation of a Wireless Area Network (WLAN). The IEEE 802.11 WLAN connects 2 or more devices capable of wireless transmission using distribution methods; the most common are OFDM and spread spectrum (DSSS). The IEEE 802.11b standard and other similar ones came to be reffered to as WiFi. WiFi technology became popular due to its usage of the unlicensed frequency spectrum (ISM). In recent years, WiFi LANs have been rapidly deployed in homes, enterprises and public areas such as coffee shops. Infrastructure WLANs are used to grant client devices access to the internet through an access point. WiFi operates in two frequency bands: 2.4 GHz and 5 GHz. The 2.4 GHz band has been in use in WiFi standards for a longer period of time than the 5 GHz one, with more users and better area of coverage. The 2.4 GHz band has 14 channels for operation. Depending

on the country, the number of channels that may be used vary. In the United Kingdom, only the first 13 channels are available for use. Channels are spaced apart by 5 MHz except for a larger gap of 14 MHz between channels 13 and 14 as can be seen in Figure 1.1. A channel is said to be 20 MHz wide with approximately 2 MHz of additional width of acting as a channel separator. Out of all the channels available normally in the 2.4 GHz band, only channels 1, 6 and 11 do not overlap. Their center frenquencies are 2.412, 2.437 and 2.462 respectively. The most prevalent problem caused by the small number of non overlapping channels is interference. The problem of interference for the 2.4 GHz band can not be resolved entirely, but it can be reduced.

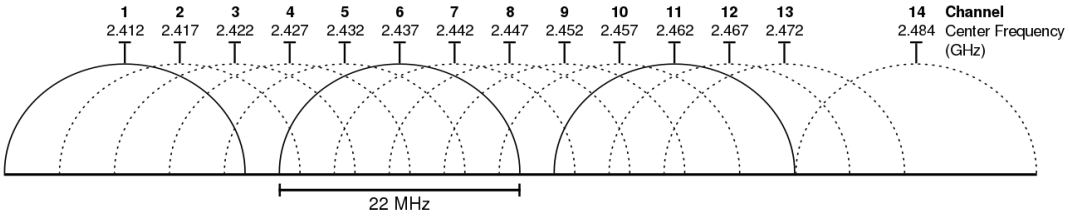

Figure 1.1: Illustration of the overlapping channels in the 2.4GHz range[2]

Motivation for minimizing interference has been highlighted in a paper by Wang et al. [3] by representing a value called signal-to-interference-and-noise ratio (SINR). The SINR value is expressed as:

$$SINR = \frac{IntendedSignal}{BackgroundNoise + InterferenceSignal}$$

Background Noise is said to usually be constant representing other sources which may not come from IEEE 802.11 devices. Intended Signal refers to the signal the receiver senses from the station. Lastly, Interference Signal refers to the superposition of signals received from other stations on the same channel. If the Intended Signal stays fixed (typical of residential situations), the Interference Signal is the only value which may change with channel assignment. Therefore, minimizing the Interference Signal value for one AP will result in better throughput for its users.

Problem formulation

WiFi technology is accessible at low costs with interoperability between devices thanks to the IEEE 802.11 standards. In densely populated areas such as urban residential neighbourhoods and public ”hotspots” such as cafes, the number of WLANs operating in the 2.4GHz spectrum

is high. Moreover, most of these networks can be considered chaotic networks. Akella et al. [4] define a chaotic network deployment as *unplanned* and *unmanaged*; meaning that most deplyoments tend to use a default channel. This causes congestion on the same channel, also called co-channel interference. There is also the possibility of adjacent channel interference i.e. channels interfering with nearby channels. In this paper, the focus is on reducing co-channel interference for better throughput in a chaotic environment for an Extended Service Set (ESS) of 3 Access Points (APs). The solution to the problem proposed makes use of blockchain smart contracts to coordinate the cluster of APs in the ESS. With the chaotic environment in mind, the aim is for an automated system that may suggest different AP channel configurations over time.

1.2 Report Structure

The report is structured in the following way:

- **Chapter 2** gives background information on the topics covered in the paper.
- **Chapter 3** gives an overview of the project setup.
- **Chapter 4** gives in-depth information on the implementation and setup of the project.
- **Chapter 5** discusses the results and evaluation of the project.
- **Chapter 6** concludes the usefulness of the project and discusses future work which can be done.
- **Chapter 7** discusses the legal, social, ethical and professional issues taken into account during the project.

Chapter 2

Background & Literature Review

2.1 IEEE 802.11 Overview

IEEE 802.11 is a member of the IEEE 802 family which is a series of specifications for LAN technologies [5]. Networks in the scope of IEEE 802 are part of the Data Link Layer and Physical Layer of the OSI model.

2.1.1 Nomenaclature and Design

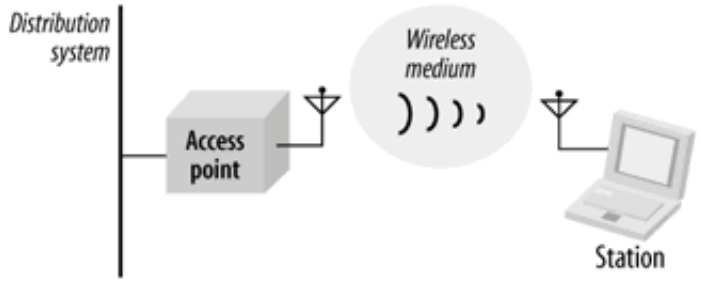

Figure 2.1: Components of the 802.11 LAN [5]

Components are:

- **Stations (or clients)**: computing devices with wireless network interfaces. [5]
- **Access Points (APs)**: devices that perform the wireless-to-wired bridging function among others. [5]
- **Wireless Medium**: Radio Frequency (RF) is the standard medium.

- **Distribution System**: the logical component of 802.11, used to forward frames to their destination. [5]

2.1.2 Types of Networks

The building block of an 802.11 network is the basic service set (BSS) which is a group of stations that communicate with each other. The communication takes place in an area called the basic service area. There are two main types of BSSs: *infrastructure* and *independent*. [5]

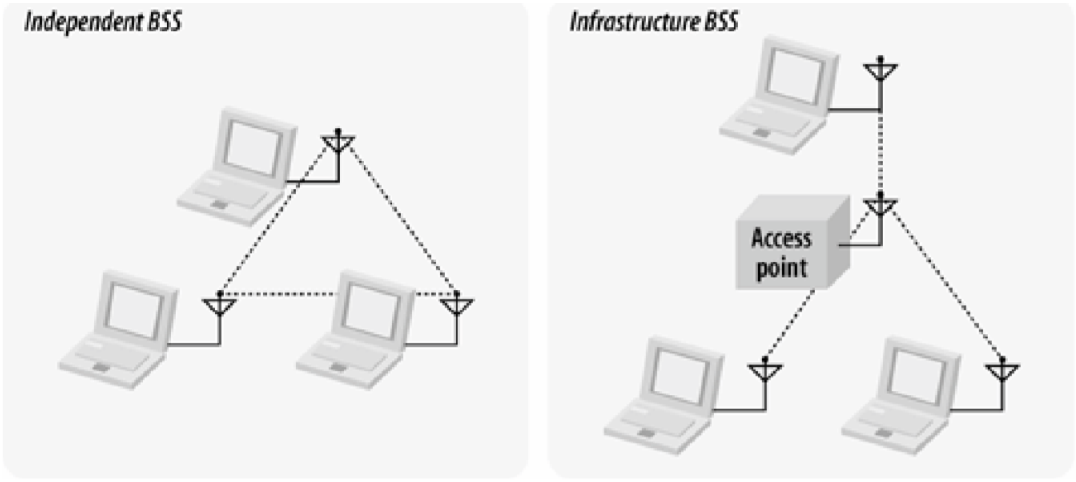

Figure 2.2: Infrastructure and Independent BSSs [5]

Independent BSSs on the left in Figure 2.2 are also called ad-hoc networks. In such a network, stations communicate with each other directly and must be within communication range. They are most commonly used for a short period of time in order to share data between stations. [5]

Infrastructure BSSs (right in Figure 2.2) are differentiated by their usage of an access point. In this type of BSS, communication between two stations is done via the AP, thus requiring two hops in the process. With all communications relayed through an access point, the basic service area is defined as the area in which transmissions from the AP can be received. Stations must associate with the AP to obtain network services. The association process is initiated by the station. The AP decides whether to grant access or not. Stations may be associated with only one access point at a time. [5]

Extended Service Area

An extended service area is created by linking multiple BSSs into a Extended Service Set (ESS) with a backbone network. ESS are used to cover a larger area than it is possible with only

one BSS such as offices, universities etc. All the APs in the ESS are given the same *service set identifier* (SSID) serving as a network "name". [5]

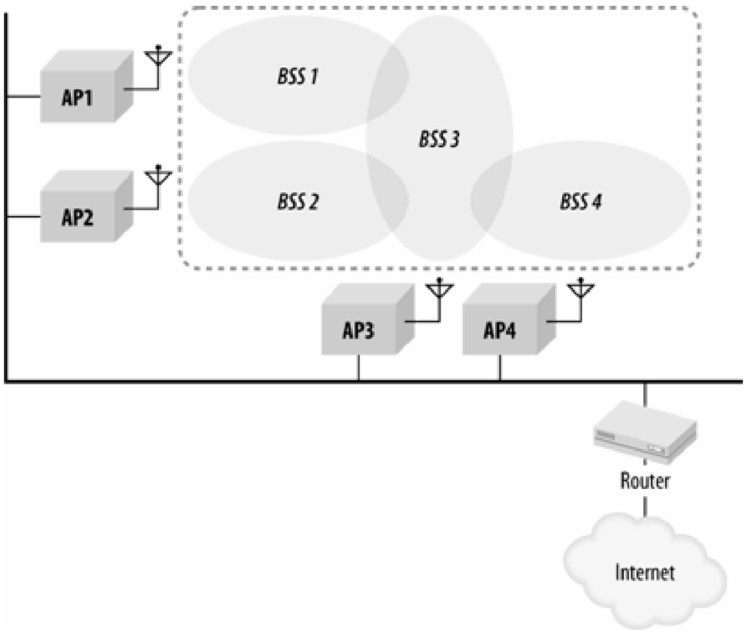

Figure 2.3: Extended Service Set [5]

2.1.3 802.11 Standards

There are several 802.11 standards for WLAN technology in the 802.11 family. A common characteristic of the different standards is the usage of CSMA/CA as their medium access protocol. The "Wifi" standards in popular use are summarized in Table 2.1. [6, pg.560]

The first standard has been created in 1997 and was simply called 802.11. The maximum bandwidth supported was 2 Mbps, making it unusable in for today's needs. The standards presented in Table 2.1 are in order of appearance with the last two being released in 2013 and 2015 respectively. [6, pg.560]

Standard	Frequency Range	Data Rate
802.11b	2.4 GHz	up to 11 Mbps
802.11a	5 GHz	up to 54 Mbps
802.11g	2.4 GHz	up to 54 Mbps
802.11n	2.4 Ghz and 5 GHz	up to 450 Mbps
802.11ac	5 Ghz	up to 1300 Mbps

Table 2.1: Summary of IEEE 802.11 standards [6]

2.1.4 802.11 MAC Protocol: CSMA/CA

CSMA/CA which stands for Carrier Sense Multiple Access with Collision Avoidance is a random medium access protocol for WiFi devices [6]. The protocol works by each station sensing the channel before transmitting and refraining from transmissions if the channel is busy. In collision avoidance, when a station begins to transmit a frame, it is transmitted in its entirety.

2.2 Dynamic Channel Assignment Algorithms

Dynamic Channel Assignment is a process of managing RF channel assignments for a group of APs. A DCA algorithm implements the assignment process. The algorithm can be of two types: *centralized* and *decentralized*.

In the case of centralized DCA algorithms, a device must act as controller. The controller, receives scanning information from the APs and decides on channel asignments that best suit the whole system.

Decentralized (or Distributed) DCA algorithms do not take into account what other APs in the same group (managed by the same domain) choose and take an individual choice.

The following subsections present proposed DCA algorithms from past papers. In the scope of the project, a centralized algorithm is needed in order to make use of smart contracts as a feature in blockchains.

Least Congested Channel Search (LCCS)

This is one of the most basic channel assignment algorithms. Being a distributed algorithm, it is meant to be run on each AP separately. The algorithm works by the AP running a scan of the channels and then selecting the least congested one i.e the channel with the least traffic between APs and clients [7]. The main issue of the algorithm is the Hidden Terminal Problem. The problem can be visualised in Figure 2.4. There are 2 APs: A and B, each having one client X and Y respectively. Clients X and Y interfere with each other (running on the same channel), but A and B can not discover each other. Therefore, the channel selection of the two APs causes interference for their clients. [7]

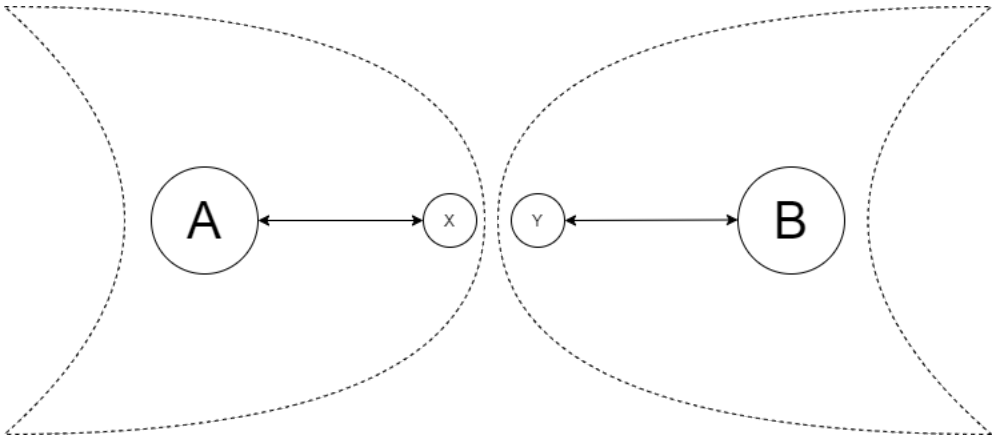

Figure 2.4: Illustration of the Hidden Terminal Problem

MAXchop algorithm

The MAXchop algorithm proposed by Mishra et al. [7] looks to improve fairness in chaotic deployments. Fairness is calculated at the AP level and not at client level. Perfectly fair performance is defined as the aggregated throughput of AP1's clients being equal to the aggregated throughput of AP2's clients where AP1 and AP2 are in proximity of each other. The proposed solution to the problem involves a process called channel hopping. Channel hopping involves APs spending a fixed amount of time (a slot) on a channel and then switch to a different channel as given by a predetermined hopping sequence. All clients of the AP switch channels together accordingly. There is said to be a minimal overhead in the switching process compared to gains. The result is the long-term throughput of an AP becoming an average of the throughputs achieved during individual channel assignments used by the network as a whole, thus achieving fairness. The algorithm utilizes information from neighboring APs to compute a good hopping sequence. The approach towards collecting information is client-driven with the AP requesting its clients to perform scans.

SCIFI's channel allocation algorithm

The following algorithm has been found ideal for the project's implementation. The DCA channel algorithm proposed by Balbi et al. [8] aligns really well with the capabilities of a blockchain's smart contract feature.

The SCIFI (Intelligent Control System for Wireless Networks) features a controller that orders each controlled AP to execute tasks such as performating spectral scanning and collecting statistics about associated clients. Based on the collected data, the controller executes the

allocation algorithm.

The channel allocation algorithm is modeled by an ”interference graph” where nodes represent APs and if there is interference between 2 nodes, they are connected by unidirectional nodes. Therefore, the problem becomes a classic graph coloring problem where colors are represented by channels. The color choice is made based on the quality of interfering signal received from other APs; the interference from APs not operating on channels 1,6 or 11 is assigned to the closest channel. As a tiebraker, the number of clients associated with the each controlled AP is used. The goal of the algorithm is to find a set of channels for the controlled APs such that the interference generated by neighbouring APs. The choice of channel is restricted to 1, 6 and 11.

The algorithm can be summarized as follows:

1. The controller collects data from the APs.

2. Based on the data collected, the algorithm determines the interfering neighbours of each controlled AP (including other controlled APs).

3. The interference graph is constructed: vertices representing APs and weighted edges representing the interference value; in the graph, APs that are not controlled are already colored.

4. An initial list containing the discolored nodes i.e. controlled APs is created.

5. The list is then ordered by priority in channel selection. Priority is given to the node with the highest saturation degree which is the number of different colors used by neighboring vertices. (the value being between 0 and 3). If there is a tie, the one with more associated clients will have priority. Upon another tie, priority is given to the one with the a lower IP.

6. After determining which vertex to color, its ”color” must be defined. A list containing all interference edges of the selected AP is created. The list is then iterated through to determine if there are any free channels (channels with no operating APs). A list of the unoccupied channels is also initialised. If there are unnocupied channels, first channel in the list is selected. If there are no unnocupied channels, the algorithm looks for the channel with the lowest interference sum.

7. After coloring, the node is removed from the list of discolored nodes, so that remaining nodes consider its interference.

8. The process is repeated until all nodes are colored.

2.3 Blockchain Overview

A Blockchain is a peer-to-peer, distributed ledger that is cryptographically-secure, append-only, immutable (extremely hard to change), and updateable only via consensus or agreement among peers [9]. Networks in the scope of Blockchain are built on top of the Application Layer of the OSI model.

- **P2P**: there is no central controller in the network, all nodes talk to each other directly.[9]
- **Distributed Ledger**: a ledger is spread among all the peers in the network; each peer holds a copy of the complete ledger. [9]
- **Cryptographically-secure**: cryptography is used to provide security against tampering and misuse. [9]
- **Append-only**: data can only be added to the blockchain in *time-ordered sequential order*; this property implies that once data is added to the blockchain, it is almost impossible to change the data and can be considered practically immutable. Nonetheless, it can be changed in rare scenarios wherein collusion against the blockchain network succeeds in gaining more than 51 percent of the power. [9]
- **Updateable via consensus**: In this scenario, no central authority is in control of updating the ledger. Instead, any update made to the blockchain is validated against strict criteria defined by the blockchain protocol and added to the blockchain only after a consensus has been reached among all participating peers/nodes on the network. To achieve consensus, there are various consensus facilitation algorithms which ensure that all parties are in agreement about the final state of the data on the blockchain network and resolutely agree upon it to be true. [9]

2.3.1 Blockchain as a Data Structure

Blockchain as a data structure consists of blocks chained together. A block encapsulates a set of transactions in the form of a Merkle Root hash. Each block has its own hash value which is often determined by multiple variables such as the Merkle Root hash. The chaining is done by including the hash of the previous node as block information; the first block, called the genesis

block, has no previous hash. Other important information a block includes is the timestamp which is the time of creation.

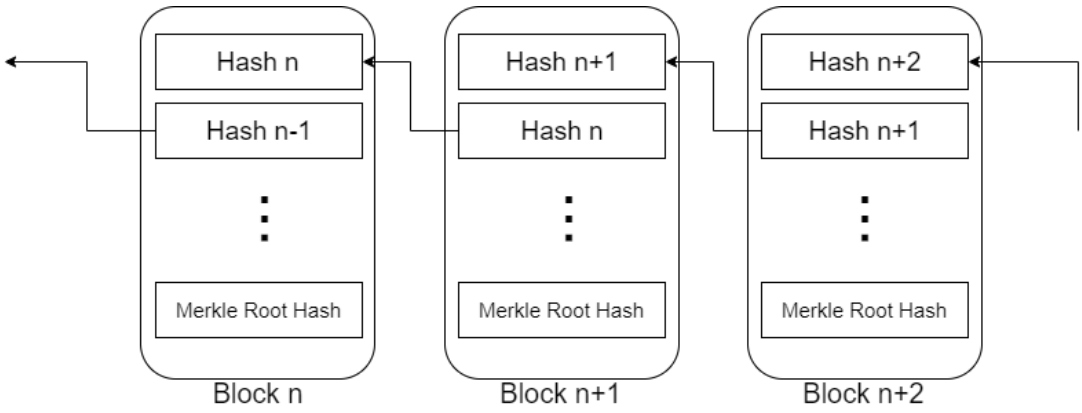

Figure 2.5: Basic structure of a blockchain

2.3.2 Consensus Protocols

There are multiple categories of consensus protocols, each designed with a different scenario in mind:

- *Proof-of-work algorithm*

  Full nodes act as miners competing with each other to solve a cryptographic puzzle, verifying all the transactions in the process. The solution is considered to be difficult to achieve, but easy to be verified by the network. The proof is that enough computational resources have been spent to build a valid block. Nodes are selected randomly in proportion to their computing capacity. [9]

- *Practical Byzantine Fault Tolerance algorithm*

  In distributed systems such as blockchains, Byzantine Fault tolerance is the ability of a network to function correctly to reach consensus in spite of malicious peers propagating incorrect information. The **pBFT** protocol came as an improvement upon the original Byzantine Fault Tolerance algorithm in a paper in 1999 by Castro and Liskov [10]. The consensus works by ordering all of the nodes in a sequence with one node being the leader and the rest backup nodes. All of the nodes communicate such that the honest nodes come to an agreement of the state through majority. This model works on the assumption that the number of malicious nodes can not equal or exceed a $\frac{1}{3}$ of the nodes in the network.

- *Proof-of-stake algorithm*

  Proof-of-Stake is a category of consensus algorithms for public blockchains that depend on a validator's stake in the network. In PoS, a set of validators take turns proposing and voting on the next block, and the weigh of each validator's vote depends on the size of its deposit (stake). The advantages of this protocol include security, reduced risk of centralization and energy efficiency. [11]

- *Proof-of-authority algorithm*

  PoA algorithms are reputation-based consensus algorithms that introduce a practical and efficient solution for private networks. The PoA model relies on a limited number of block validators which makes it a highly scalable system. Blocks and transactions are verified by pre-approved participants called sealers, who act as moderators of the system.

A consensus protocol has **three** key properties based upon which its applicability and efficacy can be determined: [12]

1. **Safety** - A protocol is considered safe if all nodes produce the same output and the outputs are valid according to the rules of the protocol
2. **Liveness** - A protocol guarantees liveness if all non-faulty nodes participating in consensus eventually produce a value
3. **Fault Tolerance** - A protocol provides fault tolerance if it can recover from failure of a node participating in consensus

2.3.3 Usage of Cryptography

Cryptography in Blockchain is used in the form of hash functions. A hash function takes an input of an arbitrary size and maps it to an output of a fixed size. The most common hash function used in the world of blockchains is SHA-256 which produces an output of 256 bits. Raikwar et al.[13] highlight the following general use cases of cryptography:

- Solving crpytographic puzzles (in PoW)
- Address generation (private and public keys)

- Shortening the size of public addreses.
- Message digests in signatures.

2.3.4 Types of nodes in the peer-to-peer network

There are 2 main types of nodes in the network: **full** nodes and **lightweight** nodes. [14]

- A full node acts as a server in a decentralized network. They are tasked with maintaining the consensus between nodes and the verification of transactions. They store a full copy of the ledger. Full nodes make decisions for the future of the network.
- A lightweight node depends on the full node to provide them with the necessary information regarding the blockchain. They don't store a full copy of the blockchain, so they can only query the current status of the blockchain and broadcast transactions for processing.

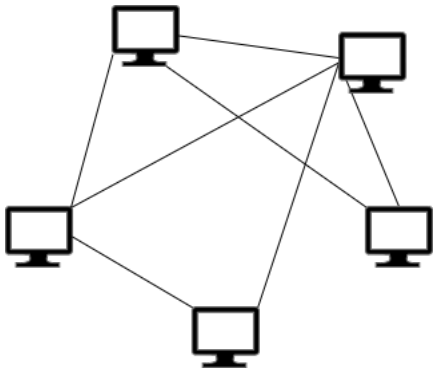

Figure 2.6: Example of peer-to-peer network

2.3.5 Types of Blockchain Networks

Private Blockchains

Such a blockchain is said to be "permissioned", meaning that not anyone can access it, as well as the identities of its users being known; as a result, it is regarded as less decentralised than the public networks. It is mainly used by enterprises to protect their data. In order to join a private network, the user needs an invitation. Each participant can have a different level of permissions. Most private blockchains employ a **PoA** consensus algorithm. Private blockchains are faster, more cost-effective and more efficient as well as being lighter. Private blockchains

are highly configurable; parameters such as the time passing between two consecutive blocks or the possible size of a block can be altered.

Public Blockchains

A public blockchain is said to be "open"; anyone can read, write or participate in the network, while also maintaining anonimity. Most public blockchains employ a **PoW** consensus or a more energy efficient **PoS** algorithm. The transaction speed is lower, due to the need for all of the full nodes in the network to verify the block.

2.3.6 The Smart Contract concept & DApps

Smart contract is a term first coined by Nick Szabo who defined it as computerized transaction protocols that execute terms of a contract. In practice, smart contracts lay down the rules for transactions in a software manner so that a credible transaction can be done without the need of a middle man to verify it (sometimes reffered to as a trustless escrow).

The concept has been used in blockchains and other DLTs as the base for application creation. Blockchain-based applications came to be reffered to as DApps which stands for Decentralized Applications. The difference between a DApp and other standard applications is its architecture.

A DApp uses the blockchain data structure as a database which also stores smart contracts which act as programs with certain rules. Interactions with the smart contract must be made via a node and an account as the identity to send from.

2.4 State of the Art Blockchain Platforms

This section gives an overview of the current smart contract blockchain platform options. The smart contract capabilities must be extensive enough for DApp implementation. These platforms allow for private deployment.

2.4.1 Ethereum

Ethereum is the first blockchain platform to have implemented smart contracts and is considered to be the most popular at the moment. Its base idea is to provide a blockchain with a built-in Turing-complete programming language that can create **smart contracts** to encode

arbitrary state functions and allow users to create applications.[15] The platform uses Ether as its cryptocurrency.

The public Ethereum network uses a **mining PoW** consensus mechanism [16, p.11]. There is also the possibility of starting your own private blockchain network using a **PoA** called Clique using a client implementation that supports it such as Geth or Parity. An Ethereum client refers to any node able to parse and verify the blockchain. Due to the energy cost-efficiency problems of the **PoW**, the Ethereum network is set to move to a **PoS** protocol called Casper in the near future.

Ethereum works by using a distributed virtual machine called the **Ethereum Virtual Machine** (EVM). The EVM allows for code to be verified and executed on the blockchain, guaranteeing the same output on everyone's machine. Unlike Bitcoin, which uses unspent transactions to keep tract of the balance of a user, Ethereum uses accounts. Thus, Ethereum stores the state of the EVM and the balance of all the accounts.

In Ethereum, a smart contract is code run on the EVM that can be written in a small set of languages although the language **Solidity** established itself as the standard.The smart contract is stored on the blockchain Applications that use smart contracts for processing are called decentralised applications or DApps. In this sense, a DApp uses the smart contract as an interface to interact with the blockchain.

The Ethereum platform is an open-source project that is being continuously worked on which allows anybody to start their own private blockchain by using one of its many client implementations.

2.4.2 EOS.IO

EOS.IO is another open source blockchain platform developed by block.one which features smart contracts. Its consensus algorithm is a Delegated Proof of Stake algorithm (DPoS) which works similarly to a PoS algorithm, but it involves a selection system for block producers. EOS.IO boasts block creation times of 0.5 second and only one producer may be chosen to create one block. Blocks are created in round of 126 with 21 unique producers (chosen) create 6 blocks each.Transactions are confirmed 99.99% of the time after an average of 0.25s. [17].

Accounts can be referenced by a readable name of up to 12 characters. Each account can send structured Actions to other accounts and may define scripts to handle Actions when they are received. The EOS.IO software gives each account its own private database which can only be accessed by its own action handlers. Action handling scripts can also send Actions to other

accounts. The combination of Actions and automated action handlers is how EOS.IO defines smart contracts. [17]

EOS.IO features a Role based Permission management system that involves determining whether or not an Action is properly authorized. The simplest form of permission management is checking that a transaction has the required signatures, but this implies that required signatures are already known. Generally, authority is bound to individuals or groups of individuals and is often compartmentalized. The EOS.IO software provides a declarative permission management system that gives accounts fine grained and high-level control over who can do what and when. [17]

Chapter 3

Design & Specification

The current project implements a dynamic channel allocation algorithm inside of an Ethereum smart contract. The choice of using the Ethereum platform stems from the its popularity (there is an abundance of tools and frameworks that help development) and familiarity with the ecosystem. Ethereum fulfils the need for a private deployment with smart contracts as feature. The whole system to be implemented can be considered a back-end Ethereum DApp as there is not front-end interface with which you can interact; the goal is for automation. The smart contract will have two types of accounts that can interact with it: controlled APs accounts and the sealer account which is considered to be the contract's owner. The network uses the Clique PoA algorithm for consensus, thus ether has no value (there is no reward for creating blocks). The nodes in the blockchain network are Raspberry PIs which are set up as APs in an ESS (under the same ESSID) and a computer that would act as the sealer. Nodes are running the Geth client.

There is no other option in the Clique PoA algorithm than using full nodes. All nodes are in a LAN behind a switch/router with static IPs assigned to them to facilitate communication and confer security as can be seen in Figure 3.1.

The Raspberry PI models used for the project are RPI3 and RPI4 (Table 3.1 for capabilities) The operating system used for the Raspberry PIs is Raspbian Buster. The computer which acts as the sealer can run any operating system; in the case of the project, a Windows 10 (64 bit) machine has been used. In order to not repeat configurations, suppose the sealer is a Linux-based machine as well. Each AP runs its own DHCP server through *dnsmasq*, while *hostapd* takes care of the AP configuration.

Model Number	Wireless Capabilities
Raspberry Pi 3 Model B	802.11n
Raspberry Pi 3 Model A+	802.11ac/n
Raspberry Pi 3 Model B+	802.11ac/n
Raspberry Pi 4 Model B	802.11ac/n

Table 3.1: Raspberry PI Models WiFi Capabilities

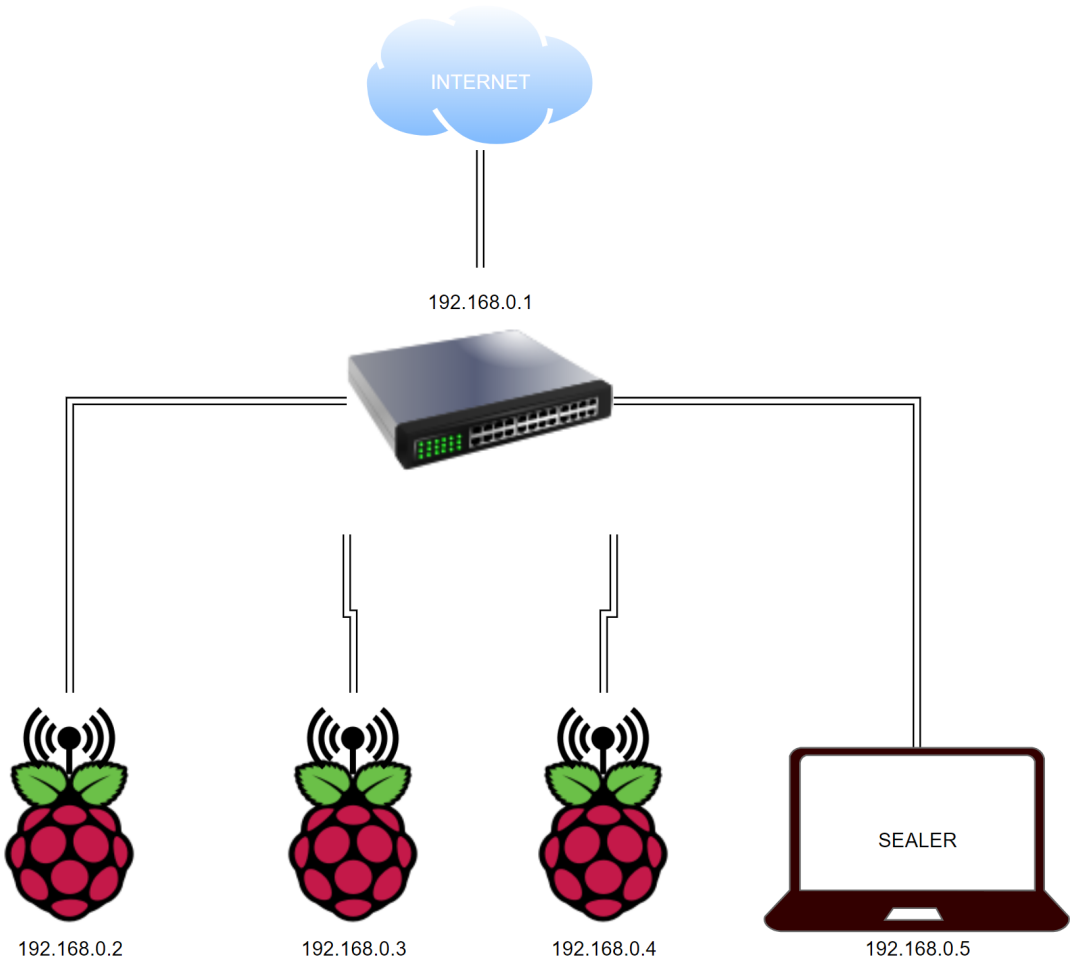

Figure 3.1: Overview of the network from the LAN perspective

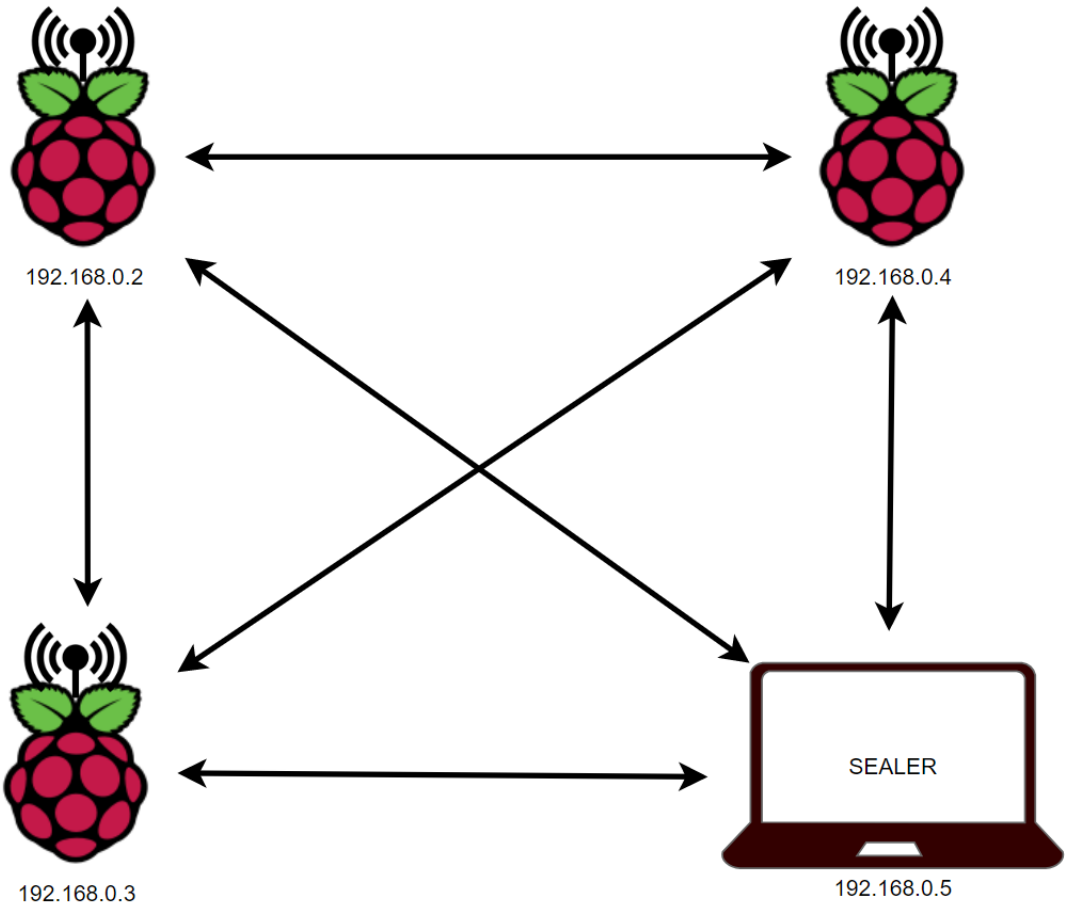

Figure 3.2: Overview of the network from the Blockchain Peer-to-Peer perspective

Chapter 4

Implementation

4.1 Creating the wired LAN

The devices in the network need to be given static IP address for later P2P configuration. In this scope, the nodes are set up in a wired sub-LAN network by ethernet cables for better security and easier setup. A router model Tenda Wireless N301 has been used purely as a switch. The ethernet network interface for each device has to be configured manually. The default setting is to request an IP address from the router's DHCP server. This must be changed to a static IP within the range of the router. For example, if the router gives out address in the 192.168.0.2-192.168.0.255 range, IPs can be set to the one illustrated in Figure 3.1. After each device is set up, ethernet cable connection is done.

On the Raspberry PIs, run the following command in the terminal:

```
$ sudo nano /etc/dhcpcd.conf
```

Configure the eth0 interface where XX is to be replaced with the actual number assigned by the router and YY with a chosen number:

```
interface eth0
static ip_address=192.168.XX.YY/24
static routers=192.168.XX.1
static domain_name_servers=192.168.XX.1 8.8.8.8
```

In the case of my project, the XX is equal to 0 and YY numbers between 2 and 5 with the sealer being assigned 5 as in Fig.3.1. Save the modifications and reboot.

4.2 Setting up the Raspberry PIs as Access Points

With the following setup, the Raspberry PIs can not use their wifi interface (*wlan0*) to become a client. In order for a Raspberry PI to become an AP, the services hostapd and dnsmasq are required.

Hostapd is a user space daemon for access point and authentication servers. It implements IEEE 802.11 access point management, IEEE 802.1X/WPA/WPA2/EAP Authenticators, RADIUS client, EAP server, and RADIUS authentication server [18]. The project uses the service for setting up the AP and managing authentication.

Dnsmasq provides network infrastructure for small networks: DNS, DHCP, router advertisement and network boot [19]. The main use of the service consists in the DHCP subsystem which will assign IP addresses to the clients of the access point.

The 2 services can be installed on the Raspberry PIs with the following command in the terminal:

```
sudo apt-get install hostapd dnsmasq
```

Stop the two services before configuring:

```
sudo systemctl stop hostapd
sudo systemctl stop dnsmasq
```

The setup walk-through is based on the guide by Raspberry Connect[20] which has been tested positively.

Hostapd configuration

Run:

```
sudo nano /etc/hostapd/hostapd.conf
```

Use the hostapd configuration in Appendix A.1.1. with the same SSID and password. Another important setting from the configuration file is the IEEE protocol in uses which is IEEE802.11g Save the file, then update the path to the config file in:

```
sudo nano /etc/default/hostapd
```

```
From: #DAEMON_CONF=""
```

```
To: DAEMON_CONF="/etc/hostapd/hostapd.conf"
```

Save the file and run the following commands:

```
$ sudo systemctl unmask hostapd
$ sudo systemctl enable hostapd
```

DNSmasq configuration

Run the command:

```
$ sudo nano /etc/dnsmasq.conf
```

And add the following lines:

```
interface=wlan0
server=8.8.8.8
domain-needed
bogus-priv
dhcp-range=192.168.50.{from},192.168.50.{to},30m
```

Value of *to* and *from* can be anywhere between 3 and 255, where *from* must be strictly smaller than *to*. This represents the range of ip addresses which the AP will give out to clients. The lease time is 30 minutes.
Run:

```
$ sudo nano /etc/dnsmasq.conf
```

And add the following to the bottom of the file:

```
nohook wpa_supplicant
interface wlan0
static ip_address=192.168.50.2/24
static routers=192.168.50.1
static domain_name_servers=8.8.8.8
```

4.3 Blockchain deployment

Geth & Tools

Go Ethereum, shortened as geth, is an Ethereum Client implementation written in the Go programming language. Beside its capability of running a full node, Geth can also be used as

a wallet, an interactive JavaScript environment to interact with the blockchain, and initialise a genesis block. The binaries of Geth & other tools can be downloaded by running the following command on Linux-based systems:

```
$ wget https://gethstore.blob.core.windows.net/builds/
geth-alltools-linux-arm7-1.9.12-b6f1c8dc.tar.gz
```

Among these binaries, there are 3 which are used in the project: *geth*, *puppeth* and *bootnode*. The path to these can be added to enviroment for easier usage.

Puppeth is used to generate the genesis configuration file. The genesis configuration file is important as it lays the parameters of the exact private deployment and it is necessary to initialise the genesis block for each node. The file is formatted as a JSON file and contains information about:

- Chain ID: a number; choosing a large number is indicated as a small numbers might be in use by public networks
- Difficulty: determines the time necessary to create a block; the smaller the number, the faster the blocks get created.
- Consensus Protocol: choice between Ethash (PoW) and Clique (PoA); in the case of Clique, the sealer accounts are set inside the field ”extraData”
- Initial allocation: amount of ether for certain accounts; when using Clique, transactions don’t cost gas (ether), but an account must have ether to be able to send transactions.

Before creating the genesis file with puppeth, an account should be created on each device. A folder is needed to act as the data directory for all things related to the geth node. This folder is to be used in the following geth commands. Geth also has an in-built wallet to manage its local accounts. An account can be created on a geth node by running the following command: [21]

```
$ geth --datadir "path/to/folder" account new
```

An account per geth node is needed, meaning 4, one for the sealer, and three for the APs. Accounts are stored inside of a folder named **keystore** within the folder dedicated to the respective node. For simplicity of testing, (not indicated in a real world scenario) the password ”**test**” has been chosen for all the accounts created.

With the accounts created, the genesis file can be created by Puppeth with the sealer account being the account created on the Windows Machine and all accounts (out of the ones created) allocated initial ether.

Creating the genesis file with Puppeth

Execute the puppeth binary.

```
$ puppeth
```

When prompted to specifiy the network name, input a name of your choosing; in the scope of the project, the name **wifiblockchain** has been chosen.

Choose option *2. Configure new genesis* then *1. Create new genesis from scratch*
Upon being prompted to choose the consensus engine, choose *2. Clique - proof-of-authority*
For fast transaction processing choose for blocks to be created in **1** second.
Next, choose which accounts are allowed to seal; i.e. the account created on the sealer node. Next up, input the addresses of all the accounts created on all nodes for initial ether funding. Upon prompted whether to pre-fund the addresses 0x1..0xff with 1 wei, choose **no**. On the next step, do not input any value i.e let it assign a random id. The configuration file has been created with the parameters chosen. To export the actual JSON file needed for initialisation, choose *2. Manage existing genesis*, then *2. Export genesis configurations* and leave the default value for the next step. In the folder where the binary has been executed, a file called wifiblockchain.json has been created. This is the file which is going to be used to initialize the nodes.

Setting up the nodes

With the genesis JSON file created, the genesis block can be initialised with the command: [21]

```
$ geth --datadir "path/to/folder" init wifiblockchain.json
```

The initialisation of the genesis block is necessary on each node with the exact same genesis file.

Setting up the bootnode

A bootnode is a light node used for easier discovery between the nodes by creating a central point through which they can find peers, instead of connecting each peer with each peer. Its

only functionality as a node is in peer discovery. Another way of connecting the nodes is by specifying all the peers from the beginning. The bootnode approach has been chosen as favorable in the project with the sealer machine hosting the bootnode.

The bootnode binary needs to be run to generate what is called a bootnode key and run the node:

```
$ bootnode -genkey bootnode.key
```

In the terminal, an enode identifier is printed; this is used to connect to the bootnode from each node. An enode is of the form:

```
enode://[Hash derived from the bootnode key]@[IP address of the machine]:[Listening port]
```

Whenever the bootnode needs to be brought up again, the command

```
$ bootnode -nodekey bootnode.key
```

can be run with the already generated bootnode.key file.

Running the nodes

Settings for a geth node may become verbose, so generating a configuration file is preferable in the long run. The file can be generated by running the following command: [21]

```
$ geth --datadir "."
       --syncmode "[full for sealer, fast for the rest]"
       --networkid "[chainIdFromGenesisFile]"
       --bootnodes "[enode of bootnode from previous section]"
  dumpconfig > configFile
```

For the sealer node, preloading a script is necessary to make sure the node seals blocks only when there are transactions available that are not yet sealed into blocks. Since geth offers an interactive JavaScript environment, JavaScript programs may be preloaded into the console. The script *mineWhenTransactions.js* (Appendix B.1.1) makes sure that the sealer does the sealing process only when needed in order to avoid creating unnecessary blocks. Moreover, unlocking the sealer account is necessary for sealing blocks successfully. Create a folder **scripts** inside the node folder and place the script. The *nat* setting needs to be set to none so that the node can only be addressed by its static IP in the LAN.

Run the sealer with the command: [21]

```
$ geth --preload "scripts/mineWhenTransactions.js"
       --config "configFile"
       --unlock "{Ethereum account address of the sealer}"
       --nat "none"
  console
```

The password of the account to unlock is going to be prompted in the console. After starting the node, the sealer will create blocks automatically. The account needs to stay unlocked.

And the other nodes with:

```
$ geth --config configFile
       --nat "none"
  console
```

The command console launches a JavaScript interactive console with which the blockchain ca be interacted. The following is a list of variables/functions which are useful:

- admin.nodeInfo : prints out information about the node
- admin.peers : prints out the peers discovered by the node
- net.peerCount : prints out the number of peers discovered
- eth.accounts : prints the accounts managed by the node

4.4 The Smart contract & Interactions

4.4.1 Smart contract implementation

The smart contract implements the SCIFI DCA algorithm proposed by Balbi et al. [8] (explained in further detail in Subsection 2.2) inside of an *allocate channels* function of a smart contract named *Controller*. This section explains the implementation of the smart contract used in the channel allocation DApp. The main challenge in designing the contract has been the relative minimalism of the Ethereum Virtual Machine instruction set available through the Solidity language. Smart contract functions may only receive arguments of fixed size such as fixed-size array or integers. Moreover, smart contracts are meant to contain only the most essential information possible presented in a simple way. Thus, the part of processing the scan

output falls down to the APs themselves. There is no ”interference graph” in the implementation of the smart contract as information is condensed as much as possible by using structs and mappings.

The struct **AP** is used to represent a controlled AP containing information such as the Ethereum address (account) associated with the AP, the channel it's currently set to operate on, the number of clients associated to the AP upon the latest scan and the AP's MAC address. The MAC address is necessary to differentiate between APs when the scanning output contains multiple APs of the same ESSID so that interference can properly be assigned to the correct AP. The three AP objects are stored in the state of the smart contract as a dynamic array of type AP.

The struct **scanInfo** represents scanning information about a certain channel (from the perspective of an AP) and its aggregated interference value (sum of all interference edges for an AP for a particular color/channel). The struct is used in a mapping named scan of type Ethereum address to fixed size (size 3) array of type scanInfo; the mapping represents the scanning information regarding the three non overlapping channels for the AP associated with the account address.

The struct **controlledAPscanInfo** represents strictly the interference another controlled AP generates from the perspective of one of the controlled APs. The current channel it is interfering on not important as all APs are discolored at the beginning of allocation. Thus it's value is not included in an APs **scan** mapping until after a channel is allocated to the AP. A separate mapping of type address to size-two array of type controlledAPscanInfo called *otherAPscanInfo* is kept for this information.

The two most important functions of the Controller contract are **submitScanInfo** (Algorithm 1) and **allocateChannels** (Algorithm 2).

input : A size 3 array of type int **channels**
A size 3 array of type int **totalInterferences**
Int value **noOfAssociatedClients**
A size 2 array of type string **MACaddreses**
A size 2 array of type int **controlledInterferences**
output: None; changes made to contract's state

mapping otherAPscanInfo ← $(MACaddresses, controlledInterferences)$

for $i \leftarrow 0$ **to** 3 **do**
  mapping scan[function caller][i] ← $scanInfo(channels[i], totalInterferences[i)$
  if *ListOfAPs[i].ethaddr == function caller* **then**
    ListOfAPs[i].noOfAssociatedClients = noOfAssociatedClients ;
  end
  mapping submitted[function caller] ← *true*
  if *allAPsSubmitted == true* **then**
    emit event AllSubmitted ;
  end
end

Algorithm 1: submitScanInfo function

input : None; contract state is read
output: None; changes made to contract's state

initialize AP[3] discoloredNodes ← $ListOfAPs[]$

sort discoloredNodes by **priority**: higherSaturationDegree, higherAssociatedClients, olderTimestamp;

for *i*← 0 **to** 3 **do**
  selectedAP ← $discoloredNodes[i]$
  if *checkIfFreeChannel(selectedAP) == true* **then**
    assignFreeChannel(selectedAP);
    addInterferenceToScanInfoForRestAPs(assignedChannel);
  end
  else
    assignChannelWithLeastInterference(selectedAP) ;
    addInterferenceToScanInfoForRestAPs(assignedChannel);
  end
end
for *j*← 0 **to** 3 **do**
  if *assignedChannel(discoloredNodes[j]) != currentChannel* **then**
    setChannel(assignedChannel(discloredNodes[j])) ;
  end
  set mapping submitted[discoloredNodes[j] ← $false$
end

Algorithm 2: allocateChannels function

4.4.2 DApp architecture & the allocation cycle

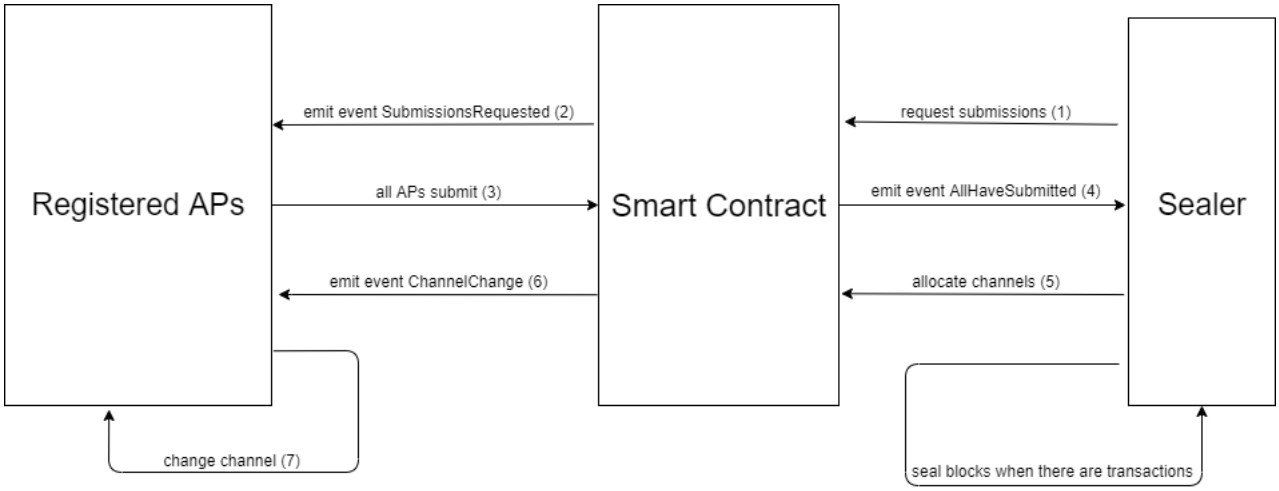

Figure 4.1: Interaction between nodes and smart contract

A walkthrough of an allocation cycle has the following steps as can be seen in Figure 4.1:

1. The sealer requests for the APs to submit their scan information.
2. The smart contract emits an event for which the APs listen.
3. The APs submit their scan information.
4. The smart contract emits an event letting the sealer know that all of the APs have submitted their information.
5. The sealer calls for the smart contract to allocate channels to the APs.
6. If the allocated channel is different from the current channel, emit an event for the AP to change their channel configuration.
7. The APs change their channel configuration accordingly.

The smart contract *Controller.sol* is written in the Solidity smart contract programming language with pragma version **0.6.4** (and any compatible versions)[22]. Moreover, an experimental pragma is in use in order to be able to compare strings. The name has been chosen due to the contract's functionality of changing an APs assigned channel. The contract is written so that the account that is used to deploy it becomes the owner. The owner is able to add the respective accounts managed by the nodes on the APs and call certain functions. The registered APs are the only ones able to submit scanning information to the contract. The full

source code of the smart contract can be found in Appendix B.2.1.

The implementation of the allocation algorithm works as follows:

1. Make a copy of the list of all the registered APs. (discoloredNodes)
2. Order the discoloredNodes list by number of APs scanned decreasingly.
3. Iterate through the ordered discoloredNodes list and check if the selected node has any channel that is completely free. If such a channel is found, appoint the AP to that channel; else find the channel with the least total interference value.
4. After all channels are allocated, change the state of the APs' channels and emit an event if there has been a change in channel.

4.5 Solc.js & Compiling smart contracts

Solc.js is a JavaScript library for the "compilation" of a Solidity smart contract. It can be downloaded through npm (Node Package Manager) which is part of **Node.js**.
Install solcjs globally by running the command:

```
$ npm install -g solcjs
```

The library can then be used as a terminal command to generate the **ABI** (Application Binary Interface) and compile a smart contract's code. The ABI is necessary for deploying a smart contract and sending transactions to it. The Application Binary Interface is stored as a JSON structure which describes a deployed contract and its functions acting as the contract's interface. It is therefore needed to call a smart contracts functions. The binary is a compact representation of the compiled smart contract code; it is used in the deployment of a smart contract and is stored in the blockchain after deployment.

The following commands can be used to generate the ABI and binary:

```
$ solcjs --bin {NameOfContract.sol}
$ solcjs --abi {NameOfContract.sol}
```

Two files will therefore be generated containing the needed outputs:

- `NameOfContract_sol_NameOfContract.abi`
- `NameOfContract_sol_NameOfContract.bin`

4.6 Web3.js & Interacting with the Blockchain

The Web3.js library is an important part in the automation of the project. Web3.js is a JavaScript library which allows programmatical interaction with the blockchain through a HTTP or IPC connection to a running node, local or remote. The version of web3.js used in the project is **1.2.6**. In all of the scripts, the connection to the node is done through IPC (Inter-Process Communications) interface which allows all of the management APIs, as all scripts are run on machines that have a local running node. The path to IPC needs to be set accordingly on each machine. On Linux based systems, this is the path to a file called **geth.ipc** which gets generated upon running the node inside of the node's data directory.
The library features mainly asynchronous functions which require JavaScript syntax involving Promises. The following syntax has been used consistently in the Node.js scripts developed:

```
asynchronousFunction(arguments).then(callbackFunction)
                                .catch(errorHandler)
```

Central to the web3.js library is the *eth* module. Out of its functionalities, the following submodules are used in the scope of the project:

- web-eth-personal: allows interaction with the node's accounts

  Functions used in source code:

  ```
  web3.eth.personal.getAccounts([callback])
  ```

  Returns the list of the nodes controlled by the node. The list returned might be empty if there are no accounts managed by the node. Callback function is optional.

  ```
  web3.eth.personal.unlockAccount(address, password, unlockDuration [,
      callback])
  ```

  Unlocks the account at the specified valid address with the given password (a string) for a given unlock duration. An account must be unlocked in order to send and sign transactions/data. The specified account has to be one managed by the node. If the unlock duration passed is 0, the account stays unlocked indefinitely. The unlock duration is in milliseconds. The callback function is optional.

  ```
  web3.eth.personal.lockAccount(address [, callback])
  ```

  The opposite of the unlock function. The specified address must be a valid account managed by the node. Attempting to lock an already locked account does not result into an

error.

- web3.eth.Contract : an object which facilitates interaction with Ethereum smart contracts.

  When creating the object, the JSON ABI (Application Binary Interface) of the respective smart contract is passed as an argument. Web3.js will then transform all calls to the contract object instance into low level ABI calls over RPC. A contract object can be created as follows:

  ```
  new web3.eth.Contract(jsonInterface[, address][, options])
  ```

  The jsonInterface is the parsed JSON ABI. The address is optional. If the smart contract is already deployed onto the blockchain,

  ```
  myContract.deploy(options)
  ```

  As its name suggests, the function deploys a smart contract to the blockchain. This function is to be called on an Contract object. The argument *options* is an object containing two fields, **data** which a string representing smart contract's compact binary (as 0x concatenated with the binary value) and **arguments** (optional) which is an array containing the arguments to be passed to the contract's constructor.

  ```
  myContract.methods.myMethod(arguments).send(options[, callback])
  ```

  Sends a transaction to a smart contract's function capable of altering the contract's state. The function *send* is called upon a contract's method (function). Arguments necessary to the method can be passed. The *options* argument is an object containing multiple fields out of which a field called **from** is necessary. This represents the account from which the transaction is to be sent. A callback function is optional.

  ```
  myContract.events.MyEvent([options][, callback])
  ```

  A function that subscribes to a contract's event. If the specified event is emitted (fired), a callback function can be called.

4.7 Developed Node.js scripts

The following scripts have been developed to automate the interaction between nodes and the smart contract as modeled in Figure 4.1. All scripts have been written with the possiblity of

Proof-of-Work in mind. For the current setup of Clique Proof-of-Authority, the estimation of gas for transactions is not useful. Also, all scripts use the IPC interface to connect to their local node. By default, all geth nodes have their IPC interface enabled, but the path convention is different between Windows machines and Linux/Mac OS machines.
Windows machines use named pipes for IPC. Therefore, the path is by default:

```
"\\\\.\\pipe\\geth.ipc"
```

Linux/Mac OS machines have their geth.ipc file inside of the node's data directory. Therefore the path is of the format:

```
"path/to/nodeDataDir/geth.ipc"
```

4.7.1 Script order execution

- Run *deployContract.js* on sealer machine (this part is run, in theory, just once as the same smart contract can then be used indefinitely as account addresses or MAC addresses are constant)
- Get smart contract address and update in *sealer.js* script and *AP.js* script for each AP node.
- Run *AP.js* scripts on each AP node.
- Run *sealer.js* on sealer node.

4.7.2 Sealer Scripts

Deploying the smart contract & registering the APs

The smart contract can be automated through a Node.js script by compileing (solcjs) and then deploying (web3js) it. The deployment part is separated from main sealer script as the contract address has to be known to the APs before they can react to smart contract events. In the project, the file **deployContract.js** (B.1.2) is to be run on the sealer node to deploy the contract to the blockchain, as the sealer account is also meant to be the smart contract's owner. In the script, the solcjs library is used to generate the **ABI** and **bytecode** necessary for the deployment from the Solidity file containing the smart contract code (Controller.sol).
Depending on the configuration, there are two values which need to be adjusted accordingly: the IPC path and the list of accounts to register.

The script deploys the contract, then registers a list of account addresses as APs to the smart contract. These are registered with a default channel value of 2 which doesn't represent the actual channel setting of the APs. This will be changed after the first allocation cycle. The address of the smart contract will be printed out; this is needed in further scripts to interact with the smart contract.

The execution of the script can be summarized as follows:

1. Execute the command **"solcjs –abi –bin Controller.sol"**
2. Read synchronously the two files generated (ABI and BIN) and store in variables.
3. Get the account then unlock it. (Just in case it is locked, the sealer account must stay open in order to create blocks)
4. Create the Contract object from the ABI variable.
5. Deploy the contract. At this step, the contract's address is printed in the console.
6. Register three account addresses and their respective MAC address as APs with an initial channel of 2.

Run the script with:

```
$ node deployContract.js
```

Main Sealer Script

The script *sealer.js* is designed so that it runs across a single allocation cycle. It is the one that kicks off the cycle by requesting the registered APs to the scannnig information to the smart contract. After the APs have all submitted their scan info for the channel allocation round, an event is automatically emitted by the smart contract, then the sealer reacts to the event by calling the allocation function of the smart contract.

The execution of the script can be summarized as follows:

1. Get the default account.
2. Create a Contract object with the appropriate contract address and ABI generated previously.
3. Call the request submissions function of the smart contract.
4. Listen for the *AllSubmitted* event. Upon receiving the event, call the allocation function.

```
$ node sealer.js
```

4.7.3 AP Script

The scanning script is used by the APs to listen to events such as SubmissionsRequested and ChangeChannel and to react accordingly. It is more complex than the rest of the scripts as its functionalisties include data collection and processing, but also file writing. This script needs to be running on all registered APs before the sealer may begin an allocation cycle.

The execution of the script can be summarized as follows:

1. Get the default account.

2. Create a Contract object with the appropriate contract address and ABI generated previously.

3. - Listen for the **SubmissionRequest** event and start the submission sequence upon triggering. The submission starts with scanning; it involves two parts: getting the number of clients currently associated to the AP and performing a spectral scanning and processing the output.

     The number of associated clients is directly queried from the hostapd service (using hostapd CLI).

     The processed output filters scanning information to include the 2.4GHz spectrum only. It then separates information on controlled APs from neighboring (uncontrolled) APs. The interference values are transformed to represent percentages out of a hundred and it represents the strength of the signal received by the AP from other APs.

     Information on controlled APs is about their MAC address and interference, while information on neighbouring APs is summarized as channel information i.e the number of the channel (1, 6 or 11) and the aggregated interference value of all the APs operating on the respective channel. This information is then submitted to the smart contract in multiple arrays.

   - Listen for the ChannelChange event. This event is triggered only upon a discrepancy between what the contract currently has assigned as the AP's channel and the allocation algorithm's output. The event logs two variables: the ethereum address of the AP's account and the channel number that needs to be changed. Based on this information, the channel number is then updated for the hostapd service.

The script needs to be run with sudo privileges to work properly:

```
$ sudo node AP.js
```

Chapter 5

Results/Evaluation

5.1 Testing Results & Known bugs

Testing with an Ethereum DApp can be quite complicated due to the need for a peer to peer network structure with running nodes. The testing done for the project has been done on the following levels.

5.1.1 The Blockchain network & the nodes

There might be a lot of different problems that arise when trying to get the network running.

Peering problems

- **Bootnode not being reachable because of firewall:** as the bootnode is being hosted on the sealer which is a Windows 10 machine the firewall is by default blocking connections from other machines in the LAN. The problem is solved through creating an inbound rule for connections from the IPs of the APs to bypass the firewall.
- **Nodes not finding each other due to slightly different node settings:** out of all the node settings, having the same ChainID set and genesis block initiated are crucial. To avoid this problem, the configuration file approach has been used.

Sealer node problems

- **Sealer not creating blocks due to locked account:** the account might be locked if the account password passed in incorrect. Another way the account could potentially be

locked is through one of the Node.js scripts. A way to solve using a wrong password is using a password file to read from;

AP node problems

- **Blockchain size becomes too big to store on RPI:** this is a problem specific to the PoA Clique implementation which does not allow other node settings beside full nodes.
- **One of the nodes crashes randomly:** there was no solution found for the problem, which occurs mainly when running the script.

5.1.2 The Solidity smart contract

The Solidity smart contract *Controller* was developed and tested/debugged on the online IDE Remix [23]. The IDE features a compiler, debugger and is able to deploy contracts and run transactions. It offers a simulated environment to test contracts in or even connect to a local node. The most debugging and testing was prioritised on the allocation function as it is the most computationally complex.
The testing on the allocation function has been done using dummy data; the problem with this approach is that there was no way found to simulate a WiFi environment with APs so that the input data is realistic in a real way.

The following scenario showcases the output of the allocation function.

Scenario

In this scenario, the 3 APs : AP1, AP2, AP3 all have the three non overlapping channels ocuppied.

AP	No. associated clients
AP1	0
AP2	2
AP3	3

Table 5.1: Associated Clients

- AP1:

	AP1
AP2	50
AP3	67

Table 5.2: Interference about other APs from AP1

Channel no.	Aggregated Interference
1	50
6	68
11	105

Table 5.3: Neighbour channel interference for AP1

- AP2:

	AP2
AP1	35
AP3	72

Table 5.4: Interference about other APs from AP2

Channel no.	Aggregated Interference
1	123
6	201
11	95

Table 5.5: Neighbour channel interference for AP2

- AP3:

	AP3
AP1	65
AP2	85

Table 5.6: Interference about other APs from AP3

Channel no.	Aggregated Interference
1	200
6	107
11	108

Table 5.7: Neighbour channel interference for AP2

Allocation output:

AP1 : Channel 1

AP2 : Channel 11

AP3 : Channel 11

5.1.3 The Node.js scripts

All of the testing done for the Node.js scripts has been manual due to difficulties with asynchronous functions. Critical functions have been isolated and tested. The only bug observed is with the scanning function: when executing the scanning command via iwlist scan an error might occur ("Device or resource busy"). A mechanism to retry if scanning fails has been tried so that the cycle does not get stuck.

5.2 Implementation Considerations & Evaluation

The section looks to debate choices made over the course of the project's implementation.

Why the Ethereum blockchain?

The Ethereum Blockchain platform is by far the most popular one at the time of writing. Because of its popularity, there are a lot of tools and frameworks which make developing Ethereum DApps easier. Moreover, a lot of the issues which may arise during development can be solved relatively easily thanks to the community behind it. Finally, the choice of using Ethereum was mainly down to familiarity with it.

Why Raspberry PIs as Access Points?

The choice of using Raspberry PIs as access points was taken early in the course of the project. The need for a programmable access point is satisfied by the Raspberry PI. Even more so, its size makes it mobile in the sense of placing it in different places and its cost makes it affordable for a cheap setup.

Proof-of-Authority versus Proof-of-Work

Over the course of the project, Proof of Authority was used to facilitate setup and testing. The PoA setup is a good choice in the short term, but in the long term it does not hold as well due to lack of dynamic block limits and bad portability to other clients according to author Szilágyi in EIP-225 [24]. Moreover, Clique PoA does not allow fast or light synchronization modes which can prove to be a problem for the AP nodes which do not have a lot of memory available. Also, if the blockchain network is to be expanded, PoW is more scalable and secure.

Smart contract design considerations

There are several consideration to be made about the current smart contract design. The main problem about it is in regards to the submit scan information function. At the moment, the number of arguments it takes is definitely to big. Smart contracts need to be structured as atomically as possible. Therefore, the function should be restructured into three different functions: one that takes scan information about one channel for one AP (which results in needing to send three separate transactions to submit all the neighboring AP info), one that submits the number of associated clients of the AP and one that submits scan information about the other controlled APs. While this design would be much cleaner, it adds unneeded complexity to the Node.js scripts.

Another consideration to make would be to have the APs submit their own MAC address to add automation and remove the uncertainty of potential mistakes when setting the MAC addresses for each AP. This could add another step to the allocation cycle.

Chapter 6

Conclusion and Future Work

6.1 Proposed Future Work

6.1.1 Extending the controller's functionalities

The implemented system so far already resembles an Wireless LAN Controller. While a WLAN Controller is a centralised point which handles all access points configuration, the blockchain approach offers a decentralised version of the controller. The peer to peer network facilitates inter-AP communication. If implemented successfully, the project could become an open source free alternative to paid WLAN Controllers. The project so far has focused on channel allocation by minimizing interference, but there are more aspects to consider in managing access point.

First of all, it could offer a **"centralised" authentication** function. So far, the APs have their own DHCP servers; having a central point to map MAC addresses to leased IPs could in theory solve the problem, if implemented correctly.

Load balancing is another feature that the controller could offer. Balancing the number of clients associated between nearby APs improves throughput for clients as there are fewer contenders in CSMA/CA.

An interesting feature to explore would be using the blockchain's cryptograhically secure account system for user identification. Users could then be given different privileges. Pairing this up with the blockchain's cryptocurrency system could offer clients the possibility to be prioritised in some of the controller's functions, therefore having better WiFi access.

6.1.2 Improving the DCA's performance and functionality

It is safe to say the current DCA implementation lacks completeness. When APs scan, they only take into account information on channel 1, 6 and 11. Therefore, adjacent channel interference is not taken into account at all. A way to include adjacent channel interference in the current system could be to assign it as interference to the closest non-overlapping channel. This approach would not be the most complete either, so a way to quantify interference taking into account both adjacent and co channel interference should be looked into.

Moreover, the DCA algorithm does not take into account whether two controlled APs are interfering with each other when assigning channel. With current assignments of multiple APs on the same channel being done by the DCA algorithm, the result would be two or more of the controlled access points heavily interfering with each other; by the algorithm's logic, interference is minimized by having these APs sharing the same channel although it is very harming in practice.

6.1.3 Better DApp automation

The project in its current state lacks full automation. A future goal would be to achieve automation to the point of executing a single script that runs a whole allocation cycle. This could be further extended to run regular allocation cycles or to find a criteria by which the allocation is to be kicked off such as an increase in channel utilisation for one of the APs.

6.1.4 Managing the blockchain's size

A current threat to the current system is for the Raspberry PIs to run out of memory. The PoA consensus employed limits the types of nodes to be run. Therefore, switching to PoW in the future would be advised to be able to use the APs as light or fast nodes. Moreover, a blockchain reinitiation routine could be implemented as the information stored in the smart contract is not crucial in any way for deciding future allocations.

6.2 Conclusion

As a conclusion, the paper takes into account the ease of implementing and its efficacy when compared to more standard architectures.

First of all, the blockchain network facilitates the communication between the nodes, but it is unnecessary for the nodes to hold a full copy of the blockchain. The nodes need to be as light as possible to not interfere with its AP functionalities.

Secondly, the implementation of the system has been considerably tedious to the point that a centralised controller approach could have been preferable.

In conclusion, blockchain could be used successfully for automation in the context of Wifi Access Points as the nodes/accounts can be scripted to run on their own. Further exploration of other controller features might showcase the usefulness of the blockchain implementation better.

Chapter 7

Legal, Social, Ethical and Professional Issues

Throughout the project, the British Computing Society's "Code of Conduct & Code of Good Practice" was taken into account. Therefore, the decisions made have thoroughly considered the public interest. The software used in the completion of the project is open-source with the intention being for the solution's accessibility to the public. Moreover, the pieces of code written for the project are entirely my own work and any ideas presented that are outside my area of competency are backed by sources. The main focus of the paper was to explore the compatibility of an upcoming technology trend such as blockchain in the scope of an unexplored domain.

References

[1] Shanhong Liu. Blockchain technology use cases in organizations worldwide as of april 2019. URL https://www.statista.com/statistics/878732/worldwide-use-cases-blockchain-technology/.

[2] Wikipedia. URL https://en.wikipedia.org/wiki/List_of_WLAN_channels#/media/File:2.4_GHz_Wi-Fi_channels_(802.11b,g_WLAN).svg.

[3] Bo Wang, William Wu, and Yongqiang Liu. Dynamic channel assignment in wireless lans. 2008.

[4] Aditya Akella, Glenn Judd, Srinivasan Seshan, and Peter Steenkiste. Self-management in chaotic wireless deployments. 2006.

[5] Matthew Gast. *802.11 Wireless Networks: The Definitive Guide 2nd Edition*. 2005.

[6] Ross Kurose. *Computer Networking: A Top-Down Approach 7th Edition*. 2017.

[7] Arunesh Mishra, Vivek Shrivastava, Dheeraj Agrawal, Suman Banerjee, and Samrat Ganguly. Distributed channel management in uncoordinated wireless environments. 2006.

[8] Helga Balbi, Natalia Fernandes, Felipe Souza, Ricardo Carrano, Celio Albuquerque, Debora Muchaluat-Saade, and Luiz Magalhaes. Centralized channel allocation algorithm for IEEE 802.11 networks. 2012.

[9] Imran Bashir. *Mastering Blockchain: Distributed Ledger Technology, Decentralization, and Smart Contracts Explained, 2nd Edition*. 2018.

[10] Miguel Castro and Barbara Liskov. Practical byzantine fault tolerance, 1999.

[11] Ethereum Foundation. What is proof of stake, 2019. URL https://github.com/ethereum/wiki/wiki/Proof-of-Stake-FAQ#what-is-proof-of-stake.

[12] Arati Baliga. Understanding blockchain consensus models. 2017.

[13] Mayank Raikwar, Danilo Gligoroski, and Katina Kralevska. SoK of Used Cryptography in Blockchain. 2019.

[14] John Evans. Blockchain nodes: An in-depth guide. URL https://nodes.com/.

[15] Ethereum Foundation. A next-generation smart contract and decentralized application platform, 2019. URL https://github.com/ethereum/wiki/wiki/White-Paper.

[16] Dr. Gavin Wood. Ethereum: A secure decentralised generalised transaction ledger, 2019. URL https://ethereum.github.io/yellowpaper/paper.pdf.

[17] block.one. Eos.io technical white paper v2. URL https://github.com/EOSIO/Documentation/blob/master/TechnicalWhitePaper.md.

[18] Jouni Malinen. hostapd: IEEE 802.11 AP, IEEE 802.1X/WPA/WPA2/EAP/RADIUS Authenticator. URL http://w1.fi/hostapd/.

[19] Simon Kelley. Dnsmasq. URL http://www.thekelleys.org.uk/dnsmasq/doc.html.

[20] Raspberry Connect. Raspberry Pi - Hotspot/Access Point dhcpcd method. URL https://www.raspberryconnect.com/projects/65-raspberrypi-hotspot-accesspoints/168-raspberry-pi-hotspot-access-point-dhcpcd-method.

[21] The go-ethereum Authors. Using geth: Command-line options. URL https://geth.ethereum.org/docs/interface/command-line-options.

[22] Solidity documentation. URL https://solidity.readthedocs.io/en/v0.6.4/.

[23] Remix ide. URL https://remix.ethereum.org/.

[24] Péter Szilágyi. Eip 225: Clique proof-of-authority consensus protocol. URL https://eips.ethereum.org/EIPS/eip-225.

[25] Andreas M. Antonopoulos and Dr. Gavin Wood. *Mastering Ethereum: Building Smart Contracts and DApps*. 2018.

[26] 2.4ghz spectrum. URL https://www.msdist.co.uk/support/articles/spectrum-and-licensing/uk-spectrum-2-4-ghz.

[27] web3.js - ethereum javascript api. URL https://web3js.readthedocs.io/en/v1.2.6/.

Appendix A

Configuration files

A.1 Access Point configuration

A.1.1 Hostapd configuration

```
driver=nl80211

ctrl_interface=/var/run/hostapd

ctrl_interface_group=0

auth_algs=1

wpa_key_mgmt=WPA-PSK

beacon_int=100

ssid={Make sure it is the same for all APs; eg:WifiBlockchain}

channel={Choose 1, 6 or 11 for initial setup}

hw_mode=g

ieee80211n=0

wpa_passphrase={same password for all APs}

interface=wlan0

wpa=2

wpa_pairwise=CCMP

country_code=GB

ignore_broadcast_ssid=0
```

A.2 Controller contract ABI file


```
[{
    "inputs": [],
    "stateMutability": "nonpayable",
    "type": "constructor"
}, {
    "anonymous": false,
    "inputs": [],
    "name": "AllHaveSubmitted",
    "type": "event"
}, {
    "anonymous": false,
    "inputs": [{
        "indexed": true,
        "internalType": "address",
        "name": "_ethaddr",
        "type": "address"
    }, {
        "indexed": false,
        "internalType": "uint8",
        "name": "channel",
        "type": "uint8"
    }],
    "name": "ChangeChannel",
    "type": "event"
}, {
    "anonymous": false,
    "inputs": [],
    "name": "SubmissionsRequested",
    "type": "event"
}, {
    "inputs": [{
        "internalType": "uint256",
        "name": "",
        "type": "uint256"
    }],
    "name": "ListOfAPs",
    "outputs": [{
        "internalType": "address",
        "name": "ethaddr",
        "type": "address"
```

```
    }, {
        "internalType": "uint8",
        "name": "channel",
        "type": "uint8"
    }, {
        "internalType": "uint256",
        "name": "noOfAssociatedClients",
        "type": "uint256"
    }, {
        "internalType": "string",
        "name": "MACaddr",
        "type": "string"
    }],
    "stateMutability": "view",
    "type": "function"
}, {
    "inputs": [{
        "internalType": "address",
        "name": "",
        "type": "address"
    }],
    "name": "accessPointChAllocation",
    "outputs": [{
        "internalType": "uint8",
        "name": "",
        "type": "uint8"
    }],
    "stateMutability": "view",
    "type": "function"
}, {
    "inputs": [{
        "internalType": "address",
        "name": "_addr",
        "type": "address"
    }, {
        "internalType": "uint8",
        "name": "_currentChannel",
        "type": "uint8"
    }, {
        "internalType": "uint8",
        "name": "_noOfAssociatedClients",
        "type": "uint8"
```

```
    }, {
        "internalType": "string",
        "name": "MACaddr",
        "type": "string"
    }],
    "name": "addAP",
    "outputs": [],
    "stateMutability": "nonpayable",
    "type": "function"
}, {
    "inputs": [],
    "name": "allocateChannels",
    "outputs": [],
    "stateMutability": "nonpayable",
    "type": "function"
}, {
    "inputs": [],
    "name": "requestSubmissions",
    "outputs": [],
    "stateMutability": "nonpayable",
    "type": "function"
}, {
    "inputs": [{
        "internalType": "address",
        "name": "",
        "type": "address"
    }, {
        "internalType": "uint256",
        "name": "",
        "type": "uint256"
    }],
    "name": "scan",
    "outputs": [{
        "internalType": "uint8",
        "name": "channelNo",
        "type": "uint8"
    }, {
        "internalType": "uint256",
        "name": "totalInterference",
        "type": "uint256"
    }],
    "stateMutability": "view",
```

```
    "type": "function"
}, {
    "inputs": [{
        "internalType": "uint8[3]",
        "name": "channelNo",
        "type": "uint8[3]"
    }, {
        "internalType": "uint256[3]",
        "name": "totalInterference",
        "type": "uint256[3]"
    }, {
        "internalType": "uint256",
        "name": "_noOfAssociatedClients",
        "type": "uint256"
    }, {
        "internalType": "string[2]",
        "name": "MAC_addresses_ControlledAP",
        "type": "string[2]"
    }, {
        "internalType": "uint256[2]",
        "name": "interferenceControlledAP",
        "type": "uint256[2]"
    }],
    "name": "submitScanInfo",
    "outputs": [],
    "stateMutability": "nonpayable",
    "type": "function"
}, {
    "inputs": [{
        "internalType": "address",
        "name": "",
        "type": "address"
    }],
    "name": "submitted",
    "outputs": [{
        "internalType": "bool",
        "name": "",
        "type": "bool"
    }],
    "stateMutability": "view",
    "type": "function"
}]
```

Appendix B

Source Code

B.1 JavaScript Source Code

These are the following scripts used in the project.

B.1.1 mineWhenTransactions.js

```
var mining_threads = 1

function checkWork() {
    if (eth.getBlock("pending").transactions.length > 0) {
        if (eth.mining) return;
        console.log("== Pending transactions! Mining...");
        miner.start(mining_threads);
    } else {
        miner.stop();
        console.log("== No transactions! Mining stopped.");
    }
}

eth.filter("latest", function(err, block) { checkWork(); });
eth.filter("pending", function(err, block) { checkWork(); });

checkWork();
```

B.1.2 deployContract.js

```
const Web3 = require('web3');
const net = require('net');
const {
    exec
} = require('child_process')
const fs = require('fs');
let ABI;
let bin;


class AP {
    constructor(ethAddr, MACaddr) {
        this.ethAddr = ethAddr;
        this.MACaddr = MACaddr;
    }
}
const web3 = new Web3({CORRECT IPC PATH}, net);
let account;
let contractToDeploy;
let APsToRegister = [new AP('0x1..', "MACADDRESS1"), new AP('0x2..',"MACADRESS2"
    ), new AP('0x3..', "MACADRESS3")
]; //change AP information accordingly.

exec("solcjs --abi --bin Controller.sol").on("close", (err, res) => {
    if (err) {
        console.log(err)
        process.exit(1);
    }
    read();
    unlockAccAndDeploy();
})

let gasPrice;
let getGasPrice = () => {
    web3.eth.getGasPrice().
    then((averageGasPrice) => {
        console.log("Average gas price: " + averageGasPrice);
        gasPrice = averageGasPrice;
    }).
    catch(console.error);
```

```
}


let unlockAccAndDeploy = async () => {

    web3.eth.getAccounts().then(res => {
        account = res[0];
        web3.eth.personal.unlockAccount(account, "test").
        then(() => {
            console.log('Account unlocked');
            contractToDeploy = new web3.eth.Contract(JSON.parse(ABI));
            deploy();
        }).catch(console.error);
    }).catch(err => console.log(err))

}

let gas;

let deploy = () => {
    contractToDeploy.deploy({
        data: '0x' + bin
    }).estimateGas({
        from: account
    }).then((estimatedGas) => {
        console.log("Estimated gas: " + estimatedGas);
        gas = estimatedGas * 10000000;
        console.log("Gas for contract:", gas)
    }).catch(err => console.log(err))

    getGasPrice();

    contractToDeploy.deploy({
            data: '0x' + bin
        }).send({
            from: account,
            gas: gas,
            gasPrice: parseInt(gasPrice)
        }, function(error, transactionHash) {
            if (error) console.log(error);
            console.log("Transaction Hash:", transactionHash)
        })
```

```
82          .on('error', function(error) {
83              console.log(error)
84          })
85          .on('receipt', function(receipt) {
86              console.log("New contract address: ", receipt.contractAddress)
87          })
88          .on('confirmation', function(confirmationNumber, receipt) {
89              console.log("Confirmation number:", confirmationNumber);
90              if (confirmationNumber == 1) {
91                  //do nothing...
92              }
93          })
94          .then((contractInstance) => {
95              register(APsToRegister[0], contractInstance).then(
96                  register(APsToRegister[1], contractInstance).then(
97                      register(APsToRegister[2], contractInstance)
98                  ).catch(err => console.log(err))
99              ).catch(err => console.log(err))
100         }).catch(err => console.log(err));
101 }
102
103 let read = async () => {
104     ABI = fs.readFileSync('Controller_sol_Controller.abi', 'utf8');
105     bin = fs.readFileSync('Controller_sol_Controller.bin', 'utf8');
106 }
107
108 const register = async (AP, myControllerContract) => {
109
110     getGasPrice();
111     myControllerContract.methods.addAP(AP.ethAddr, 2, 0, AP.MACaddr).estimateGas
            ({
112             from: account
113         })
114         .then((estimatedGas) => {
115             console.log("Estimated gas: " + estimatedGas);
116             gas = estimatedGas + estimatedGas * 50 / 100;
117             console.log("Gas:", gas);
118             myControllerContract.methods.addAP(AP.ethAddr, 2, 0, AP.MACaddr).
                    send({
119                     from: account,
120                     gas: parseInt(gas),
121                     gasPrice: gasPrice
```

```
                })
                .on('confirmation', function(confirmationNumber, receipt) {
                    console.log("Confirmation number:", confirmationNumber);
                })
                .then(() => {
                    console.log("submitted");
                })

        }).catch(err => console.log(err))
}
```

B.1.3 sealer.js

```
const Web3 = require('web3')
const net = require('net');
const fs = require('fs')
const web3 = new Web3({CORRECT IPC PATH}, net);

let ABI = JSON.parse(fs.readFileSync('Controller_sol_Controller.abi', 'utf8'));
    \\ read ABI from a file, alternatively, ABI can be stored as a variable.
const contractAddress = "0x.."; //correct contract address

let accountAddress;
let myControllerContract;

const errHandler = (err) => {
    console.log(err)
}

web3.eth.getAccounts().then(res => {

    let gasPrice;
    accountAddress = res[0];

    myControllerContract = new web3.eth.Contract(ABI, contractAddress, {
        from: accountAddress
    });

    const unlockAccount = async () => {
        web3.eth.personal.unlockAccount(accountAddress, "test", 0).then(console.
            log("Account unlocked")).catch(errHandler)
    }
```

```
28    const getGasPrice = async () => {
29        web3.eth.getGasPrice().then((averageGasPrice) => {
30            console.log("Average gas price: " + averageGasPrice);
31            gasPrice = averageGasPrice;
32        }).catch(console.error);
33    }
34
35
36    const request = () => {
37        unlockAccount().then(
38            myControllerContract.methods.requestSubmissions().send({
39                from: accountAddress
40            }).then(() => {
41                console.log("Requested");
42            }).catch(errHandler)
43        ).catch(errHandler)
44    }
45    request();
46
47
48    const startAllocation = async (gasPrice) => {
49
50        console.log("Allocation started")
51        myControllerContract.methods.allocateChannels().estimateGas({
52            from: accountAddress
53        }).then((estimatedGas) => {
54            console.log("Estimated gas: " + estimatedGas);
55            gas = estimatedGas + estimatedGas * 20 / 100;
56            unlockAccount().then(
57                myControllerContract.methods.allocateChannels().send({
58                    from: accountAddress,
59                    gas: parseInt(gas),
60                    gasPrice: gasPrice
61                })
62                .on('confirmation', function(confirmationNumber, receipt) {
63                    console.log("Confirmation number:", confirmationNumber);
64                })
65                .then(() => {
66                    console.log("submitted");
67                })
68            ).catch(errHandler)
69
```

```
        }).catch(errHandler)
    }

    myControllerContract.events.AllHaveSubmitted((err, event) => {
        if (err) console.log(err);
        unlockAccount().then(
            getGasPrice().then(gasPrice => startAllocation(gasPrice))
        ).catch(errHandler)
    })

}).catch(errHandler)
```

B.1.4 AP.js

```
const {
    exec
} = require('child_process')
const fs = require('fs')
const contractABI = fs.readFileSync('Controller_sol_Controller.abi', 'utf8');
const contractAddress = "0x.." //replace accordingly
const Web3 = require('web3')
const net = require('net');
const web3 = new Web3({CORRECT IPC PATH}, net);
let myControllerContract;
let accountAddress;


class AccessPointInfo {
    constructor(MACaddr, channel, signalStrengh) {
        this.MACaddr = MACaddr;
        this.channel = channel;
        this.signalStrengh = signalStrengh;
    }
}

class ChannelInfo {
    constructor(interferenceValue, count) {
        this.interferenceValue = interferenceValue;
        this.count = count;
    }
}

```

```
29
30 async function isUnlocked(address) {
31     try {
32         await web3.eth.sign("", address);
33     } catch (e) {
34         return false;
35     }
36     return true;
37 }
38
39 const main = async () => {
40
41     const errHandler = (err) => {
42         console.log(err)
43         console.log(isUnlocked(accountAddress))
44     }
45     web3.eth.getAccounts().then(res => {
46         accountAddress = res[0];
47         console.log("Account address is:", accountAddress)
48         console.log("Contract address is:", contractAddress)
49         myControllerContract = new web3.eth.Contract(JSON.parse(contractABI),
               contractAddress, {
50             from: accountAddress
51         })
52         const unlockAccount = async () => {
53             web3.eth.personal.unlockAccount(accountAddress, "test", 0).then(
                   console.log("Account unlocked")).catch(errHandler)
54         }
55         const lockAccount = async () => {
56             web3.eth.personal.lockAccount(accountAddress).then(console.log("
                   Account locked")).catch(errHandler)
57         }
58         const getGasPrice = async () => {
59             web3.eth.getGasPrice().then((averageGasPrice) => {
60                 console.log("Got gas price")
61                 return averageGasPrice;
62             }).catch(errHandler);
63         }
64
65         const submit = async (result, gasPrice) => {
66             console.log("Started submission")
67
```

```
68              myControllerContract.methods.submitScanInfo(result[0][0], result
                    [0][1], result[1], result[0][2], result[0][3]).estimateGas({
69                  from: accountAddress
70              }).then((estimatedGas) => {
71                  console.log("Estimated gas: " + estimatedGas);
72                  gas = estimatedGas + estimatedGas * 20 / 100;
73
74                  myControllerContract.methods.submitScanInfo(result[0][0], result
                        [0][1], result[1], result[0][2], result[0][3]).send({
75                      from: accountAddress,
76                      gas: parseInt(gas),
77                      gasPrice: gasPrice
78                  })
79                  .on('confirmation', function(confirmationNumber, receipt) {
80                      console.log("Confirmation number:", confirmationNumber);
81                  })
82                  .then(() => {
83                      console.log("submitted");
84                      lockAccount();
85                  })
86
87              }).catch(errHandler)
88
89          }
90
91          myControllerContract.events.SubmissionsRequested((err, event) => {
92              if (err) console.log(err);
93              unlockAccount().then(
94                  getGasPrice().then(gasPrice => startSubmission(gasPrice))
95              ).catch(errHandler)
96          })
97
98          myControllerContract.events.ChangeChannel((err, event) => {
99              if (err) console.log(err);
100
101             if (event.returnValues._ethaddr == accountAddress) {
102                 updateChannel(event.returnValues.channel);
103             }
104         })
105
106         const startSubmission = async (gasPrice) => {
107             scan().then((output) => {
```

```
                submit(output, gasPrice)
            }).catch(errHandler)
        }
    }).catch(errHandler)
}

const calculateResults = async (output) => {

    output = output.replace(/\s\s+/g, ' ');
    output = output.trim().split("Cell").slice(1);

    let arrayOfAccessPointsInfo = [];
    let arrayOfControlledAccessPointsInfo = [];
    output.forEach(entry => {
        if (/Channel:[1-9] |Channel:[1][0-1] /g.test(entry) == true) {
            let splitEntry = entry.split(" ")
            let signalQuality = splitEntry[12].split("-")[1]
            let signalPercentage = (110 - signalQuality) * 10 / 7 //convert x/70
                 to percentage out of 100
            let newAccessPoint = new AccessPointInfo(
                splitEntry[4],
                splitEntry[5].split(":")[1],
                signalPercentage
            )
            if (splitEntry[16].split('"')[1] != "WifiBlockchain")
                arrayOfAccessPointsInfo.push(newAccessPoint);
            else {
                arrayOfControlledAccessPointsInfo.push(newAccessPoint)
            }
        }
    })


    let mapOfChannels = new Map();

    for (let i = 1; i <= 11; i++) {
        mapOfChannels.set(i, new ChannelInfo(0, 0))
    }

    arrayOfAccessPointsInfo.forEach(entry => {

        let entryChannel = parseInt(entry.channel);
```

```
149         let currentCount = mapOfChannels.get(entryChannel).count;
150         let currentInterference = mapOfChannels.get(entryChannel).
                interferenceValue;
151 
152         mapOfChannels.set(entryChannel, new ChannelInfo(currentInterference +
                entry.signalStrengh, currentCount + 1))
153     })
154 
155     let channels = [1, 6, 11]
156     let interf = []
157     let stations = []
158     let controlledMACaddresses = [];
159     let controlledInterf = [];
160 
161     channels.forEach(channel => {
162         let channelInf = mapOfChannels.get(channel);
163         interf.push(parseInt(channelInf.interferenceValue));
164         stations.push(channelInf.count);
165     })
166     arrayOfControlledAccessPointsInfo.forEach(entry => {
167 
168         controlledMACaddresMACaddresses.push(entry.MACaddr);
169         controlledInterf.push(entry.signalStrength);
170 
171     })
172 
173 
174     return [channels, interf, controlledMACaddresses, controlledInterf];
175 }
176 
177 const updateChannel = async (channelNo) => {
178 
179     const path = "/etc/hostapd/hostapd.conf"
180     exec('cat ' + path, (error, stdout, stderr) => {
181         if (error) {
182             console.warn(error);
183         }
184         console.log(stdout);
185         let changedOutput = stdout.replace(/channel=[1-9]|channel:[1][0-1]/g, "
                channel=" + channelNo);
186         exec(`sudo echo` + changedOutput + ` > '/etc/hostapd/hostapd.conf'`, (
                error, stdout, stderr) => {
```

```
            if (error) {
                console.warn(error);
            }

            fs.writeFile(path, changedOutput, err => {
                if (err) throw err;
                console.log("Config succesfully saved!");
                exec('systemctl restart hostapd', (error, stdout, stderr) => {
                    if (error) {
                        console.warn(error);
                    }
                    console.log("Hostapd service restarted, channel changed to "
                         + channelNo);
                })
            })
        })


    })
}
const scan = async () => {
    let outputScan;
    let outputClients;

    function execScan() {
        return new Promise((resolve, reject) => {
            try {
                exec('sudo iwlist wlan0 scan', (error, stdout, stderr) => {
                    if (error) {
                        console.warn(error);
                        reject(error)
                    }
                    resolve(stdout);
                });
            } catch (err) {
                setTimeout(() => {
                    console.log("Timeout finished");
                    execScan();
                }, 5000);
            }
        });
    }
```

```
228
229     function execCmd() {
230         return new Promise((resolve, reject) => {
231             try {
232                 exec('sudo hostapd_cli all_sta', (error, stdout, stderr) => {
233                     if (error) {
234                         console.warn(error);
235                         reject(error)
236                     }
237                     resolve(stdout);
238                 });
239             } catch (err) {
240                 console.log(err);
241             }
242         });
243     }
244
245     outputScan = await execScan().then();
246     outputClients = await execCmd();
247     let calculatedOutput = calculateResults(outputScan);
248     let re = /^([0-9A-Fa-f]{2}[:-]){5}([0-9A-Fa-f]{2})$/g;
249     let noStations = (outputClients.match(re) || []).length;
250     return [calculatedOutput, noStations];
251 }
252
253
254
255 main();
```

B.2 Solidity Source Code

B.2.1 Controller.sol

```
pragma solidity ^0.6.4;
pragma experimental ABIEncoderV2;

contract Controller{

    address owner;

    struct AP{
        address ethaddr;
        uint8 channel;
        uint noOfAssociatedClients;
        string MACaddr;
    }

    struct scanInfo{
        uint8 channelNo;
        uint256 totalInterference;
    }

    struct controlledAPscanInfo{
        string MACaddr;
        uint256 interference;
    }

    mapping(address => scanInfo[3]) public scan;
    mapping(address => bool) public submitted;
    mapping(address => uint8) public accessPointChAllocation;
    mapping(address => string) ethAddrToMACaddr;
    mapping(string => address) MACaddrtoEthAddr;
    mapping(address => controlledAPscanInfo[2]) otherAPscanInfo;
    mapping(address => uint256) priority;

    event AllHaveSubmitted();
    event SubmissionsRequested();
    event ChangeChannel(address indexed _ethaddr, uint8 channel);

    AP[] public ListOfAPs;

```

```
39      modifier ifNotRegistered(address _ethaddr){
40          require(
41              checkIfAlreadyRegistered(_ethaddr),
42              "Address is already registered"
43          );
44          _;
45      }
46
47      modifier onlyRegisteredAPs(){
48          require(
49              checkIfValidAddress(msg.sender),
50              "Only registered APs can call this function."
51          );
52          _;
53      }
54
55      modifier onlyOwner(){
56          require(msg.sender == owner, "Only the owner may call this function");
57          _;
58      }
59      constructor () public{
60
61          owner = msg.sender;
62      }
63
64      function addAP(address _addr, uint8 _currentChannel, uint8
            _noOfAssociatedClients, string memory MACaddr) public
65      {
66          ListOfAPs.push(AP(_addr, _currentChannel, _noOfAssociatedClients,
                MACaddr));
67          ethAddrToMACaddr[_addr] = MACaddr;
68          MACaddrtoEthAddr[MACaddr] = _addr;
69          priority[_addr] = now;
70      }
71
72
73
74      function submitScanInfo(uint8[3] memory channelNo, uint256[3] memory
            totalInterference, uint _noOfAssociatedClients,
75                              string[2] memory MAC_addresses_ControlledAP, uint256
                                    [2] memory interferenceControlledAP)
76      public onlyRegisteredAPs(){
```

```
77
78          otherAPscanInfo[msg.sender][0] = controlledAPscanInfo(
                MAC_addresses_ControlledAP[0], interferenceControlledAP[0]);
79          otherAPscanInfo[msg.sender][1] = controlledAPscanInfo(
                MAC_addresses_ControlledAP[1], interferenceControlledAP[1]);
80
81
82          for(uint8 i = 0; i < 3; i++)
83          {
84              scan[msg.sender][i] = scanInfo(channelNo[i], totalInterference[i]);
85
86              if(ListOfAPs[i].ethaddr == msg.sender)
87                  ListOfAPs[i].noOfAssociatedClients = _noOfAssociatedClients;
88          }
89
90
91
92          submitted[msg.sender] = true;
93
94
95          if(checkIfAllSubmitted() == true){
96              emit AllHaveSubmitted();
97          }
98      }
99
100     function requestSubmissions() public onlyOwner {
101         emit SubmissionsRequested();
102     }
103
104     function allocateChannels() public onlyOwner{
105
106         AP[] memory discoloredNodes = ListOfAPs;
107
108         for(uint8 i = 0; i < discoloredNodes.length - 1; i++)
109         {
110             uint8 currentNodeSD = getSaturationDegree(discoloredNodes[i].ethaddr
                    );
111             uint8 nextNodeSD = getSaturationDegree(discoloredNodes[i+1].ethaddr)
                    ;
112             if(currentNodeSD < nextNodeSD)
113             {
114                 AP memory temp = discoloredNodes[i+1];
```

```
115                 discoloredNodes[i+1] = discoloredNodes[i];
116                 discoloredNodes[i] = temp;
117             }
118             if(currentNodeSD == nextNodeSD)
119             {
120 
121                 uint currentNodeAC = getNoOfAssociatedClients(discoloredNodes[i
                        ].ethaddr);
122                 uint nextNodeAC = getNoOfAssociatedClients(discoloredNodes[i+1].
                        ethaddr);
123 
124                 if(currentNodeAC < nextNodeSD)
125                 {
126                     AP memory temp = discoloredNodes[i+1];
127                     discoloredNodes[i+1] = discoloredNodes[i];
128                     discoloredNodes[i] = temp;
129                 }
130                 if(currentNodeAC == nextNodeAC)
131                 {
132                     uint currentNodeTimestamp = priority[discoloredNodes[i].
                            ethaddr];
133                     uint nextNodeTimestamp = priority[discoloredNodes[i+1].
                            ethaddr];
134                     if(currentNodeTimestamp < nextNodeTimestamp)
135                     {
136                         AP memory temp = discoloredNodes[i+1];
137                         discoloredNodes[i+1] = discoloredNodes[i];
138                         discoloredNodes[i] = temp;
139                     }
140                 }
141             }
142 
143         }
144 
145         for(uint8 j = 0; j < discoloredNodes.length; j++)
146         {
147             AP memory selectedNode = discoloredNodes[j];
148 
149             uint8 freeChannel = getFreeChannel(selectedNode.ethaddr);
150 
151             if(freeChannel != 0)
152             {
```

```
153                 accessPointChAllocation[selectedNode.ethaddr] = freeChannel;
154                 addInterferenceToScanInfo(selectedNode.ethaddr, freeChannel);
155 
156             }
157             else
158             {
159                 uint8 channelWithLeastInterference =
                        getChannelWithLeastInterference(selectedNode.ethaddr);
160                 accessPointChAllocation[selectedNode.ethaddr] =
                        channelWithLeastInterference;
161                 addInterferenceToScanInfo(selectedNode.ethaddr,
                        channelWithLeastInterference);
162             }
163         }
164 
165         for(uint8 k = 0; k < ListOfAPs.length; k++){
166 
167                 if(ListOfAPs[k].channel != accessPointChAllocation[ListOfAPs[k].
                        ethaddr]){
168                     ListOfAPs[k].channel = accessPointChAllocation[ListOfAPs[k].
                            ethaddr];
169                     emit ChangeChannel(ListOfAPs[k].ethaddr, ListOfAPs[k].
                            channel);
170                 }
171                 submitted[ListOfAPs[k].ethaddr] = false;
172         }
173 
174 
175     }
176 
177 
178     function getChannelWithLeastInterference(address _ethaddr) private view
            returns (uint8 channelNo){
179 
180         uint256 minValue = scan[_ethaddr][0] .totalInterference;
181         channelNo = scan[_ethaddr][0].channelNo;
182 
183         for(uint8 i = 1; i < 3; i++)
184         {
185             if(scan[_ethaddr][i].totalInterference < minValue)
186             {
187                 minValue = scan[_ethaddr][i].totalInterference;
```

```
188                 channelNo = scan[_ethaddr][i].channelNo;
189             }
190         }
191 
192         return channelNo;
193     }
194 
195 
196 
197     function checkIfValidAddress(address addr) private view returns (bool
        isValid){
198 
199         isValid = false;
200         for(uint8 i = 0 ; i < ListOfAPs.length; i++)
201         {
202             if(addr == ListOfAPs[i].ethaddr){
203                 isValid = true;
204                 break;
205             }
206         }
207 
208         return isValid;
209     }
210 
211 
212     function checkIfAlreadyRegistered(address _ethaddr) private view returns (
        bool isAlreadyRegistered){
213 
214         isAlreadyRegistered = false;
215         for(uint8 i = 0 ; i < ListOfAPs.length; i++)
216         {
217             if(_ethaddr == ListOfAPs[i].ethaddr){
218                 isAlreadyRegistered = true;
219             }
220         }
221 
222         return isAlreadyRegistered;
223     }
224 
225     function checkIfAllSubmitted() private view returns (bool allSubmitted){
226 
227         allSubmitted = true;
```

```
228         for(uint8 i = 0 ; i < ListOfAPs.length; i++)
229         {
230             if(submitted[ListOfAPs[i].ethaddr] == false)
231                 allSubmitted = false;
232 
233             if(allSubmitted == false)
234                 break;
235         }
236 
237         return allSubmitted;
238     }
239 
240     function getFreeChannel(address _ethaddr) private view returns (uint8
        freeChannel) {
241 
242         for(uint8 j = 0; j < 3; j++)
243         {
244                 if(scan[_ethaddr][j].totalInterference == 0){
245                     return scan[_ethaddr][j].channelNo;
246 
247                 }
248         }
249 
250         return 0;
251 
252     }
253 
254     function getNoOfAssociatedClients(address _ethaddr) private view returns (
        uint256) {
255 
256         for(uint8 j = 0; j < 3; j++)
257         {
258                 if(ListOfAPs[j].ethaddr == _ethaddr){
259                     return ListOfAPs[j].noOfAssociatedClients;
260                 }
261         }
262 
263         return 0;
264 
265     }
266 
267 
```

```
268     function getSaturationDegree(address _ethaddr) private view returns (uint8){
269
270         uint8 counter = 0;
271         for(uint i = 0; i < 3; i ++)
272         {
273           if(scan[_ethaddr][i].totalInterference != 0)
274             counter++;
275         }
276
277         return counter;
278
279     }
280
281     function addInterferenceToScanInfo(address _ethaddr, uint8 channel) private
            {
282
283
284         for(uint8 i = 0; i < 3; i++)
285         {
286             if(ListOfAPs[i].ethaddr != _ethaddr)
287             {
288                 uint256 interferenceToAdd = 0;
289
290                 for(uint j = 0; j < 2 ; j ++)
291                 {
292                     if(keccak256(abi.encodePacked(otherAPscanInfo[ListOfAPs[i].
                            ethaddr][j].MACaddr)) == keccak256(abi.encodePacked(
                            ethAddrToMACaddr[_ethaddr])))
293                     {
294                         interferenceToAdd = otherAPscanInfo[ListOfAPs[i].ethaddr
                                ][j].interference;
295                     }
296                 }
297
298                 if(interferenceToAdd == 0) {break;}
299                 else{
300                     if(channel == 1)
301                     {
302                         scan[ListOfAPs[i].ethaddr][0].totalInterference = scan[
                                ListOfAPs[i].ethaddr][0].totalInterference +
                                interferenceToAdd;
303                     }
```

```
304                     if(channel == 6)
305                     {
306                         scan[ListOfAPs[i].ethaddr][1].totalInterference = scan[
                                ListOfAPs[i].ethaddr][1].totalInterference +
                                interferenceToAdd;
307                     }
308                     if(channel == 11)
309                     {
310                         scan[ListOfAPs[i].ethaddr][2].totalInterference = scan[
                                ListOfAPs[i].ethaddr][2].totalInterference +
                                interferenceToAdd;
311                     }
312                 }
313             }
314         }
315     }
316
317 }
```