An investigation into the use of blockchain to record NHS GOSH meetings

Shubham Bakshi *

MEng Computer Science

Supervisor: Dr. Graham Roberts

Special thanks to Sirvan Almasi and Prof. Neil Sebire

Submission date: 29 April 2019

Abstract

Multidisciplinary Team (MDT) meetings are conducted by GOSH to make medical decisions about patients. These meetings have real consequences for the patient, and thus there is a need to record them in a way that they can be audited later. Traditional databases are not sufficient to guarantee the authenticity and integrity of recorded events that have taken place in a meeting, as participants can refute, and even accuse the software system of tampering with the data. This project investigates if blockchain can provide a solution that delivers on the guarantees.

The project provides a proof of concept implementation of a system that records a MDT meeting, and then stores the very minimum amount of data on the Ethereum blockchain while maintaining the integrity promises. The data stored in the blockchain can then be used by any participant to verify the occurrence of the events. The project also integrates with DeeID to provide identity for the users in the system, and thus guarantee authenticity. All in all, it is concluded that blockchain can be used to solve the stated problem, at a reasonable cost.

Contents

List of Figures

1 Introduction

This project is about designing a system to record medical decisions that are made in meetings, with the help of blockchain, so they can be audited later.

1.1 Motivation and Problem Description

Great Ormond Street Hospital (GOSH) is a NHS hospital for children that treats many conditions and diseases including cancer. Cancer is a complex illness, and patients have to follow complex pathways from their diagnosis to their treatment. The decision-making process in the patient's pathway are done though a meeting consisting of a group of professionals from different teams where they discuss the diagnosis, treatment and the management of patients one-by-one. This is known as a Multi-disciplinary team (MDT) meeting [1].

Since these meetings have real consequences for the patient, there is a need to record the decisions that are made in the meeting to allow the medical staff to refer back to them if there are any disputes or disagreements in the future. A traditional solution is to have the meeting video recorded by GOSH, and make them available for auditing. However, there are two issues with such an approach; (i) the solution requires the participants to trust a single entity who is responsible for recording and storing the video; (ii) it is difficult to record certain actions through video such as voting processes, passive disapproval of comments and more. Although the latter can be rectified through software systems, there is still the issue of trust between the participants of the meeting and the system. Furthermore, with current advancements in artificial intelligence and image processing, even videos can be tampered with to generate fake footage [2]. The participant must therefore trust the system completely to provide the correct data, which may not be the case when there is a dispute. For example, if a participant is under investigation for an event that took place in a meeting, and the system is audited, the participant can just refute the events claiming the data is incorrect or worse, tampered with by the system or an external adversary. This argument also flows vice versa, as a participant could also be falsely accused of incidents and decisions that they might not have been involved with.

The solution to this problem needs to provide strong notions of data integrity and support the non-repudiation of events that take place in a meeting. In order to provide these properties, it is necessary to understand more about the meetings, and the nature of the events that take place. The project was hence carried out in close collaboration with Professor Neil Sebire, who is a medical professional at NHS GOSH, and has been the host of many MDT meetings. He mentioned the following events should be considered for recording:

- Attendance: Recording which participants attended the MDT meeting and at what time, will allow the auditors to know what information the participant was exposed to and what decisions they were involved in.
- Votes and Polls: Most of the decisions that are made in the meeting usually involve some sort of democratic process (such as a show of hands or verbally saying "yes"). This process can be formalised by introducing a voting system, where all or a subset of participants vote and a consensus is reached. How each person votes, as well as the tally of the votes will be recorded in the system..
- Comments: Participants may leave arbitrary comments, so they are officially recorded in the system. These comments may be left by anyone participating in the meeting.

- Images: Images of scans and results that is displayed in the meeting should be stored, so auditors know exactly what information the participants were exposed to.

Prof. Sebire was also consulted on what promises the solution should provide, that would be of interest to GOSH, and what existing technologies and systems should be considered. It was understood that the investigation should be about a system that is able to store meeting data, which all the participants agree on; and then later, when these events are audited, the system should be able to provide guarantees about the integrity and authenticity of the data. It's inevitable that adversaries will try to corrupt the data if there is enough incentive. So, such attempts should be detectable, and perhaps even correctable. There is also a parallel project in GOSH that aims to provide secure identities for patients and staff, known as Omnee (DeeID), which should also be investigated to see if it can provide any value to this project. During the consultation, one of the suggested approaches to solve the problem, was to look into blockchain, and how it can be utilized to deliver on the guarantees. Blockchain itself is quite a broad term, and can refer to anything from hash pointers and the immutability of data, to consensus protocol and the lack of trust in central entity. However, a technology that is of particular interest is decentralised apps (dApps), which uses Ethereum (a decentralised computing platform) and smart contracts to achieve some of the properties that we are pursuing in this project. This is further investigated in section 2.1.

So in summary, the premise of this project is to:

> Investigate the use of blockchain for storing data, in a way that guarantees the authenticity and integrity of the data, when audited.

1.2 Aims and Objectives

- Investigate how MDT meetings are currently held and formalize the format; as well as understand the current standards for data sharing in NHS.
- Learn about blockchain and the cryptography concepts behind it.
- Learn about Ethereum, smart contracts, and how they are used in decentralised apps.
- Acquire experience in writing smart contracts in Ethereum.
- Acquire experience with web frameworks for web/UI development.
- Investigate decentralised identity and how every patient and staff can be represented in the system.

1.3 Goals and Deliverables

In order of importance:

- An architecture for storing meeting metadata in a way that can be audited later, with data integrity, authenticity and accountability in mind.
- A proof-of-concept implementation of the architecture (including servers).
- A proof-of-concept of a web app (UI) that will be used by the users to participate in the meetings.
- A login system that utilizes DeeID identification scheme.

1.4 Report Outline

An outline of the report and it's chapters are described below:

- **Chapter 2** discusses the research that was done to understand the problem better and the background needed to understand the solution. It takes a deeper dive into how MDT meetings are conducted, and also analyses the different technologies including blockchain, decentralised identity, smart contracts and more that is used in the project.
- **Chapter 3** analyses the abstract requirements gathered from the initial meeting, and formalises them to more concrete functional, non-functional and security requirements. It also illustrates the use cases from a user's perspective.
- **Chapter 4** goes through the design and implementation of the solution, giving details about the components that are involved, as well as the decisions that were made.
- **Chapter 5** describes the testing strategies that were employed to ensure the correct behavior of the system.
- **Chapter 6** evaluates the system as a whole, going through requirements to see if they have been met. It also evaluates the cost and performance overheads.
- **Chapter 7** describes the future work that is needed, and concludes.

2 Context

2.1 Blockchain

By definition, blockchain is a data structure that is resistant to mutations, however the term blockchain has been used to refer to various different concepts and protocols in the industry. Usually it's the amalgamation of these concepts working together that give the advantages that is needed for the solution. So, it was important to explore these concepts in detail. In this section, an overview of how Bitcoin works is provided, followed by an analysis on Ethereum and smart contracts, and finally, a look into other protocols such as Quorum is taken.

2.1.1 Bitcoin

The popularity of blockchain and the applications of it, emerged from the success of cryptocurrencies, and the first cryptocurrency, and still the largest is Bitcoin with a market capitalisation of $100bn [3]. Developed under the pseudonymous name Satoshi Nakamoto [4], Bitcoin laid the foundations for decentralised currency system with a high Byzantine Fault Tolerance (BFT). To understand how Bitcoin achieves this, the system has been broken down into three layers: (i) The transaction layer; (ii) The consensus layer; (iii) and finally the network layer. A brief overview of these layers is provided in the section below. Please note that the descriptions and data structures have been truncated and simplified for brevity. More specific details about the system can be found in Bitcoin documentation [5].

2.1.1.1 The Transaction Layer The transaction layer deals with how the currency is represented, and how it flows from one owner to another. Bitcoin uses a so-called UTXO model. In this model, instead of tracking every account and how many bitcoins they own, the system tracks every coin that has ever been minted in the system, and tracks the ownership of those coins. Every change of ownership is represented in a object called a transaction. A transaction in general contains of following:

- List of Inputs, every input then contains:
  - An output that is not used yet, also know as a Unspent Transaction Output (UTXO)
  - A signature by the user who owns the UTXO (represented as an executable script known as $scriptSig$)
- List of Outputs, every output then contains:
  - The id of the user who will be the owner of the bitcoins (represented as an executable script known as $scriptPubKey$)
  - The value i.e. how many bitcoins being sent

To verify if a transaction is valid, a node needs to perform three checks: (i) The UTXOs referred in the inputs have not already been used (to prevent double spending); (ii) The $scriptSig$ and $scriptPubKey$ executes without error; (iii) The values from inputs $\geq$ the output values.

The challenge of storing this information in a way that is resistant to changes was considered by Nakamoto, and the solution was to use hash pointers. The idea was to collate a group of transactions into a larger object known as a block. Then the hash of the block will become its address and id, which the next block will refer to (as shown in Figure 2.1). This creates a chain of block objects also known as a blockchain. This chain is resistant to mutations, as any change to

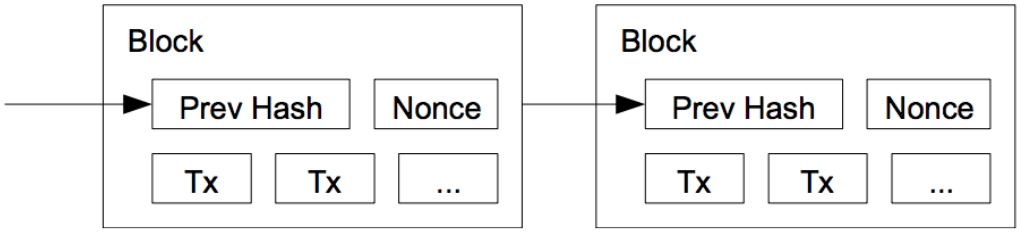


Figure 2.1: Block object consisting of transactions and pointer to the previous block

the contents of the block, will change the hash, and therefore the next block's reference becomes invalid. To achieve a valid blockchain, the adversary will need to change all the blocks from the block that was changed to the most recent one, which is not a trivial task. However, as one can see, having one entity control this chain would be dangerous, as they will be able to control the rules of a valid transaction, and also the order of them. Therefore, the system needed to be decentralised.

2.1.1.2 The Consensus Layer Bitcoin allows anyone to become a node in the system, allowing them to download a copy of all the blocks that have been generated, process validity of new transactions and propose new blocks. However, this creates an issue of how the nodes reach a consensus especially when the nodes cannot trust each other. The consensus layer deals with this problem. The consensus protocol presented in Bitcoin paper, is known as the Nakamoto consensus. The idea is that one of the nodes will be selected to propose the next block, however the selection itself will not be random, but weighted in favour of the nodes that have higher computational capability. This is achieved by using a 'puzzle-friendly' hash. Every node tries to compute the next block who's validity is given by:

$$Hash(previousBlock||transactions||nonce) < targetValue$$

Since the hash chosen by Bitcoin is 'puzzle-friendly', the only feasible way to calculate a valid hash is to iterate through every nonce. This process of finding the correct nonce is known as mining. The nodes that have higher computational capability, have more chance of computing the correct nonce, and thus proposing the next block. The target value is adjusted by all nodes to have a constant time of 10 minutes between each block.

This protocol achieves strong Byzantine Fault Tolerance. Nodes cannot broadcast invalid blocks or transactions, as it will be rejected by other honest nodes and the blocks that have been accepted are difficult to change (due to hash pointers). However, as nodes accept the longest chain of valid blocks, a node could potentially calculate a chain of valid blocks that is longer than the one currently accepted by all other nodes. This causes issues with the immutability of the blockchain, and its implications in the real world. However, computing a competing chain of valid blocks is difficult as it requires at least 50% of the computational power of all nodes, and becomes exponentially more difficult as deeper the fork gets. In the real world, most entities will wait six blocks (60 minutes), before considering a block immutable, as the probability of a competing blockchain overtaking approaches zero[4][6].

2.1.1.3 The Network Layer The network layer deals with how the transactions and blocks relayed between nodes. Bitcoin uses a peer to peer network to broadcast transactions and blocks. However, as with peer to peer networks, the usual problems such as discovery exists. The bitcoin nodes are usually hard coded with other nodes known as DNS seed that act similar to traditional DNS server. The node upon joining the network, downloads a list of peers from the DNS seed [5]. The peer then connects to one or more nodes and announces its presence, this announcement then propagates throughout the network. The peers use the Gossip protocol [7] to announce transactions and blocks. Although, peer to peer protocols are inherently decentralised, concentrations of power in the Bitcoin system does exist. For example, the hard coded DNS seeds may provide a list of dishonest nodes, or if one of the peers knows the topology of the network (such as an ISP), they may be able to perform eclipse attacks [8].

2.1.2 Ethereum and dApps

Much like Bitcoin, Ethereum [9] is another cryptocurrency that has gained significant popularity in recent years [3]. The intent behind Ethereum was to create a decentralised platform for apps. It maintains similar design decisions as Bitcoin, however there are significant differences. Unlike Bitcoin's UTXO model, Ethereum utilizes an account based model. In an account based model, every account is tracked, and they are updated by transactions which transfer Ether (Ethereum currency) from one account to another. Although, it's more intuitive to have accounts, it's much harder to prevent double spends. An account in ethereum contains of the following:

- Nonce - Used for verification.
- Ether balance - How much ether the account has.
- Contract code (if any) - Used by smart contracts (discussed below).
- Account storage.

One of the limitations in bitcoin is that the script with which $scriptSig$ and $scriptPubKey$ are written (see section 2.1.1.1), is very limited and not Turing complete. Although this simplifies the protocol and improves security [5], it also discourages developers from writing complex code. Ethereum improves on this by providing a Turing complete Ethereum Virtual Machine (EVM). This allows developers write complex scripts in higher level language such as Solidity [10].

2.1.2.1 Smart Contracts Ethereum allows two types of accounts to exist: (i) accounts which are externally owned (for e.g. by humans); (ii) contract accounts that are essentially autonomous agents controlled by their contract code. These contract accounts are also known as 'smart contracts'. Ethereum transactions not only allows ether to be transferred between accounts, but also allows accounts to call functions in the smart contract. When an account calls a smart contract code, the account needs to compensate the node that computes the code. A transaction in Ethereum consists of the following:

- Recipient - The id of the recipient.
- Signature - The signature of the caller.
- Amount - The amount of ether to be transferred.
- Data - Optional field for smart contract.
- GASPRICE - Fee the caller is willing to spend for computation.

- STARTGAS - Maximum fee the caller is willing to spend.

For creating a smart contract, the fields of the transaction has the following semantics:

- Recipient - The address where the contract will reside.
- Signature - The contract author.
- Amount - The amount of ether to be sent to the node that will compute.
- Data - The contract code.

For calling a smart contract, the fields of the transaction has the following semantics:

- Recipient - The address of the smart contract to be called.
- Signature - The caller.
- Amount - The amount of ether to be transferred.
- Data - Data need to execute the contract code such as function inputs.

This allows Ethereum to essentially act as a distributed computing platform, allowing developers to write apps with distributed architecture also known as dApps (decentralised apps).

2.1.3 Quorum

Quorum is a permissioned/private blockchain solution developed by J.P. Morgan, that is based on Ethereum but with a number of modifications:

- The network layer has been modified so it is no longer follows the P2P protocols, but only connects to permissioned nodes.
- Instead of using the proof-of-work (Nakamoto consensus), it gives a choice of two protocols: (i) Raft - which is a consensus protocol based on Paxos, and doesn't have any Byzantine Fault Tolerance (BFT); (ii) Istanbul BFT which does provide BFT.
- GasPrice is no longer used.
- Private Transactions and contracts, that only allow a selective number of users to view the transaction and account.
- Higher performance than Ethereum.

Quorum [11] was designed with the financial services in mind, providing a distributed ledger for entities (such as banks) that is private to them. Although this system could be explored further to see if it can be used in NHS and in MDT meetings, one of the biggest disadvantages of Quorum is that all users will now have to trust the owner of the private blockchain (in our case NHS) to maintain the immutability of the. which may not be ideal.

2.2 Identity

The problem requires the investigation of an identity system known as DeeID (formerly Omnee) which was developed by Sirvan Almasi [12]. The reason for this requirement becomes clear, when considering that there is a need to guarantee the authenticity of the events that happen in a meeting. As such, a simple database maintained by GOSH is not sufficient, as a participant of a meeting can claim that the system deliberately tampered with identities, and accused them falsely.

2.2.1 DeeID

DeeID is a identification system that uses Ethereum smart contracts and Fierge-Fiat-Shamir identification scheme [13] to verify user's identity. The idea is that there is a trusted entity (such as the government), who is responsible for verifying the user's identity in real life, and creating keys to identify the user in the future. This trusted entity is trusted by both the user as well as the other entities that want to verify the user's identity (such as GOSH). The identity itself is stored in a smart contract for permanence.

There are two parts in the DeeID identity system: (i) Registration; (ii) Verification. These parts are briefly described in this section. For full explanation and the details on the cryptographic primitives behind this identification scheme, please refer to the DeeID paper [12].

2.2.1.1 Registration

- To register, the user deploys a DeeID smart contract using it's Ethereum account. This smart contract is responsible for storing public keys that the user can later use for other purposes. The user then generates a string I, that uniquely represents them. The string I can include personal information such as name, email, bio-metric data and most importantly the DeeID smart contract address.
- The trusted entity, who knows the factorisation of some modulus $n = pq$, makes n public along with some pseudo-random function f which maps$I$ to a range $[0, n)$. The trusted entity verifies I (in real life, using Passport etc.), and generates a number of secret keys $s_1...s_k$ which are given back to the user.
- The user then stores these secret keys in their DeeID wallet/app.

2.2.1.2 Verification

- The user wants to now prove their identity with another entity (lets consider GOSH as an example). The user provides GOSH with the string I (which contains personal information about the user).
- GOSH computes the set of public keys $v_1...v_k$ using the publicly available function f and the value n. GOSH then challenges the user by selecting a random r such that $r < n$ and sending $x = (\pm r^2) \mod n$.
- The user then chooses a random vector $e_1...e_k$, who's values are either 0 or 1.
- The user then computes and sends:

$$y = r \prod_{e_j=1} s_j \mod n \tag{1}$$

- The verifier (GOSH) then verifies x, by computing:

$$x = y^2 \prod_{e_j=1} v_j \mod n \tag{2}$$

If the verification is successful, the verifier (GOSH) knows that the user was verified by the trusted entity (in our example, the government), and the DeeID contract (from string I), contains the keys added by the user.

2.3 Problem Research

2.3.1 MDT meeting

As the problem specialises on Multidisciplinary Team (MDT) meetings, it was necessary to research how these meetings are held. I was invited to attend a MDT meeting on 10^{th} of January for the purpose of fact-finding.

2.3.1.1 Layout and Structure The meetings are held in a conference room, with approximately 50 participants, with one of the participants acting as the host of the meeting (as illustrated in Figure 2.2). The host is responsible for going through the agenda of the meeting, as well as operating the software that shows the patient details. The agenda usually includes around 5-20 patients, each of whom are discussed individually. The MDT meetings can last up to 3 hours, and participants join and leave at any time during meeting.

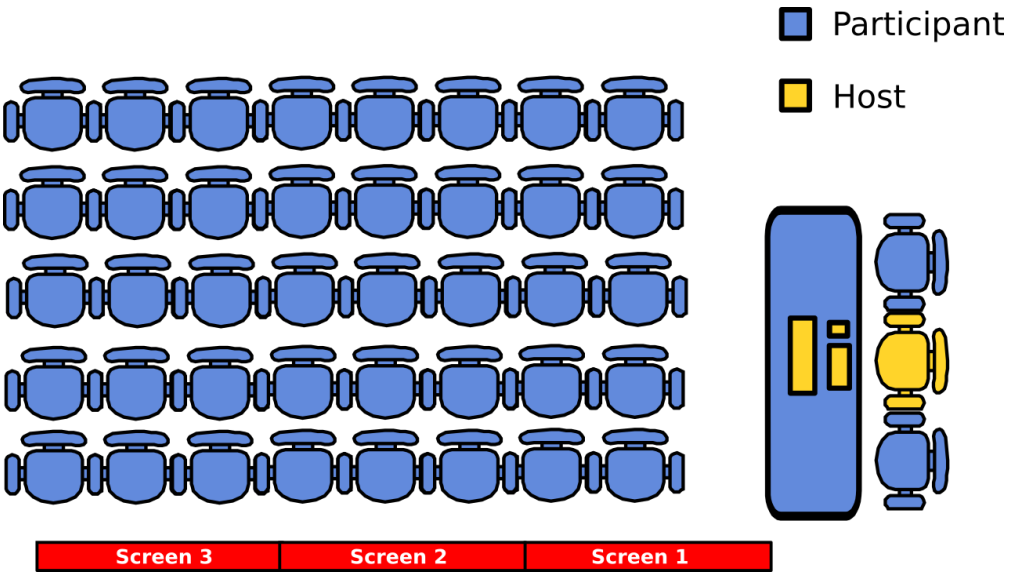


Figure 2.2: Illustration of the layout of MDT meetings in GOSH

2.3.1.2 Software and Information The software that the host operates is known as InfoFlex, developed by Chameleon Information Management Services Ltd. [14]. It displays two types of information across three screens; (i) one type of information is images (for example scans from medical tests) which is shown across two of the screens; (ii) and the second type is the details (text) of the patient that is being discussed at the time, which is shown in one of the screen. The latter type can be further split into two categories:

- Patient's personal details which include:
  - Name
  - Date of Birth
  - Patient Number

  - Hospital Number

- Patient's MDT meeting discussion details that changes per meeting, which include:
  - Group
  - MDT Outcome
  - Investigation
  - Questions for MDT
  - Surgery
  - Main Opinion
  - LC

2.3.1.3 Voting After each patient has been discussed, there would usually be a vote to determine the outcome (for example, whether the patient needs further tests). The way voting was done in the meeting is either by raising hands or yelling 'yes'. No formal procedure of voting was observed, and the privacy of the votes did not seem to be an issue. Additionally, the majority of the participants would usually abstain from voting.

2.4 Development Research

During the development of the solution, a number of libraries, frameworks and development tools were utilised. A few of them which were significant to the development have been described in this section.

2.4.1 Truffle Suite

Truffle Suite is a collection of open-source tools that provide a development environment and a testing framework for writing Ethereum smart contracts [15]. It allows developers to manage the life cycle of smart contracts and automatically deploy updates to existing smart contracts (migration). It also allows the developer to interact with the smart contract through a simple command line interface. One of the tools available in Truffle Suite is Ganache [16], which creates a private Ethereum blockchain network for the developer to test and deploy smart contracts on. Ganache also has a GUI which helps in visualising the blocks and the transactions within them.

2.4.2 Web3.js and Metamask

Web3.js is a collection of JavaScript libraries that allows a website to interact with Ethereum blockchain node, which might be hosted locally (such as geth or Ganache) or remotely (such as Infura [17]). It also provides the cryptography libraries for signing, verifying and creating ethereum addresses (public, private key pairs). There is also a similar library for python known as Web3.py which provides similar functionality and interface. Metamask is a chrome extension that connects to an ethereum node, which may be hosted locally. As a website cannot access a locally hosted client, Metamask provides the bridge, by injecting the Web3.js library when the website loads, thus allowing websites to interact with locally hosted Ethereum nodes.

2.4.3 Angular Framework

Angular is an open-source web app framework, developed by Google that enables developers to build web apps [18]. Typescript is the language used in Angular instead of JavaScript. There are two reasons for using Angular instead of other frameworks such as React, Vue.js or server side rendering templates. The reasons are: (i) Familiarity with Typescript and experience with the older version of the Angular, which sped up the development process; (ii) Angular is a single page application development framework, this allows for better separation of the front-end from the back-end, so in the future, the GUI can be replaced without the need to tweak the back-end.

2.4.4 Flask

Flask is a library for Python that enables developers to create web servers. The reason for using this library instead of Django, Bottle or other libraries, is because the code base for the DeeID system (see Section 2.2.1) is written in Python/Flask, so it helped speed up the integration process, reducing interoperability issues. Additionally, the simplicity of Flask's interface also enabled rapid prototyping.

3 Requirement Analysis

As described in Section 1, the initial meeting with the MDT staff members revealed two important requirements; (i) the solution should capture attendance, voting, and comments from the MDT meeting; (ii) the solution should also guarantee integrity and authenticity of the data generated in the meeting. To better understand how this can be achieved, these abstract requirements have been broken down into more specific functional, non-functional and security requirements that will be considered during design and implementation of the solution. Since this is such a large project with numerous facets, it was necessary to prioritise each requirement, this has been done using the MoSCoW methodology.

3.1 Definitions

Before drafting the requirements, some terms need to be defined which will be used in further sections:

- **Participant:** A participant is any user who is authorized to participate in a meeting.
- **Host:** A host is a participant with higher privileges, able to perform actions that a ordinary participant cannot.
- **DeeID:** An Ethereum address associated with the user's identity (see Section 2.2.1).
- **DeeID system:** Refers to the DeeID identification scheme and the system as a whole.

3.2 Functional Requirements

ID	Requirement	Priority	Complexity
Meeting management			
F1.1	The system must allow users to create meetings.	M	Low
F1.2	The system must allow users to cancel meetings.	M	Low
F1.3	The system must allow users to edit meetings.	M	Low
F1.4	The system must allow users to view meetings.	M	Low
Active meeting			
F2.1	The system shall allow the host to start a meeting.	M	Low
F2.2	The system shall allow the host to end a meeting.	M	Low
F2.3	The system shall allow the participants to join meetings.	M	Low

F2.4	The system shall allow the participants to leave meetings.	M	Low
F2.5	The system shall allow the participants to record their voice.	W	High
F2.6	The system shall allow participants to leave comments in the meeting.	M	Medium
F2.7	The system shall allow participants to reply to comments.	C	Medium
F2.8	The system shall allow participants to record their disagreement with any event in the meeting.	S	Low
F2.9	The system shall capture the host's screen for the duration of the meeting.	W	High
F2.10	The system shall allow the host to add patient details that are meeting specific.	C	Medium
F2.11	The system shall allow the host to create a poll.	M	Medium
F2.12	The system shall allow participants to vote in polls.	M	Medium
F2.13	The system shall allow participants to see the poll results.	S	Medium
Stakeholder requirements			
F3.1	The physical attendance of participant in a meeting must be recorded.	M	High
F3.2	The system shall record what time the participant joined the meeting.	S	Medium
F3.3	Participants must not know the results of votes until the poll has ended.	C	High
F3.4	The system shall allow retrospective auditing of events that happened in a meeting.	W	Medium
F3.5	Events must be recorded in the order that they happened.	C	Medium
F3.6	The system shall allow users to login using DeeID.	C	Medium

3.3 Security, Integrity and Transparency Requirements

ID	Requirement	Priority	Complexity
S1.1	The system must protect the privacy of voters in a meeting poll.	C	High
S1.2	The system must be able to detect any tampering or corrupting of data.	M	High
S1.3	The system must not store any patient or meeting data on the public domain.	M	Medium
S1.4	The system shall allow participants to see what meeting data was stored.	C	Medium
S1.5	The system shall be open sourced.	W	High

3.4 Non functional requirements

ID	Requirement	Priority	Complexity
N1.1	The system shall be user-friendly.	C	Medium
N1.2	The system shall be secure.	M	High
N1.3	The system shall be performant.	W	High
N1.4	The system shall be scalable.	W	High

3.5 Stakeholders

- **Great Ormond Street Hospital:** Since the idea of the project came from GOSH, they will be the first to consider the deployment of the system, as such they need to be consulted frequently on all design decisions.
- **Staff at GOSH:** If the system is ever realised, the staff members of GOSH will be the users of the system. Their opinion of the system is important, and must be considered during the design process.

- **Patients:** Since the system will handle patient data, the patients inevitable become stakeholders, and thus they should have a say in the design. However, since this is a proof of concept and an investigation, the project assumes that the patients hold no objections.
- **Author of DeeID system:** This project is one of the first to realise the DeeID identity system, as such there is a great deal of interest from the author, to see the benefits and the flaws of the identity system in a concrete implementation.

3.6 Use Cases

Figures 3.1 shows the use cases from the host and participant's perspective in an on-going meeting. Please refer to Appendix C for the use case documents.

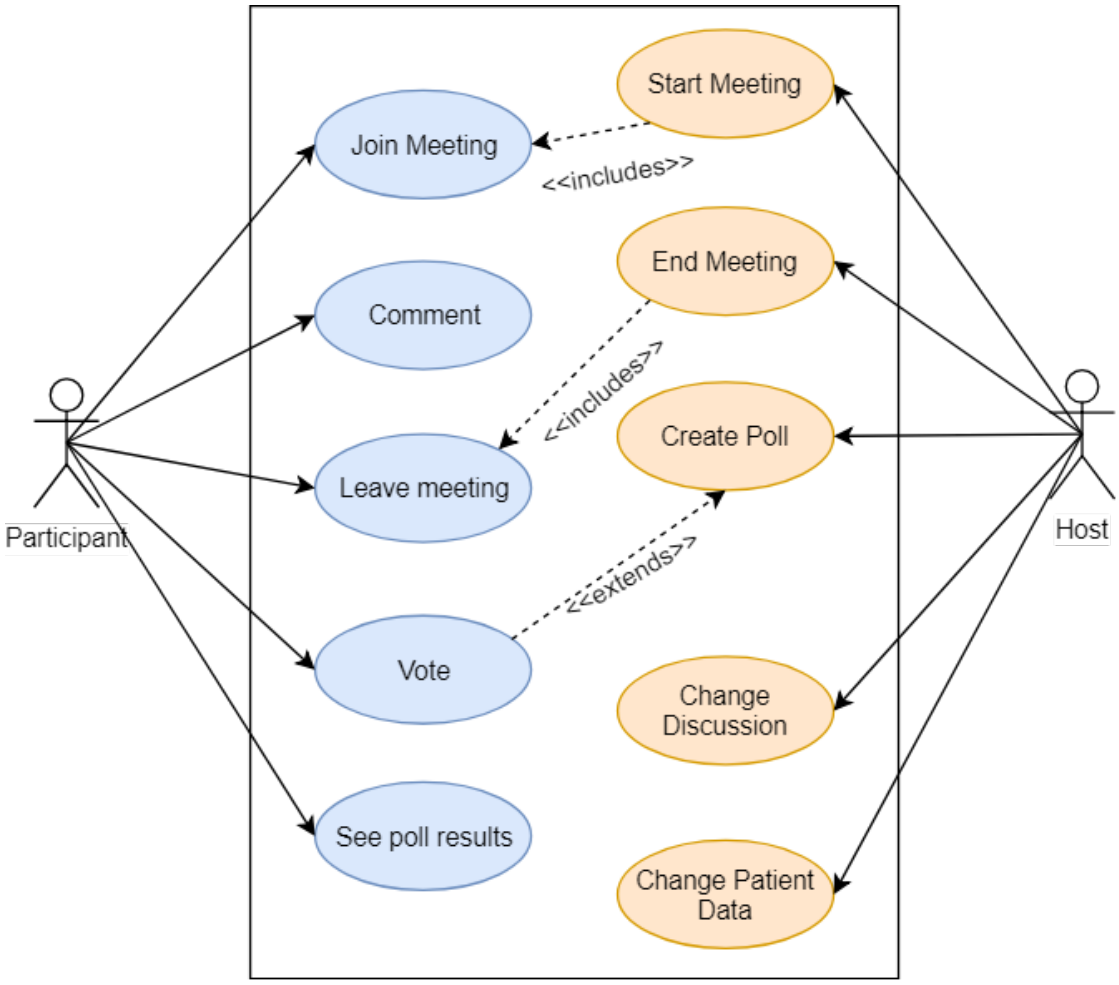


Figure 3.1: Use case diagram for host and participant of an ongoing meeting

4 Design and Implementation

4.1 Overview

During the literature review, it was realised that there wasn't much related work which met the requirements that were set out initially, so existing systems could not be used to construct a solution. Therefore, a bespoke system was designed which is dubbed as c-meet (short for crypto meeting), that aims to deliver on the requirements. The system comprises of two servers, a client (UI) and a protocol dubbed as the "Events protocol". This section gives an overview of all the components that are involved, and briefly mentions their purpose. A diagram (Figure 4.1) is also provided to show the interaction between components. Deeper dive is taken into these components in later sections.

4.1.1 C-meet components

- **Meeting Server**: This is the server responsible for facilitating the communications between participants in an active meeting. The server is also responsible for interacting with smart contract on Ethereum network. This component was developed in python using flask, socket.io and web3.
- **Rest API Server**: This is the server responsible for CRUD operations on meetings and the user sign in functionality. This component was developed in python using Flask.
- **Meeting Database**: This component is responsible for storing meeting metadata (such as title, participants etc.), patient specific meeting data (see Section 2.3.1), as well as all the events that happen in a meeting (such as votes, comments etc.). MongoDB is used in the implementation of the system, however, any DBMS would be appropriate assuming it is ACID [19] compliant.

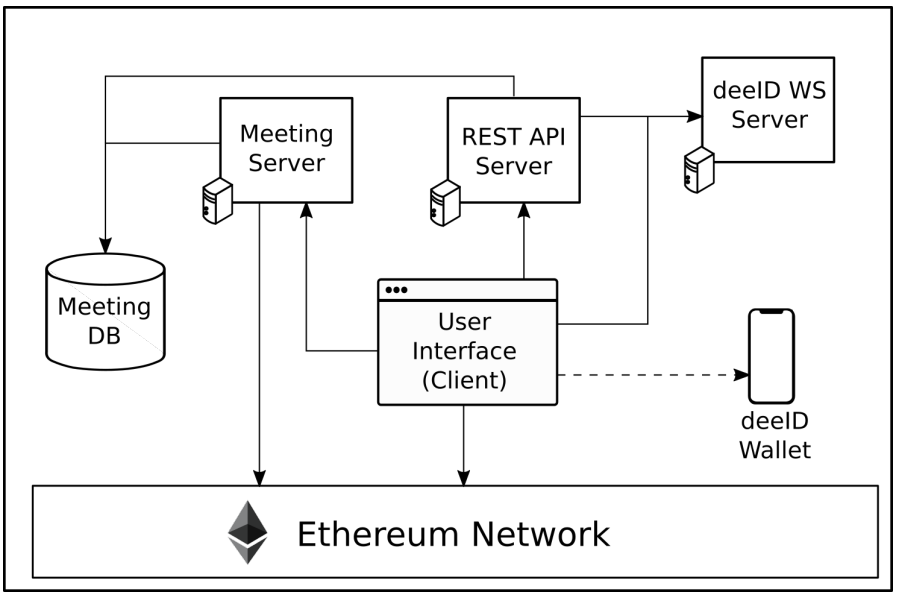


Figure 4.1: Overview of all components in the system and who they interact with.

- **User Interface**: The user interface is how all users will interact with the system. The user interface connects with the servers mentioned previously, as well as with the Ethereum network for verification. There could be many interpretations of how an UI for the system can be visualised, however a one working implementation has been provided. Angular 4 was used to develop a web app that both the host and participants can use.

4.1.2 DeeID components

The DeeID system is used for identifying staff members. Although this a separate development from this project, there was close collaboration with the author of DeeID to integrate the c-meet system with the DeeID system. As such, components from the DeeID system have been briefly described below:

- **DeeID App (Wallet)**: This is an app that all users of the system will need to possess in order to login. This app contains the wallet (private keys) needed to verify their identity.
- **DeeID Web socket Server**: A server that acts as a bridge between the c-meet system and the user's DeeID wallet. This is explained further in Section 4.2.

4.1.3 Ethereum Node

The system requires at least one full node that is connected to the main Ethereum network. However, since this is a prototype, a private network provided by Ganache has been used [16].

4.2 Login with DeeID

As mentioned previously the identification is done through DeeID, as such the login is developed using the DeeID system. During the development, it is assumed that users have already gone through the registration system provided by the DeeID system (as described in Section 2.2.1), and also the user's DeeID contract has been verified and there exists a database with the details of the users mapped to their DeeID. The login system then works as follows (see Figure 4.2):

1. The user navigates to login page. The web app contacts the DeeID WS server using a websocket connection. The web app keeps the connection active.
2. The DeeID WS server responds with a unique ID (uID).
3. The web app then generates a `LoginSig` JSON object, and displays it as a QR Code to the user (Figure 4.4), The `LoginSig` object contains the following:
   - Type - which is set to "loginSig" as required by the DeeID system.
   - uID - that is received from the DeeID WS server.
   - WsURL - the URL of the DeeID WS server.
   - Data - a field that can contain arbitrary data. This is populated with session keys, which is used later in the Events protocol (Section 4.6).
4. The user scans the QR code with the DeeID app, which then verifies all the data with the user. The app then signs the `LoginSig` object and sends it to the DeeID WS server.
5. The DeeID WS server verifies the signed object, specifically the uID (to make sure that it originated from the server), and creates a session for which the signed object is valid. The server then passes the signed object back to the web app.

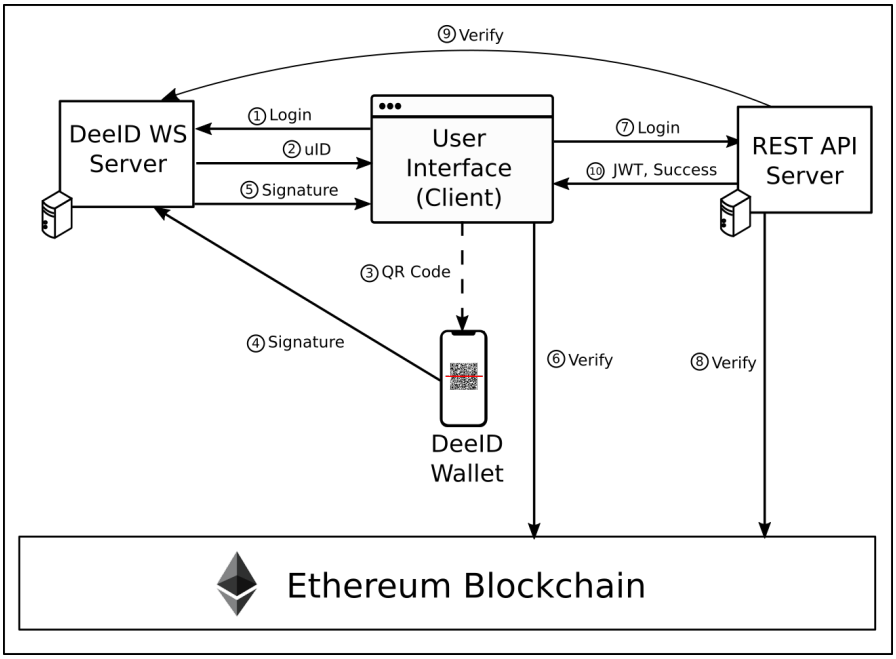


Figure 4.2: The DeeID Login procedure

6. The web app verifies that the key with which the object was signed is indeed from the user's DeeID wallet. It does this by checking the smart contract associated with the DeeID on the Ethereum blockchain.

7. The web app then connects with the REST API server using the "/login" API endpoint, presenting it with the signed `LoginSig` object.

8. The REST API server then verifies the key with the DeeID contract (similar to the web app).

9. The server then verifies if the uID is valid with the DeeID WS server.

10. If the verifications are successful, the REST API server then issues a JSON Web Token (JWT) [20], asserting that the user is authenticated. The web app can then present the JWT for all future REST API requests until the JWT token expires.

4.3 Web App

In this section an overview of how the web app is structured is provided. The web app is how the user manages their meetings, as well as participates in them. The major components of the web app are described below. A diagram to illustrate the UI flow (Figure 4.3) has also been provided. Also, please note that these are only the major components, there are many smaller components that have not been listed.

- **LoginPage (Figure 4.4)**: This component is responsible for displaying the QR code, needed for logging in the user (as described in the previous section).

- **MeetingPage (Figure 4.5)**: This is the component that will interact with user the most. This displays and handles all interactions related to an ongoing meeting. It allows creating polls, changing patient data, starting a formal discussion and ending the meeting for the host, as well as commenting, replying and voting for all participants.
- **MeetingListPage (Figure 4.6a)**: This component is responsible for fetching the meetings, sorting them by the date, and categorising them as either "previous" or "upcoming".
- **MeetingCreatePage (Figure 4.6b)**: This component displays the details of the meeting, and also allowing the user to create, edit and delete the meeting.

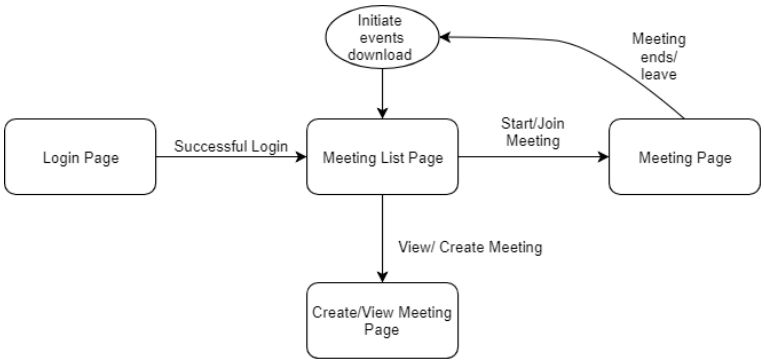


Figure 4.3: UI Flow Diagram

Figure 4.4: Login Page

Services are singletons, that run in a separate thread, they can be accessed by any component as long as they have been injected with them (using the component's constructor [18]). There are four services in the web app, they are as follows:

- **AuthService:** This service handles all authentication functionality that is requested by other components or services. This includes generating and verifying signatures, generating keys, storing JWT as well as connecting with smart contracts.
- **EventStorageService:** This is another service that assists the MeetingPage, in handling the stream of events that arrive from the Meeting server.

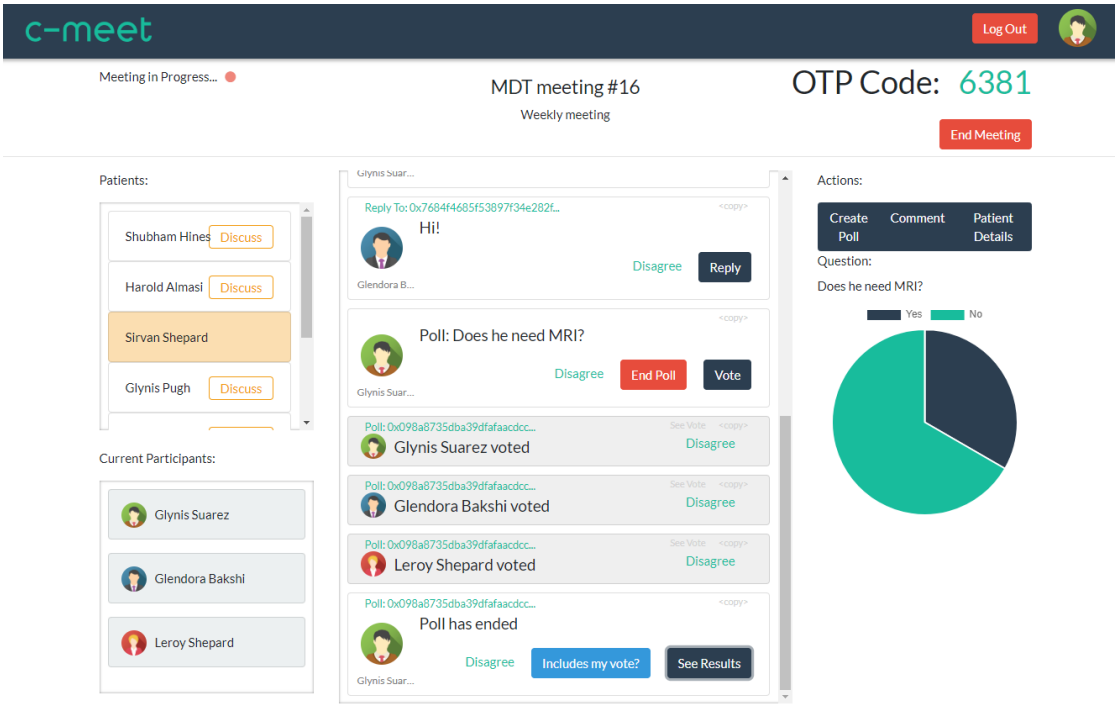
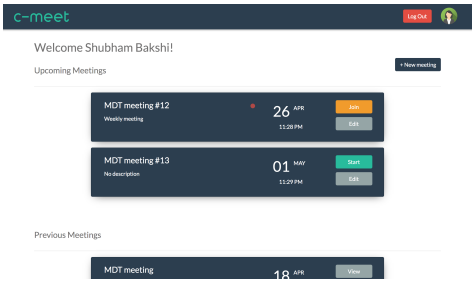
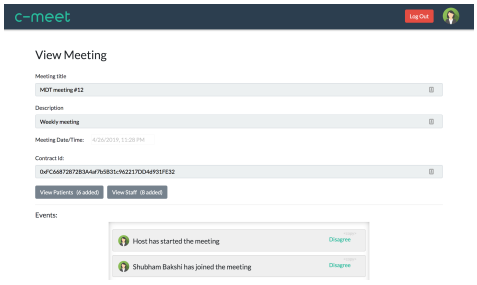


Figure 4.5: The Meeting Page

- **RESTService:** This service handles all REST API requests on behalf of all the components and services.
- **MeetngService:** This service opens a web socket connection with the Meeting server, and listens to the socket for new events in an active meeting, it also notifies other components by providing observer objects. It also handles sending events to the Meeting server.



(a) Meeting List Page



(b) Meeting View/Create Page

Figure 4.6: Web app screens

4.3.1 MVC Architecture

In the implementation of the web app, a strict Model-View-Controller architecture is followed. Since this is a proof of concept and the UI was not the focus of the project, having the functionality of the web app separated from the GUI, helped to quickly iterate and check if new ideas worked in practice. Once the functionality was deemed to be correct, efforts were diverted to improve the design and UI of the web app. Angular-cli [21] was used to generate the project, this automatically organised the project according to MVC architecture (Figure 4.7). So for every component there is:

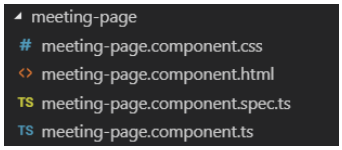


Figure 4.7: Folder structure for MeetingPage component

- *.html, *.css - this can be thought of as the view.
- *.ts, *.spec.ts - this is the Typescript file containing the logic of the code and hence can be thought of as the controller.
- model.ts - this is where all the class definitions of models exist.

4.3.2 RxJS

There is a lot of asynchronous data operations in the web app, as such it becomes quite tricky to implement correct behaviour. RxJS [22] helps to solve this issue. It implements the reactive programming paradigm, where instead of using traditional callback functions, it treats everything as a stream of events. An "observer" then subscribes to the stream of events, and performs operations on them using an iterator. This paradigm allows for more complex behaviour such as filtering, joining two streams, mapping and more, without needing to implement complicated design patterns. However, there were certain scenarios in the project where using reactive programming was unintuitive, in such cases it was more natural to use traditional callbacks and the observer pattern.

4.3.3 Metamask

As mentioned in the login procedure, the web app needs to be able to connect with the Ethereum blockchain to verify identity. However, since a web page cannot connect to Ethereum network (due to security concerns), the web app relies on Metamask extension to inject the web3.js library. This library provides a bridge between the web page and the Ethereum network. As such there are two requirements for web app:

- The web app must run on a browser that can install the Metamask plugin. As of April 2019, these are: Google Chrome, Mozilla Firefox and Brave Browser [23].
- The Metamask plugin must be configured to connect to a Ethereum node, which might be running on the user's device, or they may trust a third party such as Infura [17] to run a node for them.

4.4 Servers

There are two servers in the c-meet system. These servers are implemented using Flask, which is a library for Python that utilises the Werkzeug WSGI toolkit to interface with a web server. Flask has it's own development server that is used to serve the web pages and APIs in the project,

however, in a full deployment it is expected that a dedicated HTTP server such as Apache or nginx will be used. It is also assumed that all communications happen over TLS.

4.4.1 Meeting Server (Websocket Server)

This server's main purpose is to facilitate an ongoing meeting. All communication with the server happens through the websocket protocol [24]. When a user starts a meeting, it creates a websocket "room" for that meeting, and broadcasts any event that it receives from the any of the participant registered in the room. It validates all events it receives against the rules in Events protocol (described in Section 4.6), and stores them in the database. It is also responsible for creating smart contracts for meetings on the Ethereum network.

4.4.2 REST API Server

This is a simple HTTP server, that provides REST API endpoints for the web app to access the staff, meeting and patient database. It is also responsible for logging in the user using the DeeID system and JSON Web Tokens.

This server provides a REST API for the following operations:

Endpoint	Method	Description	Authentication
/login	GET	Performs authentication of the user. Returns a JWT if authentication successful	Not required
Meetings			
/meetings	GET	Gets all the meetings for the staff member	JWT required in request header to identify staff
/meeting	POST	Creates a new meeting	JWT required in request header to identify staff
/meeting/{id}	[GET, PUT, DELETE]	Read, Update and Delete operation on a specific meeting	JWT required in request header to identify staff
Patients			
/patients	GET	Gets list of all patients	JWT required for authorization

/patient/{id}	GET	Gets the patient id	JWT required for authorization
Staff			
/staff	GET	Gets list of all staff	JWT required for authorization
/staff/{id}	GET	Gets the staff id	JWT required for authorization
Patient Meeting Data			
/meeting/{id} /patient/{id}	[GET, PUT]	Read and Update operations for meeting specific patient data	JWT required in request header to identify staff
Events			
/events/{id}	[GET]	Lists all events that happened in a meeting	JWT required in request header to identify staff

Updating or creating events is not allowed through the REST API. This is because all events are created during a meeting, as such the Meeting server handles the creation of events; as for updating, allowing users to update the events would mean they have means to change what they contributed to a meeting, which goes against the requirements. There is also no API for the creation of meeting specific patient data. This is because the fields are automatically created by the REST API server, when the user creates or updates a meeting.

4.5 Attendance

As defined in Requirement F3.1, there is a need to record physical attendance of all users who join a meeting, as well as the time they joined at. A few approaches were considered to achieve this requirement:

- **Approach 1:** Geo-fence meeting rooms, and use GPS to determine the user location
  - **Pros:** Easy to implement. Provides accurate physical location of users.
  - **Cons** May not work in certain cases (such as multi story buildings). Can be easily spoofed by users. Users need to have a GPS enabled device.
- **Approach 2:** Use hardware (such as RFID scanner) that's present in the room, and require participants to have their IDs scanned in order to join.
  - **Pros:** Very accurate. Not easy to spoof.

  - **Cons** Expensive and difficult to implement as it will require new hardware to be installed in every meeting room.

- **Approach 3:** Ask already joined participants if they the new user is physically present in the room.
  - **Pros:** Easy to implement. Reliable.
  - **Cons** Disruptive, as participants will need to approve whenever someone joins the meeting. Prone to collusion, as users can approve even if they are not present.

- **Approach 4:** Show some unique information that is only visible to people physically in the room.
  - **Pros:** Similar to Approach 3
  - **Cons** Prone to collusion, as users may share the unique piece of information.

Approach (4) was chosen for implementation. The reason was because it was reliable and easy to implement. Approach (3) was not chosen over (4), because it was realised that there will be 50+ participants, all of whom will require the approval of someone to join. This may be impractical and inconvenient for the user. Additionally, if a user joins in the middle of a meeting, there is a possibility that they might not be approved at all, as the participants may be focused on the meeting. In the implementation of this approach, a unique four digit one time pin is shown that is only valid for that meeting. This pin is generated automatically by the host when the meeting starts, which is then used by all the other participants to join. One issue that arises from using a short pin is that it can be brute forced; to rectify this, the number of times a user can try a pin in any given time can be limited. Finally, when the user joins, the time is also automatically recorded.

4.6 The Events Protocol

In the requirements, it was determined that there is a need to record the order of events in a meeting such as polls, comments etc. (F3.5). In order to achieve this requirement correctly, the system needs to record two types of order; (i) Temporal order, that is when an event occurred in relation to time; and (ii) Logical order, that is the context an event happened under, for example a reply to a comment, or a vote to a poll. Now, it was also determined in the requirements (S1.2), that there is a need to detect if the data was corrupted (perhaps by an adversary). It was realised early on, that to achieve this, a write once read many (WORM) type of storage system is needed. Cryptocurrencies emulate the WORM property in their blockchain, so a cryptocurrency system which also has the ability to process business logic, such as Ethereum, was an ideal choice. However a single meeting can generate lots of events, all of which needs to be stored in the blockchain, which is both impractical and also expensive (as users pay for computation and storage). Thus, the information that is stored in the blockchain needs to be minimised, without compromising on either of the requirements.

In this section, a protocol is proposed which attempts to minimise the data that is stored on blockchain, without compromising on the authenticity or integrity of the events or their order. In the protocol, an event is defined using the following properties:

- **By:** This is the property that identifies the author of the event. For example, if its a comment, this field will contain the DeeID of the user who commented, if it's an acknowledgement from the server, then this field represents the server's identity using the server's Ethereum account address.

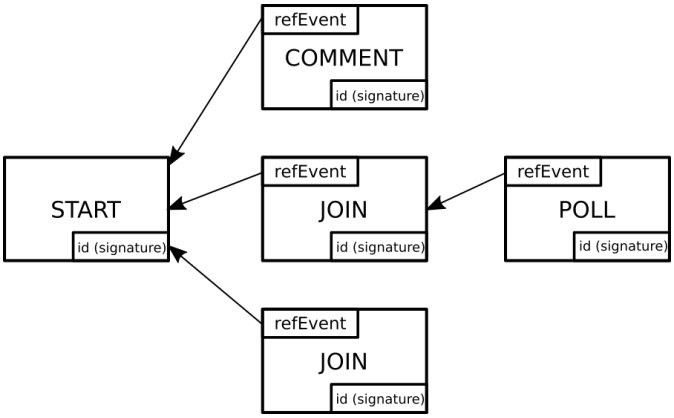


Figure 4.8: Illustration of events referencing previous events, thus forming a tree.

- **Type** This field identifies the type of event, which also influences the content field later. The types include:
  - Action Types: [START, JOIN, LEAVE, POLL, POLL_END, VOTE, COMMENT, REPLY, DISCUSSION, DISAGREEMENT, PATIENT_DATA_CHANGE, END]
  - Acknowledgement Types: [ACK, JOIN_ACK, POLL_END_ACK, END_ACK, ERROR_ACK]
- **Reference Event (refEvent):** This is the field that provides a way of maintaining logical order. Every event apart from (START) must refer to some valid event that has already happened. This creates a tree data structure, with the START event acting as the root node, and all other events being descendants (as shown in Figure 4.8).
- **Timestamp:** This field keeps the temporal order of events. This is a UNIX timestamp.
- **MeetingId:** This field identifies the meeting that the event happened in. An UUID (as defined in RFC 4122 [25]) is used in the implementation.
- **Content:** Represents the contents of the event, can be empty.
- **ID:** This field serves two purposes; (i) it provides the event with an unique ID; (ii) it provides a means of verifying the authenticity of the event. The ID is computed as:

$$ID = sign(privateKey, [by, refEvent, timestamp, meetingId, type, content]) \quad (3)$$

Before the meeting server accepts any event, it does the following check:

1. Event is formed correctly.
2. The timestamp is within tolerance of the server timestamp (10 second window is used).
3. The event has a valid reference event (except for START event).
4. The signature of the event is valid.
5. The author of the event is authorised to post the event.

Once validated, the server then sends an ACK event. The ACK event varies depending the type of the original event. If the event is invalid for any reason, then an ERROR event is sent with the following properties:

- **Error code:** A string that represents the type of error (such as unauthorized, malformed etc)
- **Details:** An object containing further details about the error

This error event (and the original event) is sent privately, and is not broadcast to other users. This is done to prevent malicious users from flooding the websocket stream, and also to prevent other honest users from parsing the event (as it could be a security risk).

4.6.1 Starting and Joining a meeting

To start a meeting, the host needs to send the meeting server an event (as defined before), with type set to "start", and content object containing the following properties:

- **OTP:** This is the one time pin generated by the host, that is used for verifying physical attendance (as detailed in previous section).
- **Session Key:** This is the public key that the host will sign future events with. The reason this is needed is because, in order to sign anything using DeeID, the user needs to use their phone to scan a QR code (as described in the Login section). So it will be inconvenient for the user to have to sign every event they generate in the meeting. So instead the user signs a DeeID object (which contains a new key) with their DeeID private key. The event itself is then signed with the newly generated session key. Once the meeting has ended, the client can throw away the private key associated with the session key.
- **DeeID Object:** This contains the signature of the new session key by the DeeID wallet (previously referred as the `LoginSig` object in the Login section).
- **Meeting:** An object with details of the meeting that is being started.

The server then performs the following check (in addition to the checks mentioned before for an event):

1. The content section is well formed.
2. The DeeID key with which the session key was signed is valid.
3. The DeeID key is in the DeeID smart contract (on Ethereum network).

Once the meeting has been started, other participants can then join the meeting, and a similar interaction will take place, with the addition that the server will now check if the one time pin (OTP) matches the one created by the host. The event will be a JOIN event (which does not contain the Meeting object), and the server will respond with a JOIN_ACK, which contains the START event as well as all the JOIN events the server has seen until that time. This is so the participant that's joining knows what keys other participants are using to sign their messages. This is just as an optimization technique to improve performance, as the user does not need to download all events to find the keys. The whole interaction between the host, participant and the server for starting and joining a meeting is visualised in Figure 4.9.

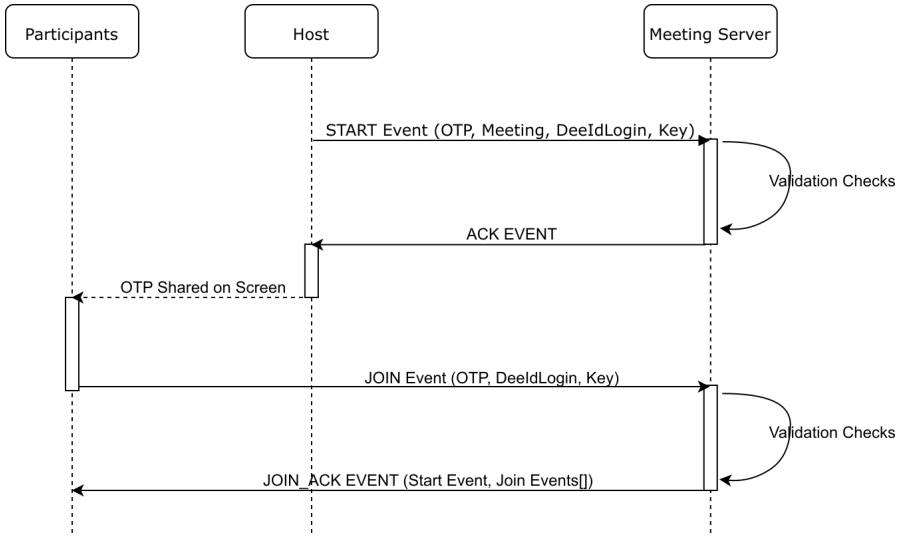


Figure 4.9: Sequence diagram of start and join interaction

4.6.2 Polls and Voting

The host can start a poll by send a POLL event to the server, with content containing the following properties:

- **Patient:** The ID of the patient that the poll concerns. This is an optional field which can be left blank.
- **Question:** The question being asked by the host
- **Options:** A list of strings which are the options that the voter will have.
- **Voting Key:** This is a public key which all participants will use to encrypt their votes.

Once the poll has been created, the participants can start voting by sending a VOTE event, which will contain the encrypted vote. The VOTE event's reference event property must refer to the POLL event. This check is done by the server when accepting the vote. After the host has given enough time, they can end the poll by sending a POLL_END event. The content property of the POLL_END event contains the following:

- Decrypt Key - This is the private key that all the participants will use to decrypt each other's votes.

Although all events are followed by a generic ACK event from the server, the POLL_END event gets a POLL_END_ACK. This event contains the IDs of all the votes that have been casted thus far. The participants then use the votes that are referenced in this POLL_END_ACK event to tally the results.The reason this is needed is because it records the logical order of the votes that were casted before the poll was ended, and thus recording what results were displayed to the user. For example, if a participant were to vote after the poll ends, even though the event itself will be valid, but since it's not referenced by the POLL_END_ACK event, we can conclude that it was not casted before the end of the poll (see Figure 4.10).

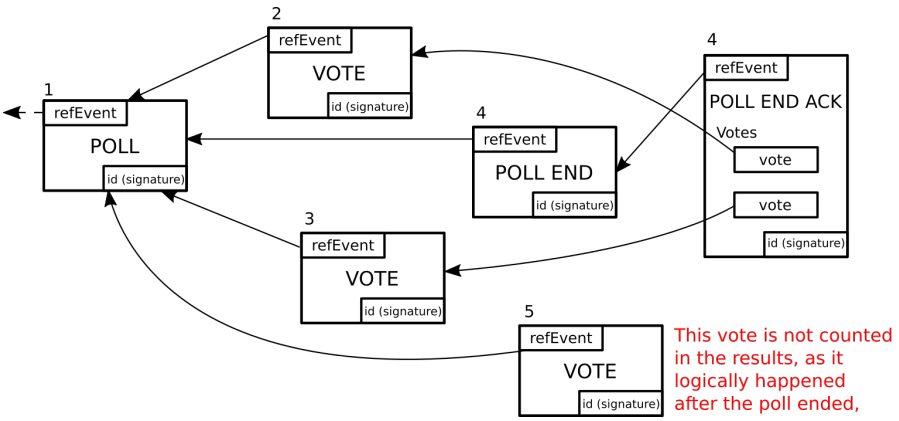


Figure 4.10: Illustration of the event tree preserving the logical order of POLL and VOTE events

However, this opens up a issue where, the user might not know if their vote was counted in the results, however since all of the events are broadcasted to everyone, they can keep a copy of the vote they casted and check its ID against the list of votes included in the POLL_END_ACK. In the web app implementation a "Includes your vote?" button is provided, to help the user 4.11. Another issue is that when the vote is casted, the server cannot check if the vote is valid (i.e. one of the options provided in the poll), and thus cannot reject them. So the number of invalid votes needs to be included when showing the results of the poll.

As per requirement F3.3, there is a need to conceal the vote until the end of the poll, this is the primary reason why there is an encryption and decryption key for votes. Although the solution is not perfect, as the participants need to "trust" the host to not reveal the votes, in my opinion, its an appropriate solution for the use case, as the host has no incentive to reveal.

4.6.3 Comment and Reply

Since the requirement F2.6 and F2.7 state that the user should be able to record comments and reply to them in the meeting, COMMENT and REPLY events have been provided. The COMMENT event has a content with two properties:

- **Comment:** The user's comment.
- **Patient:** The patient the comment refers to (can be empty).

The REPLY event is also similar to the COMMENT event, but the difference is that the REPLY event must reference either a COMMENT event or another REPLY event (as shown in 4.13).

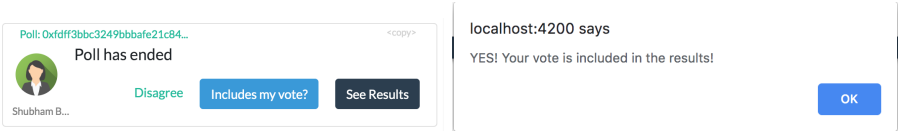


Figure 4.11: UI implementation of vote inclusion verification

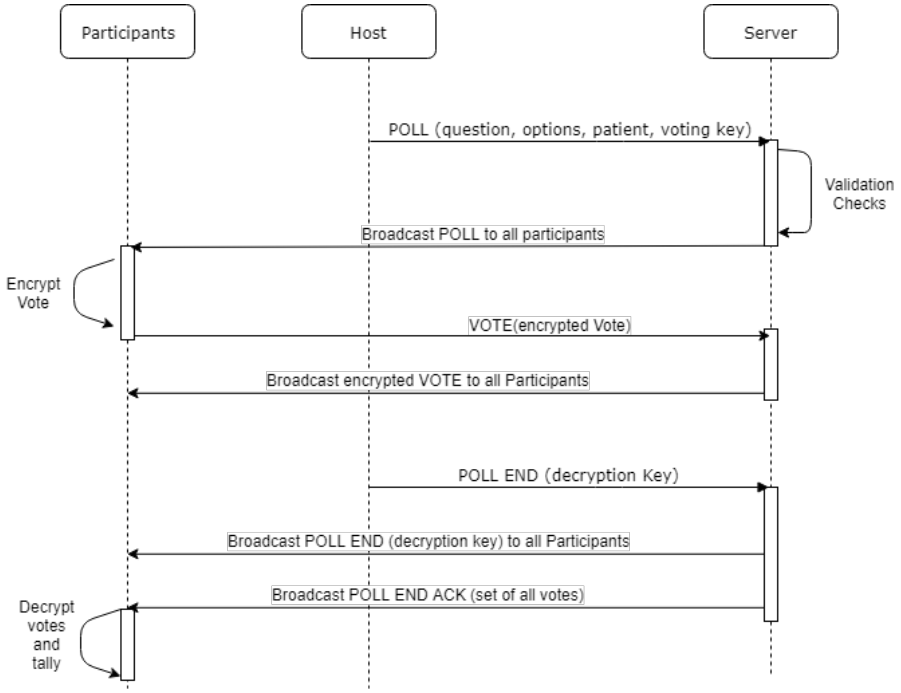


Figure 4.12: Sequence diagram interaction between host, server and participants for a poll

4.6.4 Discussion

The DISCUSSION event content has one property 'patient'. The purpose of this event is to formally state what patient is being discussed at the time. It also helps the web app (UI) know what defaults to set (such as automatically populating the patient property when posting a comment).

4.6.5 Disagreement

The DISAGREEMENT event formally records if a user disagrees with any event that has been broadcasted, as required by requirement F2.8.

4.6.6 Patient Data Change

The PATIENT_DATA_CHANGE event records, the changes made to patient data that is meeting specific. The content of this event has three properties:

1. **Patient:** The patient's who's data is being changed

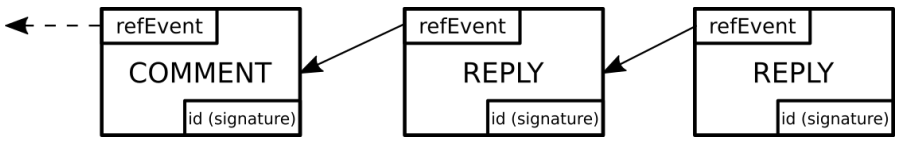


Figure 4.13: Illustration showing the preservation of order of a comment and it's replies.

2. **From:** The old state of patient's data

3. **To:** The new state of patient's data

4.6.7 Ending a meeting

To end a meeting, the host sends an END event. The END event is then acknowledged by the server using the END_ACK event. The END_ACK event contains a list of IDs of all events that have not been referenced by any other event.

4.6.8 Data Integrity and Proof of Inclusion

As mentioned earlier, the main reason for having this protocol is to minimise the amount of data that is stored on blockchain, without compromising data integrity. With this system, only the IDs of the START and the END_ACK event needs to be stored on the blockchain, to be able to determine if the data was corrupted.

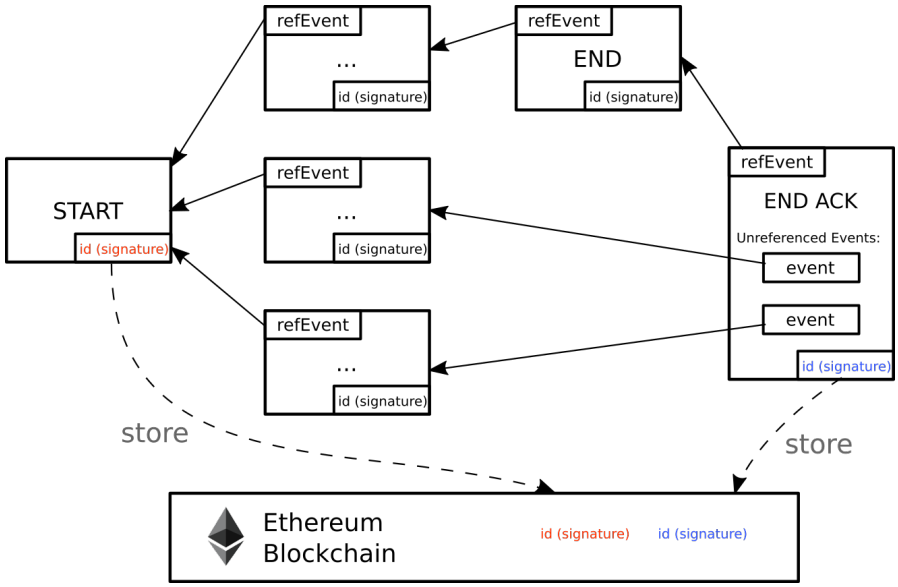


Figure 4.14: Illustration showing the storage of START and END_ACK event IDs into Ethereum.

As can be seen from Figure 4.14, every event (apart from the start) is referenced by another event. So, if one of the events is corrupted (perhaps by an adversary), then:

- The signature will have to change for it to be a valid event.
- If the signature changes, the ID of the event will also change.
- If the ID changes, the event is no longer referenced by any other event.
- Thus, the event must not have happened in the meeting.

The adversary will need to change all events up to the the END_ACK event, to be able to make their corrupted event part of the meeting, however since the END_ACK event is stored in the Ethereum blockchain, this becomes almost impossible to do, assuming that the Ethereum

blockchain is stable.

This argument also goes the other way. In order to prove that an event is part of a meeting, the user can traverse the tree and look for two conditions:

- If a path exists from the END_ACK event to that event.
- If a path exists from that event to the START event.

If both conditions are true, it can safely be concluded that the event is part of the meeting. These checks can be done using any tree traversal algorithm.

4.7 Blockchain and Meeting Contract

Once the IDs/signatures of the START and the END_ACK event have been generated, they need to be stored in the blockchain. As mentioned before, the Ethereum blockchain is used for this purpose. During the life cycle of a meeting, the following interactions take place:

- When the meeting starts, the server automatically generates a smart contract for that meeting and deploys it on the Ethereum network. The smart contract code (truncated) is shown in Listing 1. Please refer to Appendix F for the full listing.
- Once the meeting ends, the server uses `setEvents()` function to populate smart contract with START and END_ACK event IDs. The `setEvents()` function has a flag that only lets the owner (in this case the server) call it and it can only be called once.
- After the smart contract has been populated, the participants then fetch the contract ID from the REST API server and fetch the START and END_ACK event IDs.
- The participants verify if their version of events in included within the event tree (as mentioned in 4.6.8).
- If they are satisfied, they call the `approve()` function to sign the smart contract. The `approve()` function has a guard that only lets participants of the meeting call the function.

This allows all the participants, as well as the server to reach consensus on what events happened in the meeting, making it difficult to refute the events later.

```
1  contract MeetingContract {
2      struct Meeting {
3          string id;
4          mapping (address => bool) participants;
5          string eventStartHash;
6          string eventEndHash;
7          mapping (address => bool) approvals;
8      }
9
10     Meeting public meeting;
11
12     constructor (string memory id, address[] memory participants) public {
13         meeting.id = id;
14         // Populate participants' addresses - code not listed for brevity
15         owner = msg.sender;
16     }
17
```

```
18      function setEvents(string memory start, string memory end) public
19                                              onlyOwner eventsNotPopulated {
20          meeting.eventStartHash = start;
21          meeting.eventEndHash = end;
22          eventsPopulated = true;
23      }
24
25      function approve() public onlyParticipant{
26          meeting.approvals[msg.sender] = true;
27      }
28
29  }
```

Listing 1: Truncated meeting contract code in Solidity, this is deployed for each meeting.

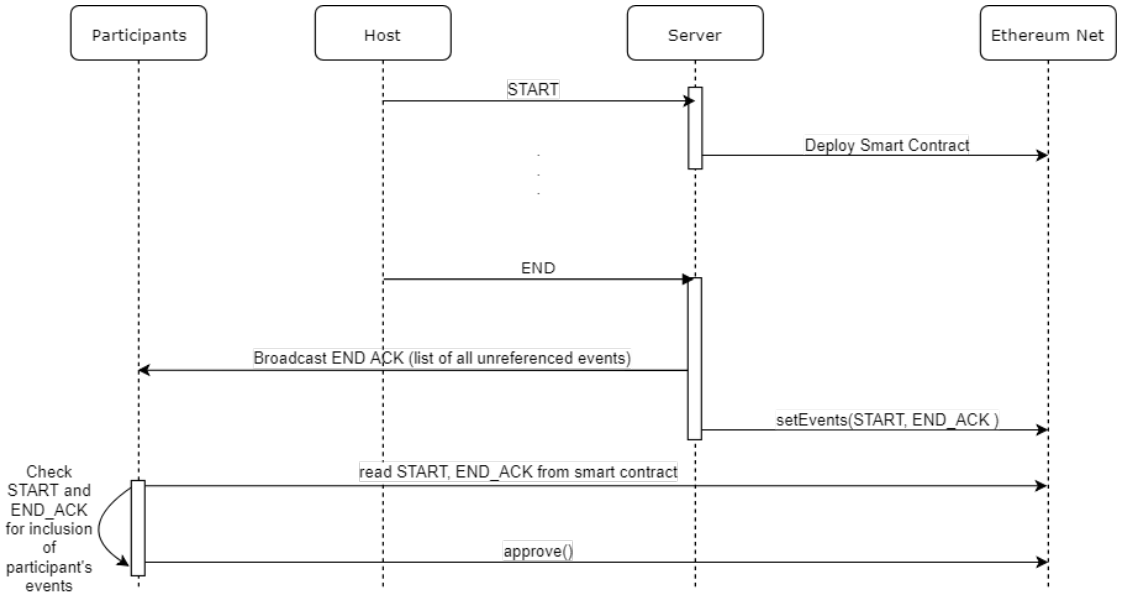


Figure 4.15: Sequence diagram showing interactions with the smart contract and Ethereum.

4.8 Implementation Choices

4.8.1 JSON and JSON Schema

In the implementation, JSON was used to represent the events (Listing 2). The reason for choosing JSON is that it is human readable (allowing for easier debugging), and also compatible with the various sub-systems (such as MongoDB). This choice however, created a need to check if the events are formed correctly. To do so, JSON Schema [26] was utilised. JSON schema is a vocabulary that allows for writing rules on how JSON documents should be formed. Rules for the event and all event content types were written using the vocabulary, and they can be found in F. Existing libraries [27] was then used to validate the JSON events before processing any of its content.

```
{
    "by": "0xb78e5bb6ff6a849e120985d32532e5067f262e19",
    "_id": "0xda5dfa7c0a7b0e020eba13d86a6a...",
    "timestamp": 1555282628,
    "refEvent": "0xbf9bbbf60077538eba133cfa72dd...",
    "meetingId": "b05674b1-400b-4411-a668-71203331c67a",
    "type": "leave",
    "content": {}
}
```

Listing 2: Example of an event implemented using JSON

4.8.2 Session Keys and Signature Scheme

The main purpose of the session keys are to sign the events created by the user. In the implementation, the JSON objects are first encoded using Bencode [28] which converts any JSON object into a string in a deterministic way (such as preserving the order of keys), an example of bencode encoding is given in Listing 3. This is done to make sure that when these events are hashed, they always result in the same string, regardless of how the JSON document is parsed. Then, Elliptic Curve Digital Signature Algorithm (ECDSA) is used for singing, which is the same scheme that is used by Ethereum. Web3.js [29], web3.py [30] and eth-crypto [31] libraries are used for performing these cryptographic functions. There were two reasons for using ECDSA:

- The keys are smaller than other schemes like RSA signature schemes [32] [33], as it uses Elliptic Curve Discrete Logarithm Problem (ECDLP). This helps reduce the size of START and JOIN events.
- The security of ECDSA and it's parameters are already well understood by the Ethereum community, as such the implementation of signature scheme was used as is in order to avoid implementing from scratch, which is out of scope for this project.

```
d3:_id33:0xda5dfa7c0a7b0e020eba13d86a6a...2:by42:0
    xb78e5bb6ff6a849e120985d32532e5067f262e197:contentde9:meetingId36:b05674b1-400
    b-4411-a668-71203331c67a8:refEvent33:0xbf9bbbf60077538eba133cfa72dd...9:
    timestampi1555282628e4:type5:leavee
```

Listing 3: Example of Bencode encoding: the JSON event in Listing 2 encoded in Bencode.

4.8.3 Websockets vs Peer to Peer (WebRTC)

One of the decisions that was made early on was whether to use server-client paradigm (and thus websockets) or a peer to peer connection (using WebRTC [34]). Some initial experiments with WebRTC was done, and it was realised that peer to peer connections is quite difficult to manage when there are 50+ participants. A lot of networking issues arise from using peer to peer system, such as using STUN [35] server for messaging, and using TURN [36] (relay) server as a fallback when a peer to peer cannot be formed. A performance slow down was also noticed when testing with 50 connections streaming lots of events, as such in order to simplify, a server-client model was used. Although this creates a central point of trust (as the server can selectively broadcast the event), but in my opinion, the Events protocol provides sufficient information to the user to detect and protest. In the future, an implementation of the Events protocol using peer to peer connections can be investigated.

5 Testing

This chapter will elaborate the testing strategies used in the development of this project. As this is a complex project, various tools were utilised to test and debug different components of the system.

5.1 Meeting Server

This is the most complex component in the c-meet system, as it implements the Events protocol, thus, testing it was difficult. Testing was done in two phases as described in sub sections below.

5.1.1 Unit Tests

The Meeting Server relies on various minor components that provide very specific functionality. For example, the *Auth* class is responsible for cryptogtaphic and authentication functionalities (signing, verification, etc.), the $MeetingContractHelper$ class, is responsible for all Ethereum contract interactions. These small components, for the most part, do not rely on other components and can be tested on their own. Furthermore, these components remained mostly unchanged as the development progressed; thus, unit tests were written to check the correctness of these components. Unit tests were also written for model classes such as *Meeting*, *Event*, *Patient*, etc. to test the marshalling of objects into JSON and vice versa. Although the model classes changed through out the development of this project, these unit tests helped catch bugs with interoperability, which often results in unusual behaviors.

5.1.2 Integration Testing

After making sure that all the minor components had the correct functionality, it was then necessary to make sure that they worked correctly when integrated together in the *meeting_server.py* script. The script is responsible for communicating meeting events with the web app (UI), as such, it was difficult to test without having the UI present. However, since the web app is a complex system in and of itself, relying on it to test the Meeting Server seemed like a bad strategy. Thus, it was necessary to develop a simple interface that would serve as an alternative to the web app, and so, the Meeting Server Client CLI was developed.

The Meeting Server Client CLI (see Figure 5.1) is a simple script written using Click [37] python library, that takes in the public and private key of any user, and generates signed meeting events (in JSON format), which is then sent to the Meeting Server using web sockets, similar to the web app. This allowed the integration tests to take place independent of the UI.

Since this was integration testing, the focus was mainly on whether the events were interpreted and handled properly by the Meeting Server, and not on the correctness of the c-meet system as a whole; however, some tests that are not traditionally considered "integration tests" (for e.g. behavior of the server

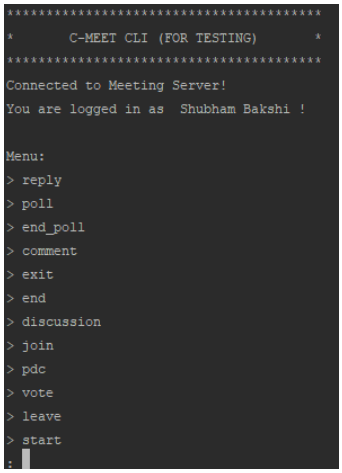


Figure 5.1: Meeting Server CLI used for testing of Meeting server without the web app.

when multiple users are logged in simultaneously) were also conducted, to eliminate any errors early.

5.2 REST API Server

The REST API server shares a lot of components with the Meeting Server (such as the *Auth* class), therefore, the unit tests that were developed for Meeting Server also helps the testing of REST API server. Additionally, integration tests were also written to check the correctness of the server, as well as manual tests using API development tools like Postman [38].

5.2.1 Mock Data Generation

In order to test the REST API server (and also the c-meet system as a whole), realistic mock data had to be generated. This data was generated using the *gen_sample.py* script. It generates patients (with realistic names and dob), staff members (with valid Ethereum keys as DeeID) and meetings (with valid staff and patients), and automatically populates the database with them. It also prints out the private keys for the staff members, which are kept safely for tests that require signing in.

5.3 UI Testing

The web app (UI) was tested separately from all other components. Angular provides powerful tools to perform unit and integration testing. It uses Jasmine testing framework, which encourages a behavior driven approach to testing [18]. The tests created for the web app mainly focus on the the UI components, and whether they render correctly. One example of this is given in Figure 5.2.

```
it('should render no meetings message, when list empty', () => {
  component.upcomingMeetings = [];
  fixture.detectChanges();

  const bannerElement: HTMLElement = fixture.nativeElement;
  const p = bannerElement.querySelector('#emptyUpcomingMessage');

  expect(p).toBeTruthy();
});
```

Figure 5.2: Example of an UI test written using Jasmine framework.

One of the biggest challenges when testing the UI was the need for DeeID app in order to login. This was especially challenging considering the fact that the wallet can only hold one key, as such allowing only one person to sign in at a time. There were attempts made to bypass the login system, however, since the DeeID system is integrated deeply into the c-meet system, it proved to be quite difficult. Furthermore, developing systems to bypass the DeeID login risks leaving back doors into the web app, and therefore risked making it insecure. In order to solve this problem, a DeeID wallet emulator was developed in JavaScript with the help of Sirvan Almasi (the author of DeeID). The emulator takes in the data generated by the QR code and the DeeID keys as a command line argument, and automatically does the authentication (described in Section 4.2) using similar libraries as the mobile app. This enabled the testing of the web app without the depending on the DeeID app.

5.4 Smart Contract Testing

The meeting contract is one of the choke points for integrity in the c-meet system. If the behavior of the contract is incorrect, it invalidates a lot of integrity guarantees that is achieved by the system. Furthermore, Ethereum smart contracts are permanent, i.e. once they are deployed, they cannot be modified. Therefore, it was necessary to write tests specifically for the meeting contract. Truffle Suite provides all the tools needed to test the deployment of a smart contract. It uses Mocha testing framework, which has a similar syntax to the Jasmine framework used by Angular, therefore the learning curve for writing smart contract tests was gentle. The tests focused on the contract functions, and checking whether they perform correctly when invoked by an Ethereum account. They also checked whether the functions reject unauthorised accounts form changing the state. An example of a smart contract test is given in Figure 5.3

```
it('non participant approving throws error', async () => {
  const contract = await MeetingContract.deployed();
  try {
      await contract.approve()
  } catch (error) {
      err = error
  }
  assert.ok(err instanceof Error)
})
```

Figure 5.3: Example of a smart contract test written using Truffle framework.

5.5 End to End Testing

After performing the unit and integration tests, and making sure that the system had the correct behavior, it was decided to perform end to end tests. The test cases were prepared using the use cases mentioned in section 3.6. The end to end tests were designed to check whether the system can perform correctly, when real users are involved, as such, no components or data (apart from patient) were mocked during the testing process. The testing was performed manually and the full list of tests can be found in Appendix B.

5.6 Testing Summary

Although testing was not the main focus of this project, as it is a proof of concept, the tests did boost confidence that the results that was acquire from this investigation is sound and correct. Various strategies were applied to overcome significant challenges that materialise from testing a complex system. Automated unit and integration tests were performed for the servers, the web app and the smart contract, as well as manual end to end tests to check the correctness of the system as a whole.

6 Evaluation

This section evaluates the achievements and the limitations of the c-meet system against the requirements that was set out initially. It also evaluates the cost and performance of operating the system.

6.1 Goals Evaluation

The goals that were set out in Chapter 1 were:

- An architecture for storing meeting metadata in a way that can be audited later, with data integrity, authenticity and accountability in mind.
- A proof-of-concept implementation of the architecture (including servers).
- A proof-of-concept of a web app (UI) that will be used by the users to participate in the meetings.
- A login system that utilizes DeeID identification scheme.

All of the four goals were delivered. An architecture was presented that utilized Ethereum smart contracts and the Events protocol (Section 4.6) to guarantee data integrity. The architecture integrated with the DeeID system to provide a concrete notion of identity, and there by providing authenticity. An implementation of the system was also provided which proved the viability of the system, and finally, an interpretation of the UI (as a web app) was provided.

6.2 Requirements Evaluation

The requirements described in Chapter 3 provided the project with specific targets for the deliverables to target, so it's logical to go back to them, and check if they have been met. Upon inspection, the following can be interpreted:

- Must have requirements met: 13/13 (100%).
- Should have requirements met: 4/4 (100%).
- Could have requirements met: 6/7 (86%).
- Would have requirements met: 1/4 (25%).

Overall, 24 out of the 28 requirements were met, the ones that were not achieved have been analysed below:

- **F2.5 and F2.9:** The requirements stated that the participants would be able to record audio and the host would be able to record their screen. As time was limited, this was not prioritised, as it didn't contribute much to the architecture of the system, and required a substantial amount of work.
- **S1.1:** The requirement stated that the privacy of a voter in a meeting poll should be protected. This proved to be very challenging, as the voter also needs to be held accountable for their vote. Although, this can perhaps be achieved using decentralised voting systems and zero knowledge proofs [39], however, such a system would be quite complex to implement, and is out of scope of this project. After consultation with GOSH, the requirement was deemed a non priority, and thus, it was not worked on.

- **S1.5:** The requirement was to open source all the code. Although, this could have been achieved, it was decided otherwise, as further work is required to make sure that no sensitive information is leaked in the source code, as well as, to make sure that proper protocols are followed before open sourcing. This can however be achieved in the future.

6.3 Cost Evaluation

6.3.1 Data generated by the Events Protocol

The Events protocol uses substantial amount of cryptography to achieve some of the goals, as such, the system generates lot of data (from hashes, keys etc.). As this was an investigation, it was necessary to evaluate the amount of data that needs to be stored in the database, to determine the system's practicality. Simulations of meetings were done to get an idea of how much data is generated per meeting, however as these were simulations, reasonable assumptions had to be made. The assumptions were as follows:

- Number of patients: 20
- Length of poll question: 100 characters
- Number of options per poll: 5
- Number of comments/replies per participants: 20
- Length of comments/replies: 100 characters
- Data per patient: 500 characters

Using these assumption, simulations were ran with varying number of participants, and the results of how much data was generated are shown in Figure 6.1. Please refer to Appendix A to see the amount of data generated per event.

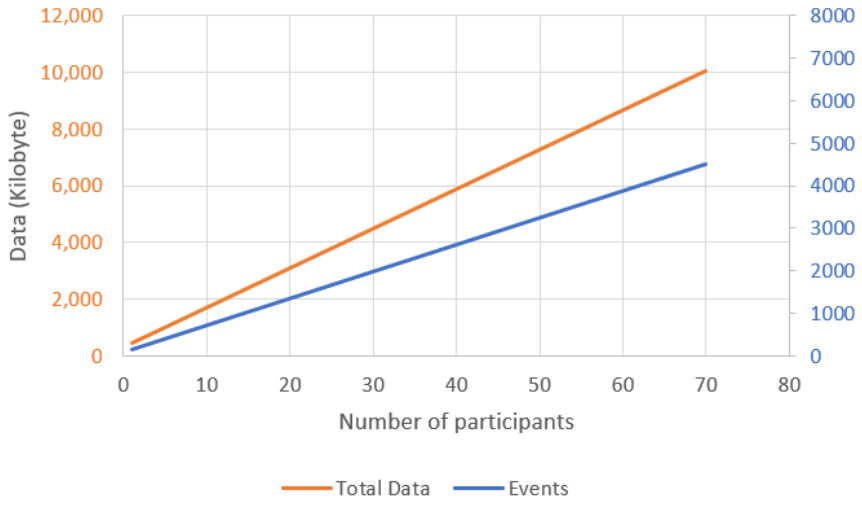


Figure 6.1: The amount of data generated per meeting.

As can be seen form figure 6.1, the amount data generated from a meeting grows linearly in relation to the number of participants. As MDT meetings have approximately 50 participants, one can expect to generate approximately **7 MB of data per MDT meeting**.

6.3.2 Cost of interacting with the smart contract

As mentioned in Chapter 4, smart contracts are utilised to store event IDs for permanence. Interacting with the smart contract costs money, as the miner that is responsible for executing the smart contract code needs to be compensated. The cost of interacting with smart contract depends on the following:

- The amount of data that is stored in the smart contract account (using arrays, maps etc.).
- The amount of computation done by the smart contract (based on the opcodes of the Ethereum Virtual Machine [40]).
- The GAS price advertised in the transaction. The GAS is essentially a unit of computation, so the price of GAS is how much the caller is willing to pay for each unit of computation. The higher the GAS price, the lower the time for the confirmation, as more nodes will be attracted to complete the transaction.

Functions that do not change the state of the smart contract are not charged in the Ethereum blockchain, as such, all read functions can be computed for free. However, there are three interactions in the c-meet system which change the state of the meeting contract. The functions along with the transaction GAS cost are as follows:

- Initial deployment of the contract: 1124564 GAS
- Storing the START and END_ACK event IDs ($setEvents()$): 289047 GAS
- Approval from each participants ($approve()$): 42115 GAS

Assuming that there are 50 participants in an average MDT meeting, and assuming a reasonable GAS price for confirmation time of approximately one minute, the total cost is:

$$Total\ GAS = 1124564 + 289047 + (50 * 42115) = 3519361$$

$$GAS\ Price:\ 11.1\ GWEI$$

$$Total\ cost\ (as\ of\ 28\ April\ 2019):\ \textbf{6.172 USD per meeting}$$

6.4 Limitations

The limitations of the c-meet system are as follows:

- **DeeID:** The DeeID system provides the identity layer for the c-meet system. It is a critical part of the solution, and without it, the solution cannot guarantee the authenticity of any events that take place in a meeting. The DeeID system proves to be a choke point for the whole solution, which is a limitation. If the DeeID wallet is compromised or if the participant's phone (containing the DeeID keys) is stolen, the adversary can then forge events on behalf of the participant, thus breaking the guarantees of authenticity. Fortunately, the DeeID author has provided protocols on what to do, if such thefts occur, for example, the victim could re-register (as described in Section 2.2) with GOSH as soon as possible, thereby nullifying the old keys. However, such systems need to be in place, and maintained by GOSH.

- **Data recovery:** The c-meet system stores only the very necessary information in the Ethereum blockchain to minimise the costs. If an adversary tampers with the events that are stored in the system database, or deletes the database itself, the system can only detect the tampering or deletion of data. It cannot recover the data back from the blockhain, which is a major limitation. Therefore, appropriate backup and security systems need to be in place to combat such acts.

- **Ethereum** Ethereum itself becomes a limitation for the system, as the system assumes that the Ethereum blockchain is stable and will be for the foreseeable future. If for any reason, the Ethereum blockchain collapses, the c-meet system cannot guarantee the integrity or authenticity of any meetings that occur in the future or in the past.

6.5 Summary

In summary, the project has achieved all the goals that was set out initially, and most of the requirements in Chapter 3. The amount data generated by c-meet system is reasonable at around 7 MB per meeting (assuming 50 participants), and the cost of interacting with the blockchain is around 6.17 USD per meeting, which is also reasonable considering the benefits the solution provides.

7 Conclusions and Future Work

7.1 Future Work

7.1.1 Mobile client for participants

In this project, an interpretation of the UI was provided through a web app. However, this is not ideal for all participants as they may not have access to a desktop/laptop during a meeting. Thus, a mobile client needs to be developed that implements the Events protocol. The most significant challenge that would come about when developing the mobile app, would be implementing the login system. Currently a QR code is used to interact with the DeeID app. However, with the mobile client, the interaction needs to happen through Inter Process communication (IPC). This would require the modification of the DeeID app to accept objects through IPC. Another challenge when developing a mobile client would be access to an Ethereum node for smart contract interactions. In the web app, this is provided by Metamask, however no such bridge exists for mobile apps yet, although it is in development [41].

7.1.2 Integration with InfoFlex system

In the investigation, a database of patients with the name, date of birth and ID was assumed to exist. In a deployment scenario, such database will be handled by the InfoFlex system. Therefore, understanding the interface and the data standards of this system is important to avoid interoperability issues. There were attempts made to procure such systems during the investigation, but unfortunately the attempts were unsuccessful.

7.1.3 Improving the Voting system

Currently, the voting system in c-meet is quite centralised. The host owns the decryption key for votes during a poll, allowing them to corrupt the results by revealing them before the end of the poll. Although this is an acceptable solution under the assumption that the host has no incentive do so, it however begs the question whether there is a better way of conducting such polls. Furthermore, there were concerns raised by GOSH staff members that the participants would prefer casting anonymous votes, as such, a system could be investigated that uses advanced cryptography to protect the privacy of the voter as well as hold them accountable under certain conditions.

7.1.4 Investigation into Private/Hybrid Blockchain

The c-meet system relies on the public Ethereum blockchain for the permanence of events and their order. However, for reasons concerning costs and privacy, it may not be possible store meeting data on the public blockchain, as such, a private blockchain (such as Quorum) could be deployed. However doing so risks diluting the non repudiation property of the system, as the organisation (in this case GOSH) hosting the blockchain will possess a significant amount of power on the consensus protocol, thereby being able to control the order of blocks. There could be hybrid solutions, for example, having a private blockchain running on all GOSH computers, and having hooks to a public blockchain, but this will need to be investigated further.

7.1.5 Further tests

The testing methodology mostly relied on simulations and predetermined scenarios, as such, it was difficult to know how the system will perform under unknown conditions. A full fledged testing with real users needs to be conducted to verify if the system behaves correctly. This would require collaboration with DeeID developer, as all the staff and patients need to be registered with the DeeID system to participate in the meeting.

7.2 Conclusion

The project was conceived to investigate whether blockchain technology could be used to record GOSH MDT meetings. The motivation being that the participants need to be held accountable for the actions and decisions they take during the meeting. The c-meet system provides a proof of concept implementation of a system that is able to store the meeting events in a way that is resistant to tampering by any entity. It utilizes the Ethereum blockchain to store key pieces of information from every meeting, which can be used later to determine the integrity of all the events. The key pieces of information that is stored in the blockchain is constant and does not depend on the length of the meeting, thus allowing the cost of interacting with the Ethereum blockchain to remain low. In order to guarantee the authenticity of the events, a novel system of identity known as DeeID was used. Finally, the Events protocol that forms a major part of the c-meet system was developed specifically for this project, but can be abstracted to have uses in other projects as well.

References

[1] NHS. Supporting information: Multidisciplinary team meeting. https://www.datadictionary.nhs.uk/data_dictionary/nhs_business_definitions/m/multidisciplinary_team_meeting_de.asp, Accessed: 2019-04-16.

[2] Steven M. Seitz Supasorn Suwajanakorn and Ira Kemelmacher-Shlizerman. Synthesizing obama: Learning lip sync from audio. 2017.

[3] Cryptocurrency market capitalizations. https://coinmarketcap.com, Accessed: 2019-04-16.

[4] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. 2008. https://bitcoin.org/bitcoin.pdf.

[5] Bitcoin developer guide. https://bitcoin.org/en/developer-guide, Accessed: 2019-04-16.

[6] Meni Rosenfeld. Analysis of hashrate-based double-spending. 2013. https://bitcoil.co.il/Doublespend.pdf.

[7] Dan; Hauser Carl; Irish Wes; Larson John; Shenker Scott; Sturgis Howard; Swinehart Dan; Terry Doug Demers, Alan; Greene. Epidemic algorithms for replicated database maintenance. *Proceedings of the Sixth Annual ACM Symposium on Principles of Distributed Computing*, page 1–12, 1987.

[8] Ethan Heilman, Alison Kendler, Aviv Zohar, and Sharon Goldberg. Eclipse attacks on bitcoin's peer-to-peer network. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 129–144, Washington, D.C., 2015. USENIX Association.

[9] Gavin Wood. Ethereum: A secure decentralised generalised transaction ledger. 2014.

[10] The solidity programming language. https://solidity.readthedocs.io/en/latest/, Accessed: 2019-04-16.

[11] JP Morgan. Quorum. https://www.jpmorgan.com/global/Quorum, Accessed: 2019-04-16.

[12] Sirvan Almasi. Decentralised identity and data management network, 2018.

[13] Uriel Feige, Amos Fiat, and Adi Shamir. Zero-knowledge proofs of identity. *Journal of Cryptology*, 1(2):77–94, Jun 1988.

[14] Infoflex. https://infoflex.co.uk/, Accessed: 2019-04-16.

[15] Truffle suite (source code). https://github.com/trufflesuite/truffle, Accessed: 2019-04-16.

[16] Ganache. https://truffleframework.com/ganache, Accessed: 2019-04-16.

[17] Infura. https://infura.io/docs/gettingStarted/makeRequests, Accessed: 2019-04-16.

[18] Angular. https://angular.io/docs, Accessed: 2019-04-16.

[19] Jim Gray. The transaction concept: Virtues and limitations. *Proceedings of the 7th International Conference on Very Large Databases*, page 144–154, Sep 1981.

[20] M. Jones, J. Bradley, and N. Sakimura. Json web token (jwt). RFC 7519, RFC Editor, May 2015. `http://www.rfc-editor.org/rfc/rfc7519.txt`.

[21] Angular cli command reference. `https://angular.io/cli`, Accessed: 2019-04-16.

[22] Reactive extensions library for javascript. `https://rxjs-dev.firebaseapp.com/guide/overview`, Accessed: 2019-04-16.

[23] Metamask - brings ethereum to your browser. `https://metamask.io/`, Accessed: 2019-04-16.

[24] I. Fette and A. Melnikov. The websocket protocol. RFC 6455, RFC Editor, December 2011. `http://www.rfc-editor.org/rfc/rfc6455.txt`.

[25] Paul J. Leach, Michael Mealling, and Rich Salz. A universally unique identifier (uuid) urn namespace. RFC 4122, RFC Editor, July 2005. `http://www.rfc-editor.org/rfc/rfc4122.txt`.

[26] A. Wright and H. Andrews. Json schema: A media type for describing json documents. Technical report, IETF, November 2017. `https://tools.ietf.org/html/draft-handrews-json-schema-01`.

[27] Json schema validator for python. `https://github.com/Julian/jsonschema`, Accessed: 2019-04-16.

[28] Bram Cohen. The bittorrent protocol specification. BEP 03, BitTorrent.org, January 2008. `http://www.bittorrent.org/beps/bep_0003.html`.

[29] Web3.js (source code). `https://github.com/ethereum/web3.js`, Accessed: 2019-04-16.

[30] Documentation for web3.py. `https://web3py.readthedocs.io/en/stable/index.html`, Accessed: 2019-04-16.

[31] Cryptographic javascript-functions for ethereum (source code). `https://github.com/pubkey/eth-crypto`, Accessed: 2019-04-16.

[32] R. L. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Commun. ACM*, 21(2):120–126, February 1978.

[33] Mishall Al-Zubaidie, Zhongwei Zhang, and Ji Zhang. Efficient and secure ecdsa algorithm and its applications: A survey. 02 2019.

[34] Anant Narayanan, Cullen Jennings, Bernard Aboba, Jan-Ivar Bruaroey, Daniel Burnett, Adam Bergkvist, and Taylor Brandstetter. WebRTC 1.0: Real-time communication between browsers. Candidate recommendation, W3C, September 2018. https://www.w3.org/TR/2018/CR-webrtc-20180927/.

[35] J. Rosenberg, R. Mahy, P. Matthews, and D. Wing. Session traversal utilities for nat (stun). RFC 5389, RFC Editor, October 2008. `http://www.rfc-editor.org/rfc/rfc5389.txt`.

[36] R. Mahy, P. Matthews, and J. Rosenberg. Traversal using relays around nat (turn): Relay extensions to session traversal utilities for nat (stun). RFC 5766, RFC Editor, April 2010. http://www.rfc-editor.org/rfc/rfc5766.txt.

[37] Welcome to click - click documentation. https://click.palletsprojects.com/en/7.x/, Accessed: 2019-04-16.

[38] Postman — api development environment. https://www.getpostman.com/, Accessed: 2019-04-16.

[39] Xuechao Yang, Xun Yi, Surya Nepal, and Fengling Han. Decentralized voting: A self-tallying voting system using a smart contract on the ethereum blockchain. In Hakim Hacid, Wojciech Cellary, Hua Wang, Hye-Young Paik, and Rui Zhou, editors, *Web Information Systems Engineering – WISE 2018*, pages 18–35, Cham, 2018. Springer International Publishing.

[40] Evm opcode gas coststs. https://github.com/djrtwo/evm-opcode-gas-costs/blob/master/opcode-gas-costs_EIP-150_revision-1e18248_2017-04-12.csv, Accessed: 2019-04-16.

[41] Metamask develops mobile client. https://www.ethnews.com/metamask-develops-mobile-client, Accessed: 2019-04-16.

A Miscellaneous

A.1 Size of events

Event Type	Data (Bytes)
START	3803
POLL	1056
POLL_END	661
DISCUSSION	631
PATIENT_DATA_CHANGE	5625
END	535
JOIN	1218
LEAVE	547
VOTE	1614
COMMENT	723
REPLY	688
DISAGREEMENT	554
ACK	545
ACK_JOIN	4106
ACK_POLL_END	612 + (132 * Participants)
ACK_END	575 + (132 * Event count)

B Test Listings

B.1 End to End Test Descriptions

- **Test 1:** Login successful
  - **Prerequisites:** None
  - **Steps:** Navigate to any page
  - **Expected Results:** Naviagated to Meeting List Page, all meetings appear.
- **Test 2:** Login failure
  - **Prerequisites:** None
  - **Steps:** Navigate to any page
  - **Expected Results:** QR code still appears, an error message also appears.
- **Test 3:** Login timeout (DeeID WS server killed)
  - **Prerequisites:** None
  - **Steps:** Navigate to any page
  - **Expected Results:** QR code still appears, an error message also appears.
- **Test 4:** Meeting creation
  - **Prerequisites:** Test 1
  - **Steps:** Click 'New meeting', populate the fields, Click 'Create'
  - **Expected Results:** Navigated back to Meeting List Page, new meeting appears.
- **Test 5:** Meeting deletion
  - **Prerequisites:** Test 1
  - **Steps:** Click 'edit' on existing meeting, Click 'Delete'
  - **Expected Results:** Navigated back to Meeting List Page, the meeting disappears.
- **Test 6:** Meeting update
  - **Prerequisites:** Test 1
  - **Steps:** Click 'edit' on existing meeting, change the date of meeting, Click 'Update'
  - **Expected Results:** Navigated back to Meeting List Page, the meeting has the updated date.
- **Test 7:** Start meeting
  - **Prerequisites:** Test 4
  - **Steps:** Click 'start' on existing meeting
  - **Expected Results:** Navigated to Meeting Page, no errors, the events stream show 'start' and 'join' events.
- **Test 8:** Join meeting

  - **Prerequisites:** Test 4 and Test 7 (from another client)
  - **Steps:** Click 'join' on existing meeting
  - **Expected Results:** Navigated to Meeting Page, no errors, 'Create poll', 'Patient Data' and 'Discuss' options are disabled, the events stream show 'join' events.

- **Test 9:** Discuss patient
  - **Prerequisites:** Test 7
  - **Steps:** Click on 'Discuss' on any patient
  - **Expected Results:** 'Discuss' event of the appropriate patient appears in the stream

- **Test 10:** Comment
  - **Prerequisites:** Test 7 or Test 8
  - **Steps:** Click on 'Comment' tab. Populate the text box. Click 'Comment'
  - **Expected Results:** Comment event appears (with the comment) in the stream

- **Test 11:** Reply
  - **Prerequisites:** Test 10
  - **Steps:** Click on 'Reply' on a comment event. Populate the text box. Click 'reply'
  - **Expected Results:** Reply event appears (with the reply) in the stream

- **Test 12:** PDC
  - **Prerequisites:** Test 7
  - **Steps:** Click on 'Patient data'. Populate the fields. Click 'update'
  - **Expected Results:** PDC event appears in the stream

- **Test 13:** Poll
  - **Prerequisites:** Test 7
  - **Steps:** Click on 'Create poll'. Populate the fields. Click 'Create'
  - **Expected Results:** Poll event appears, with the options and question

- **Test 14:** Vote
  - **Prerequisites:** Test 13
  - **Steps:** Click on 'Vote' of a Poll event. (Options should appear). Click on any option. Click 'Vote'
  - **Expected Results:** Vote event appears

- **Test 15:** Vote not decryptable
  - **Prerequisites:** Test 14
  - **Steps:** Click on 'see vote' on a vote event.
  - **Expected Results:** alert box opens stating the vote cannot be decrypted.

- **Test 16:** Poll end

- **Prerequisites:** Test 13
- **Steps:** Click on 'end poll' on a poll event.
- **Expected Results:** Poll end event appears.

- **Test 17:** Vote inclusion
  - **Prerequisites:** Test 14
  - **Steps:** Click on 'Includes my vote' on a vote event.
  - **Expected Results:** alert box opens stating the vote is included.

- **Test 18:** Poll results
  - **Prerequisites:** Test 14
  - **Steps:** Click on 'See reults' on a poll event.
  - **Expected Results:** the right panel shows a pie chart of the results

- **Test 19:** Leave meeting
  - **Prerequisites:** Test 8
  - **Steps:** Click on 'Leave meeting' on the header panel.
  - **Expected Results:** Navigated back to Meeting List page. Download of events initiated.

- **Test 20:** End meeting
  - **Prerequisites:** Test 7
  - **Steps:** Click on 'End meeting' on the header panel.
  - **Expected Results:** Navigated back to Meeting List page. Download of events initiated. The meeting shown as a past meeting.

C Use Cases

C.1 Use Case Documents

ID	1
Actor	User
Pre-conditions	Already Logged In
Description	Create a meeting
Flow	1) User clicks on 'create meeting' button 2) User populates the appropriate fields 3) User clicks 'create' button 4) The web app sends the meeting using the RESTful API 5) The database gets updated with new meeting
Post-conditions	User has successfully created a meeting

ID	2
Actor	User
Pre-conditions	User has already Logged In
Description	Cancelling a meeting
Flow	1) User clicks on 'edit meeting' button 2) User clicks 'delete' button 3) The web app sends the request using RESTful API 5) The meeting is deleted from the database
Post-conditions	User has successfully cancelled a meeting

ID	3
Actor	User
Pre-conditions	User has already Logged In
Description	Changing a meeting
Flow	1) User clicks on 'edit meeting' button 2) User populates the appropriate fields 3) User clicks 'update' button 4) The web app sends the meeting using the RESTful API 5) The database gets updated
Post-conditions	User has successfully updated a meeting

ID	4
Actor	User
Pre-conditions	None
Description	Log In
Flow	1) User navigates to '/login' page 2) User uses DeeID app to scan the QR code 3) The DeeID system authenticates 4) The web app verifies the authentication 5) The web app provides auth object to REST API server 6) The REST API server authenticates and creates JWT session 7) The web app upon receiving JWT navigates to Meeting List page
Post-conditions	User has successfully logged in

ID	5
Actor	Host
Pre-conditions	Host is logged in and a meeting is created
Description	Start a meeting
Flow	1) Host clicks on start meeting 2) Web app generates OTP and sends START event to Meeting Server 3) Meeting server authenticates and creates room 4) Meeting server sends ACK 5) Web app sends the JOIN event 6) Meeting server validates event 7) Meeting server sends the JOIN_ACK event 8) Web app renders the Meeting Page
Post-conditions	Host has successfully started the meeting

ID	6
Actor	Host
Pre-conditions	Host has started a meeting
Description	Ending a meeting
Flow	1) Host clicks on end meeting button 2) Web app sends LEAVE and END events to Meeting Server 3) Meeting server validates and responds with ACK and ACK_END events 4) Web app navigates back to Meeting List page 5) Web app initiates download of all events
Post-conditions	Host has successfully ended the meeting

ID	7
Actor	Host
Pre-conditions	Host has started a meeting
Description	Creating a poll
Flow	1) Host clicks on create poll button 2) Host populates the desired fields, and clicks 'create' 3) The Web app sends the POLL event with the poll data to Meeting server 4) The POLL event is validated by the Meeting server 5) Meeting server responds with ACK 6) The web app renders the poll event
Post-conditions	Host has successfully created a poll

ID	8
Actor	Host
Pre-conditions	Host has started a meeting
Description	Change patient meeting data
Flow	1) Host clicks on 'Patient Details button' 2) Host populates the desired fields, and clicks 'update' 3) The Web app sends the event to the Meeting server 4) The event is validated by the Meeting server 5) Meeting server updates the database 6) Meeting server responds with ACK
Post-conditions	Host has successfully updated patient meeting data

ID	9
Actor	Participant
Pre-conditions	Participant is logged in and Host has started a meeting
Description	Join meeting
Flow	1) Participant clicks 'join' 2) Web app prompts for OTP 3) Participant enters OTP and clicks 'Submit' 4) The web app sends the JOIN event 5) The Meeting server authenticates and responds with JOIN_ACK 6) The web app renders the Meeting page
Post-conditions	Participant has successfully joined a meeting

ID	10
Actor	Participant
Pre-conditions	Participant has joined a meeting
Description	Leave meeting
Flow	1) Participant clicks 'leave meeting' button 2) The web app sends the LEAVE event 3) The Meeting server authenticates and responds with ACK 4) The web app navigates to Meeting List page 5) The web app initiates download of all events.
Post-conditions	Participant has successfully left the meeting

ID	11
Actor	Participant
Pre-conditions	Participant has joined a meeting
Description	Comment
Flow	1) Participant clicks 'comment' tab 2) Participant write the comment, and clicks 'comment' button 3) The Meeting server authenticates and responds with ACK 4) The web app renders the comment event
Post-conditions	User has successfully lodged a comment

ID	12
Actor	Participant
Pre-conditions	Participant has joined a meeting and Host has created a poll
Description	Vote
Flow	1) Participant clicks 'vote' 2) Participant is presented with voting choices 3) Participant selects an options and clicks 'vote' 4) The web app encrypts the vote with the voting key 5) The web app sends the VOTE event to Meeting server 6) The server validates and sends an ACK.
Post-conditions	User has successfully voted in a poll

D System Manual

The code can be obtained from the following Github repos:

- Servers: `https://github.com/bakshi41c/mdt_server`
- Web App: `https://github.com/bakshi41c/mdt_web`

Please send an email to `shubham.bakshi.13@ucl.ac.uk` for access to these repositories.

D.1 Setting up the database

- Install MongoDB from `https://www.mongodb.com/`
- Make sure MongoDB is running on localhost:27017
- To visualise the data, MongoDB Compass can also be used (`https://www.mongodb.com/products/compass`)

D.2 Setting up the servers

- Make sure you have Python3.7+ installed
- Navigate to the code directory
- Create a python virtual environment
- Install the requirements using the command `pip install -r requirements.txt`
- Open `config.py` and check if all the configurations are appropriate
- Generate some test data using the command `python gen_sample.py`
- Run the HTTP server using command `python rest_api_server.py`
- Run the Meeting server using command `python meeting_server.py`

D.3 Setting up the web app

- Make sure you have Node v10+ and NPM installed (urlhttps://nodejs.org/en/)
- Navigate to the code directory
- Install angular-cli using command `npm install -g @angular/cli`
- Run command `npm install`. This should install all dependancies in a /node_modules folder.
- Run command `ng serve`
- The web app should now run on `http://localhost:4200`

D.4 Setting up DeeID app

- To install the DeeID app, follow instructions on `https://github.com/sirvan3tr/OmneeMobileApp`

D.5 Setting up DeeID WS server

- Make sure you have Python3.7+ installed
- To install the DeeID WS server, clone the repository: `https://github.com/deeid/websocket_server`
- Run command `python main_server.py`

D.6 Setting up Ganache (Private Ethereum network)

- Install Ganache and Truffle from `https://trufflframework.com/ganache`
- Run Ganache
- Setup a workspace by following the instructions on screen.
- The Ganache should setup a private blockchain, and a screen resembling Figure D.1 should appear
- Go to settings by clicking the cog button on top right corner. Then navigate to Server, and make sure the server is running on http://127.0.0.1:8545

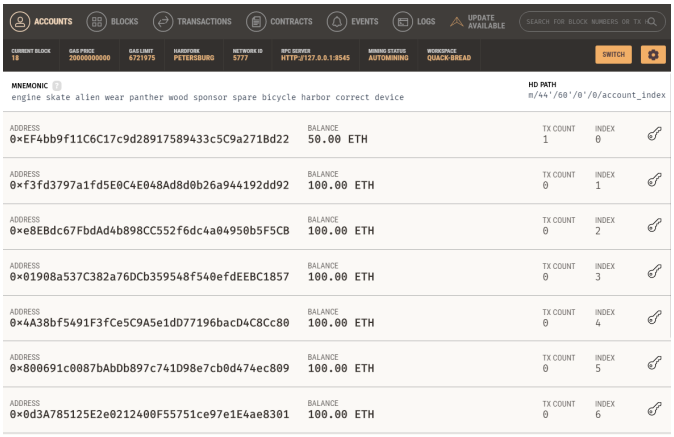


Figure D.1: Ganache Home screen

D.7 Setting up Metamask

- Make sure to have the Metamask plugin installed (currently only working for Chrome and Firefox. Link: `https://metamask.io/`
- Follow the setup instructions for Metamask
- Open the Metamask plugin
- Click on the network button (it usually says "Ropsten test network")
- Click on "custom RPC"

- A screen resembling Figure D.2 should appear.
- Fill the details with the details from Ganache. (URL: "http://127.0.0.1:8545", Network/Chain ID: 5777) and click 'Save'.
- Metamask should now be connected to Ganache Ethereum network

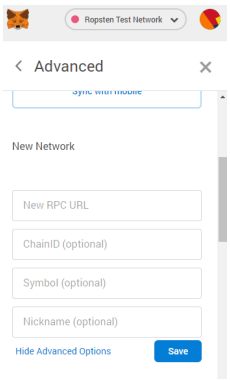


Figure D.2: Metamask Custom RPC setup Screen

D.8 Topping up the server Ethereum account

- Go to Metmask, and import the server account by importing the `server_sample_eth_key` file into Metmask.
- Go to Ganache, and pick any account (address), and click the key button on the right. This should reveal the private key.
- Import the Ganache account into Metamask using the private key (these account are usually pre topped up with 100 ETH).
- Perform a transaction, by sending some Ether from the Ganache Ethereum account to the server Ethereum account.
- The server Ethereum account should now have some Ether, which the servers can use to perform transactions.

E Documents

E.1 Project Plan

See next page

Name: Shubham Bakshi

Project Title: Application of Blockchain in Multidisciplinary team meetings in NHS (GOSH)

Supervisor's Name: Graham Roberts

External Supervisor's Name: Prof. Neil Sebire, Sirvan Almasi

Aims:

Investigate the practicality of blockchain for the purposes of monitoring, recording and auditing Multidisciplinary Team meetings (MDT meetings) in NHS (Great Ormond St. Hospital).

Objectives:

1. Review blockchain and how it is currently used for identity.
2. Review how current MDT meetings are held and formalize the format, as well as look into current standards in NHS for data sharing, such as FHIR API, Epic EHR etc.
3. Develop software tools that will aid in the recording of meetings, for e.g. app to record votes, attendance, agenda etc.
4. Extend/Modify OMNEE (a decentralised system for identity using blockchain developed by another UCL/Imperial student) for MDT meetings.
5. Investigate and create new crypto systems for the purposes of voting, with privacy as well as auditability in mind.
6. Evaluate how useful these tools are, and whether such systems can be used in real life without a change in current legislation.

Deliverables (in order of importance):

- A back-end system and an architecture for storing meeting metadata in a way that can be audited later, with privacy and integrity in mind.
- A front-end system (e.g. web app/ mobile app) for recording MDT meetings.
- A literature survey of current systems and technologies that attempt to solve the problem of decentralised identity and privacy, especially using blockchain.
- Evaluation of the entire system and its usefulness, by conducting surveys, and performing scenarios where candidates lie.

Timeline:

- **Now until end of November:** Literature review; having a deep understanding of OMNEE, Ethereum and their accompanying crypto systems. Also, understanding MDT meetings and the requirements from the client.
- **End of November – mid December:** Designing the architecture of the system from bottom up, for store meeting meta data and enable auditing.
- **Mid-December – mid-January:** Implementation of all major components of the back-end system.
- **Mid-January – mid-February:** Front-end app that enables users to interact with the system and record their meetings.
- **Mid-February – mid-March:** Testing and evaluation; performing surveys and tests on the effectiveness of the system.
- **March – mid-April:** Final Report

E.2 Interim report

See next page

Interim Report

Name: Shubham Bakshi
Project Title: Application of Blockchain in Multidisciplinary team meetings in NHS (GOSH)
Supervisor's Name: Graham Roberts
External Supervisor's Name: Prof. Neil Sebire, Sirvan Almasi

Progress made to date

1. Literature survey of existing technologies

Understood blockchain and cryptocurrency thoroughly from transaction layer, to consensus layer and finally the network layer. Understood decentralised identities and the current state of the art (e.g. uPort). Sirvan helped me understand deeID (formerly OMNEE), which he developed last year. Researched voting systems that can be used for patient voting in MDT meetings, for e.g. there are many self-tallying voting protocols which do not require a centralised authority. The write up for the literature survey is still being worked on, and it currently exists as a set of bullet points, this will be finished at the end.

2. User stories and Moscow specification

At the end of November, I created a requirement list using the MoSCoW method to understand what features need to be prioritised. One of the must haves was having an ability to record attendance of people who were in the room. The should haves were voting system, as well as storing meeting data and finally a could have was recording what was shown in the meeting. At the start of Term 2, I was invited into an actual MDT meeting to understand what happens, how they operate, what is discussed and how consensus is reached on each patient. Based on the notes taken in the meeting, I created user stories to have a clear idea on what roles would exist within the system, and how they will interact with it.

3. UI Flow of the entire system

Following from the user stories, I also created a sketch of how the UI would look and what the flow will be for users with different roles. While developing this, we realised that we would need two apps, one for the participant, which would be a mobile app used for voting and leaving comments, and one for the host of the meeting which would be a desktop/web app – used for creating polls and editing patient data for that meeting (e.g. outcome, diagnosis etc). Although this means more work, I'm confident I will be able to finish on time.

4. Implementation of the server

I've started implementing the server which will be responsible for managing meetings, users, and posting signed data to blockchain. Interaction with the server will be through a REST API. The design of the REST API has been finalised, and I have implemented it using a Flask server. I'm using a no-SQL database (MongoDB) to store the meeting data. The server can now be used to do CRUD operations on meetings, as well as fetch patient and staff data.

5. Implementation of the Web/Desktop app

In tandem with the server, I've also started creating the web app (which is mainly UI). The web app is being created using Angular and Bootstrap (for CSS stylesheets). It can currently be used to create meetings.

Remaining work to be done

- **Server – By end of February**
    - Integration with Infoflex database – Currently I'm using a mock database for patients. This will need to be replaced by Infoflex which is used by GOSH for storing patient data.
    - Storing meeting meta data in smart contracts – The attendance, polls and patient data changes that take place in MDT meeting won't be stored in the blockchain as they are too big and will bloat up the blockchain. Instead a hash of them will be stored. However, this creates questions about whether we should prioritise privacy over accountability. This needs to be further discussed and researched. I aim to finish this by **end of February**.
    - Push notification using web sockets – When meetings are created, staff who are in the meeting roster will need to be notified, this will be done using push notification (via web sockets). I aim to finish this by **mid-February.**
    - Login system and permissions to only allow people to access only their own meetings – Login system will be done using DeeId (formerly OMNEE), which uses zero-knowledge proofs to authenticate. Objects will need to be sent to DeeID for signing using the user's private key which will be stored securely in the DeeID app. The DeeID app is currently in development (by Sirvan). I aim to finish this by **end of February.**
    - Implementation of auditing
    - Implementation of OTP codes for the purposes of attendance – An OTP code will be shown to users physically present in the room, which will be used to join the meeting.
- **Web App** – **by end of February**
    - Implementation of Login system
    - Implementation of meeting page
    - Implementation of creating poll page
    - Implementation of viewing results of polls
- **Mobile App – mid February to mid-March**
    - Implementation of new meeting notifications
    - Implementation of joining a meeting
    - Implementation of voting and leaving comments
    - Implementation of viewing results of polls
- **Testing and trailing the system – mid-March to end of March**
- **Finish write up – By mid-April**

F Code Listing

Please note that the full source code is very big, thus I have only included the parts that are interesting. The full source code can be obtained from the GitHub link provided in the System Manual.

F.1 JSON Schema

F.1.1 Events Schema

```
{
  "definitions": {},
  "$schema": "http://json-schema.org/draft-07/schema#",
  "$id": "http://ucl.ac.uk/gosh/event_schema.json",
  "type": "object",
  "title": "Meeting Event Schema",
  "required": [
    "by",
    "meetingId",
    "_id",
    "timestamp",
    "type",
    "content"
  ],
  "properties": {
    "by": {
      "$id": "#/properties/by",
      "type": "string",
      "title": "By",
      "description": "Public key of sender"
    },
    "meetingId": {
      "$id": "#/properties/meetingId",
      "type": "string",
      "title": "Meeting Id"
    },
    "_id": {
      "$id": "#/properties/_id",
      "type": "string",
      "title": "Event ID",
      "description": "Signature of the event by the sender"
    },
    "timestamp": {
      "$id": "#/properties/timestamp",
      "type": "number",
      "title": "Timestamp",
      "description": "The Unix Timestamp of the event",
      "default": 0,
      "examples": [
        "1552875911"
      ]
    },
    "type": {
      "$id": "#/properties/type",
      "type": "string",
      "enum": [
        "start",
        "join",
        "leave",
```

```
        "poll",
        "pollEnd",
        "vote",
        "comment",
        "reply",
        "discussion",
        "disagreement",
        "patientDataChange",
        "ack",
        "joinAck",
        "pollEndAck",
        "ackError",
        "ackEnd",
        "end"
      ],
      "title": "EventType",
      "description": "The type of event",
      "examples": [
        "start"
      ]
    },
    "refEvent": {
      "$id": "#/properties/refEvent",
      "type": "string",
      "title": "The Reference Event",
      "description": "All events apart from start must refer a previous event"
    },
    "content": {
      "$id": "#/properties/content",
      "type": "object",
      "title": "The Content"
    }
  }
}
```

F.1.2 Comment content Schema

```
{
  "definitions": {},
  "$schema": "http://ucl.ac.uk/gosh/event_content_comment_schema#",
  "$id": "http://ucl.ac.uk/gosh/event_content_comment_schema.json",
  "type": "object",
  "title": "The Comment Content Schema",
  "properties": {
    "patient": {
      "$id": "#/properties/patient",
      "type": "string",
      "title": "The Patient ID",
      "description": "The patient for which this comment applies [optional]"
    },
    "comment": {
      "$id": "#/properties/comment",
      "type": "string",
      "title": "The Comment"
    }
  },
  "required": [
    "comment"
  ]
}
```

F.1.3 Discussion content Schema

```
{
  "definitions": {},
  "$schema": "http://ucl.ac.uk/gosh/event_content_discussion_schema#",
  "$id": "http://ucl.ac.uk/gosh/event_content_discussion_schema.json",
  "type": "object",
  "title": "The Discussion Content Schema",
  "properties": {
    "patient": {
      "$id": "#/properties/patient",
      "type": "string",
      "title": "The Patient ID",
      "description": "The ID of the patient being discussed"
    }
  },
  "required": [
    "patient"
  ]
}
```

F.1.4 Join content Schema

```
{
  "definitions": {},
  "$schema": "http://ucl.ac.uk/gosh/event_content_join_schema#",
  "$id": "http://ucl.ac.uk/gosh/event_content_join_schema.json",
  "type": "object",
  "title": "The Join Content Schema",
  "required": [
    "otp"
  ],
  "properties": {
    "otp": {
      "$id": "#/properties/otp",
      "type": "string",
      "title": "OTP",
      "examples": [
        "4256"
      ]
    }
  }
}
```

F.1.5 Poll content Schema

```
{
  "definitions": {},
  "$schema": "http://ucl.ac.uk/gosh/event_content_poll_schema#",
  "$id": "http://ucl.ac.uk/gosh/event_content_poll_schema.json",
  "type": "object",
  "title": "The Poll Content Schema",
  "required": [
    "question",
    "options"
  ],
  "properties": {
    "patient": {
      "$id": "#/properties/patient",
      "type": "string",
```

```
      "title": "Patient ID"
    },
    "question": {
      "$id": "#/properties/question",
      "type": "string",
      "title": "The Question",
      "description": "The question of the poll"
    },
    "options": {
      "$id": "#/properties/options",
      "type": "array",
      "title": "Options for the poll",
      "default": null,
      "minItems": 2
    }
  }
}
```

F.1.6 Reply content Schema

```
{
  "definitions": {},
  "$schema": "http://ucl.ac.uk/gosh/event_content_reply_schema#",
  "$id": "http://ucl.ac.uk/gosh/event_content_reply_schema.json",
  "type": "object",
  "title": "The Reply Content Schema",
  "properties": {
    "reply": {
      "$id": "#/properties/reply",
      "type": "string",
      "title": "The Reply to a comment"
    }
  },
  "required": [
    "reply"
  ]
}
```

F.1.7 Start content Schema

```
{
  "definitions": {},
  "$schema": "http://ucl.ac.uk/gosh/event_content_start_schema#",
  "$id": "http://ucl.ac.uk/gosh/event_content_start_schema.json",
  "type": "object",
  "title": "The Start Content Schema",
  "default": null,
  "required": [
    "otp",
    "meeting"
  ],
  "properties": {
    "otp": {
      "$id": "#/properties/otp",
      "type": "string",
      "title": "OTP",
      "examples": [
        "4256"
      ]
    },
    "meeting": {
```

```
      "$id": "#/properties/meeting",
      "type": "object",
      "title": "The Content"
    }
  }
}
```

F.1.8 Vote content Schema

```
{
  "definitions": {},
  "$schema": "http://ucl.ac.uk/gosh/event_content_vote_schema#",
  "$id": "http://ucl.ac.uk/gosh/event_content_vote_schema.json",
  "type": "object",
  "title": "The Vote Content Schema",
  "required": [
    "vote"
  ],
  "properties": {
    "vote": {
      "$id": "#/properties/vote",
      "type": "string",
      "title": "Vote",
      "description": "The vote that is being casted"
    }
  }
}
```

F.2 Meeting contract

```
pragma solidity ^0.5.0;


contract MeetingContract {
    struct Meeting {
        string id;
        mapping (address => bool) participants;
        string eventStartHash;
        string eventEndHash;
        mapping (address => bool) approvals;
    }

    address owner;
    bool eventsPopulated = false;

    Meeting public meeting;

     constructor (string memory id, address[] memory participants) public {
        meeting.id = id;
        for (uint i=0; i< participants.length; i++) {
            meeting.participants[participants[i]] = true;
        }
        owner = msg.sender;
    }

    modifier onlyOwner(){
            require(msg.sender == owner);
            _;
    }

```

```
31      modifier onlyParticipant(){
32              require(meeting.participants[msg.sender]);
33              _;
34      }
35
36      modifier eventsNotPopulated(){
37              require(!eventsPopulated);
38              _;
39      }
40
41      function getOwner() view public returns(address) {
42          return owner;
43      }
44
45      function getMeetingId() view public returns(string memory) {
46          return (meeting.id);
47      }
48
49      function getEvents() view public returns(string memory, string memory) {
50          return (meeting.eventStartHash, meeting.eventEndHash);
51      }
52
53      function setEvents(string memory start, string memory end) public onlyOwner
            eventsNotPopulated {
54          meeting.eventStartHash = start;
55          meeting.eventEndHash = end;
56          eventsPopulated = true;
57      }
58
59      function approve() public onlyParticipant{
60          meeting.approvals[msg.sender] = true;
61      }
62
63      function isApproved() view public returns(bool){
64          return meeting.approvals[msg.sender];
65      }
66  }
```

F.3 Meeting Server

```
1   import json
2   import time
3   from flask_socketio import close_room
4   from flask_socketio import emit as emit_ws
5   from flask_socketio import join_room, leave_room
6   from flask import Flask
7   from flask_cors import CORS
8   from flask_socketio import SocketIO
9   from enum import Enum
10  import traceback
11
12  import meeting_contract_helper
13  from db import Database
14  import config
15  from authorization import Role, get_role
16  import authentication
17  import copy
18  import event_schema_validator
19  import log as logger
20  import logging
```

```
21 from model import Event, Staff, Meeting, AckErrorContent, EventError, JoinContent,
       DeeIdLoginSigSigned, \
22     MeetingEventType, \
23     StartContent, AckJoinContent, AckEndContent, PollContent, VoteContent,
           PDCContent, AckPollEndContent
24
25 app = Flask(__name__)
26
27 app.config['SECRET_KEY'] = 'TVBam9S&W7IbTC8W'
28 socketio = SocketIO(app)
29 CORS(app)
30 config = config.get_config()
31 db = Database(config["database"]["db_name"], config["database"]["ip"], config["
       database"]["port"])
32 log = logger.get_logger('web_server_socketio')
33 timestamp_tolerance = 10  # seconds
34 ongoing_meetings = {}
35 auth = authentication.Auth(config)
36 smart_contract = meeting_contract_helper.MeetingContractHelper(config)
37
38
39 class OngoingMeeting:
40     def __init__(self):
41         self.otp = ''
42         self.host = ''
43         self.start_event = None
44         self.events = {}  # type: dict[str, Event]
45         self.polls = {}  # type: dict[str, Poll]
46         self.unref_events = {}  # type: dict[str, Event]
47         self.session_keys = {}  # type: dict[str, str]
48         self.latest_join_events = {}  # type: dict[str, Event]
49
50
51 class Poll:
52     def __init__(self):
53         self.votes = {}  # type: dict[str, Event]
54
55
56 def start(event, staff, meeting, roles) -> (bool, dict):
57     """
58     Handles the START event
59     :return: returns a tuple of (bool, dict). The bool is if the execution was ok,
60     dict returns the error event in case the execution failed
61     """
62     log.debug("Processing Start event")
63     # Check if the user is allowed to start
64     if Role.HOST not in roles:
65         return False, get_error_ack(event.id, event.meeting_id,
66                                     content=AckErrorContent(error_code=EventError.
                                           UNAUTHORISED,
67                                                             details=''))
68
69     # Check if meeting has already started, if it has send the otp, in case the
           host has forgotten
70     started, _ = validate_meeting_started(event, meeting)
71     if started:
72         otp = ongoing_meetings[meeting.id].otp
73         return False, get_error_ack(event.id, event.meeting_id,
74                                     content=AckErrorContent(error_code=EventError.
                                           MEETING_ALREADY_STARTED,
75                                                             details='Meeting
```

```
                                                    already started,
                                                    otp : ' + otp))

    # Parse the content of event as StartContent
    try:
        sc = StartContent.parse(event.content)
        dee_id_login_sig = DeeIdLoginSigSigned.parse(sc.deeid_login_sig_signed)
        new_key = sc.key
        uid = dee_id_login_sig.uid
        expiry = dee_id_login_sig.expiry_time
        sig = dee_id_login_sig.signature
    except KeyError as ke:
        log.error(ke)
        traceback.print_tb(ke.__traceback__)
        return False, get_error_ack(event.id, event.meeting_id,
                                    content=AckErrorContent(error_code=EventError.
                                        MALFORMED_EVENT,
                                                            details="Content
                                                                doesnt have
                                                                sufficient values"
                                                                ))

    # Authenticate new key
    msg = uid + staff.id + expiry + new_key
    addr = auth.get_sig_address_from_signature(msg=msg, signature=sig)
    ok = auth.ethkey_in_deeid_contract(addr, staff.id)
    if not ok:
        return False, get_error_ack(event.id, event.meeting_id,
                                    content=AckErrorContent(error_code=EventError.
                                        UNAUTHORISED,
                                                            details="New pubKey
                                                                cannot be
                                                                authenticated"))
    print("OTP: ", sc.otp)
    #  Create new meeting session
    om = OngoingMeeting()
    om.otp = sc.otp
    om.host = staff.id
    om.start_event = event
    om.events = {event.id: event}
    om.polls = {}
    om.unref_events = {}
    om.session_keys = {
        staff.id: new_key
    }
    om.latest_join_events = []

    # Create smart contract for the meeting  - COSTS MONEY
    log.debug("Deploying Smart Contract...")
    try:
        meeting.contract_id = smart_contract.new_meeting_contract(meeting)
    except Exception as e:  # We catch all exceptions as there are too many
        log.error("FAILED deploying smart contract")
        log.error(e)
        traceback.print_tb(e.__traceback__)
        return False, get_error_ack(event.id, event.meeting_id,
                                    content=AckErrorContent(error_code=EventError.
                                        INTERNAL_ERROR,
                                                            details='Smart
                                                                Contract could not
                                                                 be deployed'))
```

```
125
126         log.debug("Deployed smart contract!")
127         log.debug(meeting.contract_id)
128         ongoing_meetings[meeting.id] = om
129
130         meeting.started = True  # Set the meeting as started in the db
131         meeting.start_event_id = event.id  # Set the start event
132         db.update_meeting(meeting.id, meeting.to_json_dict())
133
134         # ACK
135         ack = get_ack(event.id, event.meeting_id)
136         return True, ack
137
138
139 def join(event, staff, meeting, roles) -> (bool, dict):
140         """
141         Handles the JOIN event
142         :return: returns a tuple of (bool, dict). The bool is if the execution was ok,
143         dict returns the error event in case the execution failed
144         """
145         log.debug("Processing Join event")
146         # Check if the user is allowed to join
147         if Role.PARTICIPANT not in roles:
148             return False, get_error_ack(event.id, event.meeting_id,
149                                         content=AckErrorContent(error_code=EventError.
                                                UNAUTHORISED,
150                                                                 details='')))
151
152         # Check if meeting has started
153         ok, err_ack = validate_meeting_started(event, meeting)
154         if not ok:
155             return False, err_ack
156
157         # Get Meeting session details
158         meeting_session_details = ongoing_meetings.get(meeting.id, None)  # type:
                OngoingMeeting
159
160         # Check if ref event is a start event
161         if event.ref_event != meeting_session_details.start_event.id:
162             return False, get_error_ack(event.id, event.meeting_id,
163                                         content=AckErrorContent(error_code=EventError.
                                                INVALID_REF_EVENT,
164                                                                 details='Ref Event
                                                                        must be the Start
                                                                        event'))
165
166         # Parse the content of event as JoinContent
167         try:
168             jc = JoinContent.parse(event.content)
169             dee_id_login_sig = DeeIdLoginSigSigned.parse(jc.deeid_login_sig_signed)
170             new_key = jc.key
171             uid = dee_id_login_sig.uid
172             expiry = dee_id_login_sig.expiry_time
173             sig = dee_id_login_sig.signature
174         except KeyError as ke:
175             log.error(ke)
176             traceback.print_tb(ke.__traceback__)
177             return False, get_error_ack(event.id, event.meeting_id,
178                                         content=AckErrorContent(error_code=EventError.
                                                MALFORMED_EVENT,
179                                                                 details="Content
```

```
                                                                    doesnt have
                                                                    sufficient values"
                                                                    ))
180
181     # Check OTP
182     if not jc.otp == meeting_session_details.otp:
183         return False, get_error_ack(event.id, event.meeting_id,
184                                     content=AckErrorContent(error_code=EventError.
                                            BAD_OTP,
185                                                             details=''))
186
187     # Authenticate new key
188     msg = uid + staff.id + expiry + new_key
189     addr = auth.get_sig_address_from_signature(msg=msg, signature=sig)
190     ok = auth.ethkey_in_deeid_contract(addr, staff.id)
191
192     if not ok:
193         return False, get_error_ack(event.id, event.meeting_id,
194                                     content=AckErrorContent(error_code=EventError.
                                            UNAUTHORISED,
195                                                             details="New pubKey
                                                                cannot be
                                                                authenticated"))
196
197     meeting_session_details.session_keys[staff.id] = new_key
198
199     # Add them to the list of staff that have joined the meeting
200     update_attended_staff(meeting, staff)
201
202     # Join the socketio room using the sessison_id
203     join_room(meeting.id)
204
205     start_event = meeting_session_details.start_event
206     latest_join_events = meeting_session_details.latest_join_events
207
208     # ACK
209     jac = AckJoinContent(start_event, latest_join_events)
210     ack = get_ack(event.id, event.meeting_id, type=MeetingEventType.ACK_JOIN,
            content=jac)
211     return True, ack
212
213
214 def leave(event, staff, meeting):
215     """
216     Handles the LEAVE event
217     :return: returns a tuple of (bool, dict). The bool is if the execution was ok,
218     dict returns the error event in case the execution failed
219     """
220     log.debug("Processing Leave event")
221     # Check if meeting actually started
222     ok, err_ack = validate_meeting_started(event, meeting)
223     if not ok:
224         return False, err_ack
225
226     # Check if they have actually joined
227     ok, err_ack = validate_join(event, staff, meeting)
228     if not ok:
229         return False, err_ack
230
231     # Check of ref event is correct
232     ok, err_ack = validate_ref_event(event, meeting)
```

```
233     if not ok:
234         return False, err_ack
235 
236     leave_room(meeting.id)
237 
238     # ACK
239     ack = get_ack(event.id, event.meeting_id)
240     return True, ack
241 
242 
243 def end(event, staff, meeting, roles):
244     """
245     Handles the END event
246     :return: returns a tuple of (bool, dict). The bool is if the execution was ok,
247     dict returns the error event in case the execution failed
248     """
249     log.debug("Processing End event")
250     # Check whether they should be allowed to end
251     if Role.HOST not in roles:
252         return False, get_error_ack(event.id, event.meeting_id,
253                                     content=AckErrorContent(error_code=EventError.
                                            UNAUTHORISED,
254                                                             details=''))
255 
256     # Check if meeting actually started
257     ok, err_ack = validate_meeting_started(event, meeting)
258     if not ok:
259         return False, err_ack
260 
261     # Check if they have actually joined
262     ok, err_ack = validate_join(event, staff, meeting)
263     if not ok:
264         return False, err_ack
265 
266     # Check if ref event is correct
267     ok, err_ack = validate_ref_event(event, meeting)
268     if not ok:
269         return False, err_ack
270 
271     # Get Meeting session details
272     meeting_session_details = ongoing_meetings.get(meeting.id, None)  # type:
            OngoingMeeting
273 
274     # ACK  -- WE get ACK early as we need the signature for smart contract later,
            and there could be errors with that
275     eac = AckEndContent(list(meeting_session_details.unref_events.keys()))
276     ack = get_ack(event.id, event.meeting_id, type=MeetingEventType.ACK_END,
            content=eac)
277     return True, ack
278 
279 
280 def poll(event, staff, meeting, roles):
281     """
282     Handles the POLL event
283     :return: returns a tuple of (bool, dict). The bool is if the execution was ok,
284     dict returns the error event in case the execution failed
285     """
286     log.debug("Processing Poll event")
287     # Check whether they should be allowed to start a poll
288     if Role.HOST not in roles:
289         return False, get_error_ack(event.id, event.meeting_id,
```

```
290                                     content=AckErrorContent(error_code=EventError.
                                            UNAUTHORISED,
291                                                                 details=''))
292
293     # Check if meeting actually started
294     ok, err_ack = validate_meeting_started(event, meeting)
295     if not ok:
296         return False, err_ack
297
298     # Check if they have actually joined
299     ok, err_ack = validate_join(event, staff, meeting)
300     if not ok:
301         return False, err_ack
302
303     # Check of ref event is correct
304     ok, err_ack = validate_ref_event(event, meeting)
305     if not ok:
306         return False, err_ack
307
308     # Get Meeting session details
309     meeting_session_details = ongoing_meetings.get(meeting.id, None)  # type:
            OngoingMeeting
310
311     # Parsing content as PollContent
312     try:
313         pc = PollContent.parse(event.content)
314     except KeyError as ke:
315         log.error(ke)
316         traceback.print_tb(ke.__traceback__)
317         return False, get_error_ack(event.id, event.meeting_id,
318                                     content=AckErrorContent(error_code=EventError.
                                            MALFORMED_EVENT,
319                                                             details="Content
                                                                doesnt have
                                                                sufficient values"
                                                                ))
320
321     new_poll = Poll()
322     poll.votes = {}
323
324     meeting_session_details.polls[event.id] = new_poll
325
326     # ACK
327     ack = get_ack(event.id, event.meeting_id)
328     return True, ack
329
330
331 def vote(event, staff, meeting, roles):
332     """
333     Handles the VOTE event
334     :return: returns a tuple of (bool, dict). The bool is if the execution was ok,
335     dict returns the error event in case the execution failed
336     """
337     log.debug("Processing Vote Event")
338     # Check whether they should be allowed to vote
339     if Role.PARTICIPANT not in roles:
340         return False, get_error_ack(event.id, event.meeting_id,
341                                     content=AckErrorContent(error_code=EventError.
                                            UNAUTHORISED,
342                                                             details=''))
343
```

```
344     # Check if meeting actually started
345     ok, err_ack = validate_meeting_started(event, meeting)
346     if not ok:
347         return False, err_ack
348 
349     # Check if they have actually joined
350     ok, err_ack = validate_join(event, staff, meeting)
351     if not ok:
352         return False, err_ack
353 
354     # Get Meeting session details
355     meeting_session_details = ongoing_meetings.get(meeting.id, None)  # type:
            OngoingMeeting
356 
357     # Check if refEvent is a poll
358     poll = meeting_session_details.polls.get(event.ref_event, None)
359     if poll is None:
360         return False, get_error_ack(event.id, event.meeting_id,
361                                     content=AckErrorContent(error_code=EventError.
                                        POLL_NOT_FOUND,
362                                                             details=''))
363 
364     # Parsing content as VoteContent
365     try:
366         vc = VoteContent.parse(event.content)
367     except KeyError as ke:
368         traceback.print_tb(ke.__traceback__)
369         log.error(ke)
370         return False, get_error_ack(event.id, event.meeting_id,
371                                     content=AckErrorContent(error_code=EventError.
                                        MALFORMED_EVENT,
372                                                             details="Content
                                                                doesnt have
                                                                sufficient values"
                                                                ))
373 
374     # Check if already voted
375     if event.by in poll.votes:
376         return False, get_error_ack(event.id, event.meeting_id,
377                                     content=AckErrorContent(error_code=EventError.
                                        ALREADY_VOTED,
378                                                             details=''))
379 
380     log.debug("Adding the vote to votes")
381     poll.votes[staff.id] = event
382 
383     # ACK
384     ack = get_ack(event.id, event.meeting_id)
385     return True, ack
386 
387 
388 def end_poll(event, staff, meeting, roles):
389     """
390     Handles END_POLL event
391     :return: returns a tuple of (bool, dict). The bool is if the execution was ok,
392     dict returns the error event in case the execution failed
393     """
394     log.debug("Processing End Poll event")
395     # Check whether they should be allowed to end poll
396     if Role.HOST not in roles:
397         return False, get_error_ack(event.id, event.meeting_id,
```

```
                                    content=AckErrorContent(error_code=EventError.
                                        UNAUTHORISED,
                                                            details=''))

    # Check if meeting actually started
    ok, err_ack = validate_meeting_started(event, meeting)
    if not ok:
        return False, err_ack

    # Check if they have actually joined
    ok, err_ack = validate_join(event, staff, meeting)
    if not ok:
        return False, err_ack

    # Get Meeting session details
    meeting_session_details = ongoing_meetings.get(meeting.id, None)  # type:
        OngoingMeeting

    #  Check if refEvent is a poll
    poll = meeting_session_details.polls.get(event.ref_event, None)
    if poll is None:
        return False, get_error_ack(event.id, event.meeting_id,
                                    content=AckErrorContent(error_code=EventError.
                                        POLL_NOT_FOUND,
                                                            details=''))

    vote_event_ids = [event.id for event in list(poll.votes.values())]
    poll_end_ack_event = AckPollEndContent(votes=vote_event_ids)

    # ACK
    ack = get_ack(event.id, meeting.id,
                  type=MeetingEventType.ACK_POLL_END, content=poll_end_ack_event)

    # Delete poll
    meeting_session_details.polls.pop(event.ref_event)

    return True, ack


def comment_reply_disagreement(event, staff, meeting, roles):
    """
    Handles COMMENT and REPLY and DISAGREEMENT event as they have the same
        protocol
    :return: returns a tuple of (bool, dict). The bool is if the execution was ok,
    dict returns the error event in case the execution failed
    """
    log.debug("Processing Comment/Reply/Disagreement event")
    # Check whether they should be allowed to vote
    if Role.PARTICIPANT not in roles:
        return False, get_error_ack(event.id, event.meeting_id,
                                    content=AckErrorContent(error_code=EventError.
                                        UNAUTHORISED,
                                                            details=''))

    # Check if meeting actually started
    ok, err_ack = validate_meeting_started(event, meeting)
    if not ok:
        return False, err_ack

    # Check if they have actually joined
    ok, err_ack = validate_join(event, staff, meeting)
```

```
454     if not ok:
455         return False, err_ack
456 
457     # Check of ref event is correct
458     ok, err_ack = validate_ref_event(event, meeting)
459     if not ok:
460         return False, err_ack
461 
462     # ACK
463     ack = get_ack(event.id, meeting.id)
464     return True, ack
465 
466 
467 def discussion(event, staff, meeting, roles):
468     """
469     Handles the DISCUSSION event
470     :return: returns a tuple of (bool, dict). The bool is if the execution was ok,
471     dict returns the error event in case the execution failed
472     """
473     log.debug("Processing Discussion event")
474     # Check whether they should be allowed to start discussion
475     if Role.HOST not in roles:
476         return False, get_error_ack(event.id, event.meeting_id,
477                                     content=AckErrorContent(error_code=EventError.
                                         UNAUTHORISED,
478                                                             details=''))
479 
480     # Check if meeting actually started
481     ok, err_ack = validate_meeting_started(event, meeting)
482     if not ok:
483         return False, err_ack
484 
485     # Check if they have actually joined
486     ok, err_ack = validate_join(event, staff, meeting)
487     if not ok:
488         return False, err_ack
489 
490     # Check of ref event is correct
491     ok, err_ack = validate_ref_event(event, meeting)
492     if not ok:
493         return False, err_ack
494 
495     # ACK
496     ack = get_ack(event.id, meeting.id)
497     return True, ack
498 
499 
500 def patient_data_change(event, staff, meeting, roles):
501     """
502     Handles PATIENT_DATA_CHANGE event
503     :return: returns a tuple of (bool, dict). The bool is if the execution was ok,
504     dict returns the error event in case the execution failed
505     """
506     log.debug("Processing PDC event")
507     # Check whether they should be allowed to start discussion
508     if Role.HOST not in roles:
509         return False, get_error_ack(event.id, event.meeting_id,
510                                     content=AckErrorContent(error_code=EventError.
                                         UNAUTHORISED,
511                                                             details=''))
512 
```

```
513     # Check if meeting actually started
514     ok, err_ack = validate_meeting_started(event, meeting)
515     if not ok:
516         return False, err_ack
517 
518     # Check if they have actually joined
519     ok, err_ack = validate_join(event, staff, meeting)
520     if not ok:
521         return False, err_ack
522 
523     # Check of ref event is correct
524     ok, err_ack = validate_ref_event(event, meeting)
525     if not ok:
526         return False, err_ack
527 
528     # Parsing content as VoteContent
529     try:
530         pdc = PDCContent.parse(event.content)
531     except KeyError as ke:
532         traceback.print_tb(ke.__traceback__)
533         log.error(ke)
534         return False, get_error_ack(event.id, event.meeting_id,
535                                     content=AckErrorContent(error_code=EventError.
                                         MALFORMED_EVENT,
536                                                             details="Content
                                                                 doesnt have
                                                                 sufficient values"
                                                                 ))
537 
538     # Check if patient actually exists
539     patient_id = pdc.patient
540     patient = db.get_patient(patient_id)
541     if patient is None:
542         return False, get_error_ack(event.id, event.meeting_id,
543                                     content=AckErrorContent(error_code=EventError.
                                         PATIENT_NOT_FOUND,
544                                                             details=''))
545 
546     # ACK
547     ack = get_ack(event.id, meeting.id)
548     return True, ack
549 
550 
551 def sign(event: Event):
552     """
553     signs an event
554     :return: signed Event
555     """
556     signed_event = auth.sign_event(event.to_json_dict())
557     return Event.parse(signed_event)
558 
559 
560 def send(signed_event, broadcast_room=None):
561     """
562     Broadcasts an event on a room
563     """
564     if broadcast_room is None:
565         emit_ws('room-message', signed_event)
566     else:
567         emit_ws('room-message', signed_event, room=broadcast_room)
568 
```

```
569
570 def update_attended_staff(meeting: Meeting, staff: Staff):
571     """
572     Updates the Database with staff members who have attended the meeting
573     """
574     if staff.id not in meeting.attended_staff:
575         meeting.attended_staff.append(staff.id)
576         db.update_meeting(meeting.id, meeting.to_json_dict())
577
578
579 def record(event: Event, meeting: Meeting):
580     """
581     Records an event into the database
582     """
583     log.debug("Storing: " + event.id)
584     log.debug(event)
585     meeting_session_details = ongoing_meetings.get(meeting.id, None)  # type:
        OngoingMeeting
586     meeting_session_details.events[event.id] = event
587     db.insert_event(event.to_json_dict())
588
589
590 def get_error_ack(ref_event, meeting_id, content=None):
591     """
592     Helper function to get a pre populated ACK_ERR Event
593     :return: an Event with type as ACK_ERR
594     """
595     ack_event = Event()
596     ack_event.type = MeetingEventType.ACK_ERR
597     ack_event.meeting_id = meeting_id
598     ack_event.ref_event = ref_event
599     ack_event.timestamp = int(time.time())
600
601     if content is None:
602         content = {}
603     ack_event.content = content
604     return ack_event
605
606
607 def get_ack(ref_event, meeting_id, type=MeetingEventType.ACK, content=None):
608     """
609     Helper function to get a pre populated ACK Event
610     :return: an Event with type as ACK
611     """
612     ack_event = Event()
613     ack_event.type = type
614     ack_event.meeting_id = meeting_id
615     ack_event.ref_event = ref_event
616     ack_event.timestamp = int(time.time())
617
618     if content is None:
619         content = {}
620     ack_event.content = content
621     return ack_event
622
623
624 def add_event_as_unref(meeting: Meeting, event: Event):
625     """
626     Appends the set of unreferenced events
627     """
628     meeting_session_details = ongoing_meetings.get(meeting.id, None)  # type:
```

```
        OngoingMeeting
    meeting_session_details.unref_events[event.id] = None


def check_and_remove_ref_event(meeting, ref_event):
    """
    Checks if the event exists, and removes it from the set of unreferenced events
    """
    meeting_session_details = ongoing_meetings.get(meeting.id, None)  # type:
        OngoingMeeting
    meeting_session_details.unref_events.pop(ref_event, None)


def validate_ref_event(event, meeting) -> (bool, dict):
    """
    Validates a reference event (checks if it exists)
    :return: returns a tuple of (bool, dict). The bool is if the execution was ok,
    dict returns the error event in case the execution failed
    """
    meeting_session_details = ongoing_meetings.get(meeting.id, None)  # type:
        OngoingMeeting
    ref_valid = event.ref_event in meeting_session_details.events

    if not ref_valid:
        err_ack = get_error_ack(event.id, event.meeting_id,
                                content=AckErrorContent(error_code=EventError.
                                    INVALID_REF_EVENT,
                                                        details=''))
        return False, err_ack
    return True, None


def validate_meeting_started(event: Event, meeting: Meeting) -> (bool, dict):
    """
    Validates if a meeting has started or not
    :return: returns a tuple of (bool, dict). The bool is if the execution was ok,
    dict returns the error event in case the execution failed
    """
    meeting_started = meeting.id in ongoing_meetings
    if not meeting_started:
        err_ack = get_error_ack(event.id, event.meeting_id,
                                content=AckErrorContent(error_code=EventError.
                                    MEETING_NOT_STARTED,
                                                        details=''))
        return False, err_ack

    return True, None


def validate_join(event, staff, meeting) -> (bool, dict):
    """
    Validates if a staff has joined the meeting or not
    :return: returns a tuple of (bool, dict). The bool is if the execution was ok,
    dict returns the error event in case the execution failed
    """
    joined = staff.id in meeting.attended_staff
    if not joined:
        err_ack = get_error_ack(event.id, event.meeting_id,
                                content=AckErrorContent(error_code=EventError.
                                    MEETING_NOT_JOINED,
                                                        details=''))
```

```
684         return False, err_ack
685
686     return True, None
687
688
689 def validate_timestamp(event) -> (bool, dict):
690     """
691     Validates the timestamp of an event
692     :return: returns a tuple of (bool, dict). The bool is if the execution was ok,
693     dict returns the error event in case the execution failed
694     """
695     current_time = int(time.time())
696     if not (current_time - timestamp_tolerance <= event.timestamp <= current_time
            + timestamp_tolerance):
697         err_event = get_error_ack(event.id, event.meeting_id,
698                                   content=AckErrorContent(error_code=EventError.
                                      TIMESTAMP_NOT_SYNC,
699                                                           details='Server
                                                              Timestamp : ' + str(
                                                              current_time)))
700
701         return False, err_event
702
703     return True, None
704
705
706 def validate_schema(event_dict) -> (bool, dict):
707     """
708     Validates the schema using JSON Schema
709     :return: returns a tuple of (bool, dict). The bool is if the execution was ok,
710     dict returns the error event in case the execution failed
711     """
712     ok, err = event_schema_validator.validate(event_dict)
713     if not ok:
714         err_event = get_error_ack("unknown", "unknown",
715                                   content=AckErrorContent(error_code=EventError.
                                      MALFORMED_EVENT,
716                                                           details=err))
717
718         return False, err_event
719
720     return True, None
721
722
723 def validate_signature(event, staff, meeting, check_contract=False) -> (bool, dict
        ):
724     """
725     Validates the signature of an event
726     :return: returns a tuple of (bool, dict). The bool is if the execution was ok,
727     dict returns the error event in case the execution failed
728     """
729     log.debug('Validating Signature...')
730
731     sig_addr = str(auth.get_sig_address_from_event(event.to_json_dict()))
732
733     #  Check session key, or if it doesnt exist, check with actual public key in
            smart contract
734     ok = False
735     if check_contract:
736         log.debug('Checking DeeId Contract')
737         ok = auth.ethkey_in_deeid_contract(sig_addr, staff.id)
```

```
738         log.debug('Eth key in contract? ' + str(ok))
739     else:
740         # Get Meeting session details to get the session key
741         meeting_session_details = ongoing_meetings.get(meeting.id, None)  # type:
                OngoingMeeting
742         meeting_session_key = None
743         if meeting_session_details is not None:
744             meeting_session_key = meeting_session_details.session_keys.get(staff.
                    id, None)
745 
746         if meeting_session_key is None:
747             err_event = get_error_ack(event.id, event.meeting_id,
748                                       content=AckErrorContent(error_code=
                                              EventError.BAD_SESSION_KEY_SIGNATURE,
749                                                               details=''))
750             return False, err_event
751 
752         ok = sig_addr.lower() == meeting_session_key.lower()
753         log.debug('Sig Addr ' + sig_addr)
754         log.debug('Session key ' + meeting_session_key)
755         log.debug('Session key matches up? ' + str(ok))
756 
757     if not ok:
758         err_event = get_error_ack(event.id, event.meeting_id,
759                                   content=AckErrorContent(error_code=EventError.
                                          BAD_SIGNATURE,
760                                                           details=''))
761 
762         return False, err_event
763 
764     return True, None
765 
766 
767 def validate_preliminary_authority(event, roles) -> (bool, dict):
768     """
769     Does preliminary checks to see if the user is authorised to post the event
770     :return: returns a tuple of (bool, dict). The bool is if the execution was ok,
771     dict returns the error event in case the execution failed
772     """
773     if not roles:
774         err_event = get_error_ack(event.id, event.meeting_id,
775                                   content=AckErrorContent(error_code=EventError.
                                          UNAUTHORISED,
776                                                           details=''))
777 
778         return False, err_event
779     return True, None
780 
781 
782 def end_meeting_session(meeting, event, signed_ack_event):
783     """
784     Gracefully ends a meeting by populating smart contract and the database
785     """
786     # Write the event hash of the meeting to smart contract  - COSTS MONEY
787     meeting_session_details = ongoing_meetings.get(meeting.id, None)  # type:
            OngoingMeeting
788     try:
789         start_hash = meeting_session_details.start_event.id
790         end_hash = signed_ack_event.id
791         smart_contract.set_event_hash(meeting, start_hash, end_hash)
792     except Exception as e:  # We catch all exceptions as there are too many
```

```
793         log.error(e)
794         traceback.print_tb(e.__traceback__)
795         return False, get_error_ack(event.id, event.meeting_id,
796                                     content=AckErrorContent(error_code=EventError.
                                        INTERNAL_ERROR,
797                                                             details='')) 
798 
799     # Mark meeting as ended
800     meeting.ended = True
801     db.update_meeting(meeting.id, meeting.to_json_dict())
802 
803     print('============= MEETING END =============')
804     print(ongoing_meetings[meeting.id])
805     print(signed_ack_event)
806     ongoing_meetings.pop(meeting.id)
807     close_room(meeting.id)
808 
809 
810 @socketio.on('room-message')
811 def room_message(event_string):
812     """
813     Main function that handles all events that enter through the web socket
814     """
815     # Parse json as dict
816     try:
817         event_json = json.loads(event_string)
818     except ValueError as ve:
819         log.error("Error Parsing room-message as JSON!")
820         log.error(ve)
821         traceback.print_tb(ve.__traceback__)
822         return json.dumps(sign(get_error_ack("unknown", "unknown",
823                                              content=AckErrorContent(error_code=
                                                 EventError.MALFORMED_EVENT,
824                                                                      details='Cant
                                                                         parse
                                                                      message
                                                                      as JSON')
                                                                      )).
                                                                      to_json_dict
                                                                      ())
825 
826     # Validate JSON dict using schema
827     ok, err_event = validate_schema(event_json)
828     if not ok:
829         log.error("Sending error msg")
830         errormsg = json.dumps(sign(err_event).to_json_dict())
831         log.error("======== error_msg: " + errormsg)
832         return errormsg
833 
834     # Parse dict as event object
835     try:
836         event = Event.parse(event_json)
837     except KeyError as ve:
838         log.error("Error Parsing room-message!")
839         log.error(ve)
840         traceback.print_tb(ve.__traceback__)
841         return json.dumps(sign(get_error_ack("unknown", "unknown",
842                                              content=AckErrorContent(error_code=
                                                 EventError.MALFORMED_EVENT,
843                                                                      details='Cant
                                                                         parse
```

```
                                                                    message
                                                                    as Event'
                                                                    ))).
                                                                    to_json_dict
                                                                    ())
844
845     # Check timestamp
846     ok, err_event = validate_timestamp(event)
847     if not ok:
848         return json.dumps(sign(err_event).to_json_dict())
849
850     # Get the staff
851     try:
852         staff = Staff.parse(db.get_staff(event.by))
853     except KeyError as ke:
854         log.error(ke)
855         traceback.print_tb(ke.__traceback__)
856         return json.dumps(sign(get_error_ack(event.id, event.meeting_id,
857                                              content=AckErrorContent(error_code=
                                                     EventError.STAFF_NOT_FOUND,
858                                                                     details='')))
                                                                         .
                                                                         to_json_dict
                                                                         ())
859     except TypeError as te:
860         log.error(te)
861         traceback.print_tb(te.__traceback__)
862         return json.dumps(sign(get_error_ack(event.id, event.meeting_id,
863                                              content=AckErrorContent(error_code=
                                                     EventError.STAFF_NOT_FOUND,
864                                                                     details='')))
                                                                         .
                                                                         to_json_dict
                                                                         ())
865
866     # Get the meeting
867     try:
868         meeting = Meeting.parse(db.get_meeting(event.meeting_id))
869     except KeyError as ke:
870         log.error(ke)
871         traceback.print_tb(ke.__traceback__)
872         return json.dumps(sign(get_error_ack(event.id, event.meeting_id,
873                                              content=AckErrorContent(error_code=
                                                     EventError.MEETING_NOT_FOUND,
874                                                                     details='')))
                                                                         .
                                                                         to_json_dict
                                                                         ())
875
876     except TypeError as te:
877         log.error(te)
878         traceback.print_tb(te.__traceback__)
879         return json.dumps(sign(get_error_ack(event.id, event.meeting_id,
880                                              content=AckErrorContent(error_code=
                                                     EventError.MEETING_NOT_FOUND,
881                                                                     details='')))
                                                                         .
                                                                         to_json_dict
                                                                         ())
882
883     # Check signature, before trusting anything it says
```

```
884
885     if event.type in [MeetingEventType.JOIN, MeetingEventType.START]:  # We check
            contract for join and start
886         ok, err_event = validate_signature(event, staff, meeting, check_contract=
                True)
887         if not ok:
888             return json.dumps(sign(err_event).to_json_dict())
889     else:
890         ok, err_event = validate_signature(event, staff, meeting)
891         if not ok:
892             return json.dumps(sign(err_event).to_json_dict())
893
894     # Get the roles
895     roles = get_role(staff, meeting)
896
897     # A preliminary authority check to see if the user can make any statements
            about the meeting
898     ok, err_event = validate_preliminary_authority(event, roles)
899     if not ok:
900         return json.dumps(sign(err_event).to_json_dict())
901
902     # Get the event type
903     event_type = MeetingEventType(event.type)
904     ack_event = None
905     ok = False
906     end_meeting = False
907     send_privately = False
908
909     if event_type == MeetingEventType.START:
910         ok, ack_event = start(event, staff, meeting, roles)
911
912     if event_type == MeetingEventType.JOIN:
913         ok, ack_event = join(event, staff, meeting, roles)
914
915     if event_type == MeetingEventType.LEAVE:
916         ok, ack_event = leave(event, staff, meeting)
917
918     if event_type == MeetingEventType.POLL:
919         ok, ack_event = poll(event, staff, meeting, roles)
920
921     if event_type == MeetingEventType.VOTE:
922         ok, ack_event = vote(event, staff, meeting, roles)
923
924     if event_type == MeetingEventType.POLL_END:
925         ok, ack_event = end_poll(event, staff, meeting, roles)
926
927     if event_type == MeetingEventType.COMMENT or event_type == MeetingEventType.
            REPLY or event_type == MeetingEventType.DISAGREEMENT:
928         ok, ack_event = comment_reply_disagreement(event, staff, meeting, roles)
929
930     if event_type == MeetingEventType.DISCUSSION:
931         ok, ack_event = discussion(event, staff, meeting, roles)
932
933     if event_type == MeetingEventType.PATIENT_DATA_CHANGE:
934         ok, ack_event = patient_data_change(event, staff, meeting, roles)
935
936     if event_type == MeetingEventType.END:
937         ok, ack_event = end(event, staff, meeting, roles)
938         if ok:
939             end_meeting = True
940
```

```
    if not ok:  # If not ok we send the error ack event privately
        return json.dumps(sign(ack_event).to_json_dict())
    else:
        # If an event has been referenced
        check_and_remove_ref_event(meeting, event.ref_event)

        # Add ack and event to unreferenced events
        add_event_as_unref(meeting, event)
        add_event_as_unref(meeting, ack_event)

        # Sign the ack
        signed_ack_event = sign(ack_event)

        if not send_privately:  # Only Broadcast if the send_privately is set to
            False
            # Broadcast event
            send(json.dumps(event.to_json_dict()), broadcast_room=meeting.id)
            send(json.dumps(signed_ack_event.to_json_dict()), broadcast_room=
                meeting.id)

        record(event, meeting)
        record(signed_ack_event, meeting)

        if end_meeting:
            end_meeting_session(meeting, event, signed_ack_event)

        # Send the ack event to the user privately as well
        return json.dumps(signed_ack_event.to_json_dict())


if __name__ == '__main__':
    socketio.run(app, host="localhost", port=51235)
```

F.4 Meeting Contract Helper

```
from web3.auto import w3, Web3
import json

from model import Meeting
import log as logpy

log = logpy.get_logger('contract_helper')


class MeetingContractHelper:
    """
    Helper class to interface with the Meeting Contract
    """
    def __init__(self, config):
        self.config = config

        # Get the ETH Key
        self.auth_key_path = config["auth"]["key_path"]

        # Get Smart contract params
        self.block_chain_provider_url = config["smart_contract"]["bc_provider_url"
            ]
        self.compiled_contract_path = config["smart_contract"]["
            meeting_contract_path"]
        self.chain_id = config["smart_contract"]["chain_id"]
        self.max_gas = config["smart_contract"]["max_gas"]
```

```
25
26          # Get the private key from the keyfile
27          with open(self.auth_key_path) as keyfile:
28              encrypted_key = keyfile.read()
29              keyfilejson = json.loads(encrypted_key)
30              self.server_private_key = w3.eth.account.decrypt(encrypted_key, "leet"
                    ) # TODO: Ask user for passphrase
31              self.server_eth_address = keyfilejson['address']
32
33          # Get the Meeting contract ABI from the ABI file
34          with open(self.compiled_contract_path) as contract_file:
35              contract_file_data = contract_file.read()
36              contract_json = json.loads(contract_file_data)
37              self.contract_abi = contract_json['abi']
38              self.contract_bytecode = contract_json['bytecode']
39
40          self.w3 = Web3(Web3.HTTPProvider(self.block_chain_provider_url))
41
42
43      def new_meeting_contract(self, meeting : Meeting):
44          """
45          Deploys a new meeting contract
46          :param meeting: the meeting object associated with contract
47          :return: Ethereum address of the contract
48          """
49          contract = w3.eth.contract(abi=self.contract_abi, bytecode=self.
                contract_bytecode)
50          nonce = w3.eth.getTransactionCount(w3.toChecksumAddress('0x' + self.
                server_eth_address))
51          log.debug('Nonce: ' + str(nonce))
52          staff_dee_ids = [Web3.toChecksumAddress(staff_dee_id) for staff_dee_id in
                meeting.staff]
53          print(staff_dee_ids)
54          contract_txn = contract.constructor(meeting.id, staff_dee_ids).
                buildTransaction({
55              'nonce': nonce,
56              'chainId': self.chain_id,
57              'gas': 2000000
58          })
59          signed = w3.eth.account.signTransaction(contract_txn, private_key=self.
                server_private_key)
60          contract_txn_hash = w3.eth.sendRawTransaction(signed.rawTransaction)
61          log.debug('Waiting for contract to be mined...')
62          tx_receipt = w3.eth.waitForTransactionReceipt(contract_txn_hash)
63          return str(tx_receipt.contractAddress)
64
65
66      def set_event_hash(self, meeting : Meeting, start_event_hash : str,
            end_event_hash : str):
67          """
68          Sets the Event Hash in the Meeting Contract
69          :param meeting: the Meeting object associated with the contract
70          :param start_event_hash: the id of START event
71          :param end_event_hash: the id of ACK_END event
72          :return: the TX reciept
73          """
74          mdt_meeting = w3.eth.contract(
75              address=meeting.contract_id,
76              abi=self.contract_abi,
77          )
78          nonce = w3.eth.getTransactionCount(w3.toChecksumAddress('0x' + self.
```

```
            server_eth_address))
79          f_call_txn = mdt_meeting.functions.setEvents(start_event_hash,
                end_event_hash).buildTransaction({
80              'nonce': nonce,
81              'chainId': self.chain_id,
82              'gas': 2000000
83          })
84          signed = w3.eth.account.signTransaction(f_call_txn, private_key=self.
                server_private_key)
85          contract_txn_hash = w3.eth.sendRawTransaction(signed.rawTransaction)
86          print('Waiting for TX to be mined...')
87          tx_receipt = w3.eth.waitForTransactionReceipt(contract_txn_hash)
88          return tx_receipt
```