ABLOC: Accountable Blockchain Logging for Offline Care

by

Joseph Adler Krysl
B.Sc., University of Victoria, 2021

A Thesis Submitted in Partial Fulfillment of the
Requirements for the Degree of

MASTER OF SCIENCE

in the Department of Computer Science

ABLOC: Accountable Blockchain Logging for Offline Care

by

Joseph Adler Krysl
B.Sc., University of Victoria, 2021

Supervisory Committee

Dr. Jens Weber, Supervisor
(Department of Computer Science)

Dr. Morgan Price, Supervisor
(Department of Computer Science)

ABSTRACT

Retroactive security is important to cyber security; it is used to hold people accountable for their actions [1]. In the medical world, it is difficult to assign proper privileges, as they can be too wide and vulnerable to misuse, or too narrow [1, 2, 3, 4] restricting access to patient data [2, 4]. Clinicians are often given wide privileges to ensure they can access the data required to care for patients [2]. Logging is relied upon to find breaches of policies [2, 3, 4, 5] but, without reliable logs, changes can be made to the data in the EMR without anyone knowing [6]. Blockchain-based logging has been proposed but requires a stable internet connection [7]. This thesis presents Accountable Blockchain Logging for Offline Care (ABLOC), a Directed Acyclic Graph (DAG) based blockchain, that is combined with a gossip protocol to improve the forensic reliability and accountability of logs. ABLOC can tolerate participating realms, the internet space that houses one or multiple pieces of medical software, going offline, recovering, and resynchronizing with the rest of the network. The ABLOC system receives log hashes, summarizes them, and shares the summary with different realms on the ABLOC network. This work presents the necessary background information, discusses the design of the ABLOC system, and evaluates the proposed system theoretically and with a prototype. The proposed system has promising results in the scalability tests performed.

Contents

List of Tables

List of Figures

Acknowledgments

I would like to thank Dr. Jens Weber and Dr. Morgan Price, my supervisors, without whom this research would not have been possible. Thank you Jens and Morgan for answering my questions, and for giving advice, and support. Thank you Dr. Bill Bird for research and non-research-related discussions, advice, and occasionally encouraging some procrastination.

Thank you to my parents for their support and encouragement throughout this degree and my undergraduate degree. Finally, thank you to my partner, Hannah, for supporting and encouraging me throughout my degrees, helping to make sure this document is grammatically correct, and putting up with all the computers that were around the house while I was running experiments.

Dedications

In memory of my grandparents, Josef and Zora Krysl.

Chapter 1

Introduction

Healthcare providers rely on digital tools to improve the efficiency, delivery, and quality of care they provide. The goal of these tools is to provide clinicians with the information they need in order to care for a patient. Each medical site should have a disaster recovery plan that includes how the site can continue to provide care in case of an internet outage or other threats.

Logging is relied upon for forensic investigations, for investigating data breaches, and other malicious actions that have impacted software. It may be difficult to trust a log file when auditing a system because it may have been modified. Mechanisms to increase the accountability of log files exist but may be vulnerable to attack and modification, themselves [8]. These systems must be reliable regardless of their connectivity status in order to continually log the actions taken if a clinic is offline but the health information system is still functioning. This work presents ABLOC: Accountable Blockchain Logging for Offline Care. ABLOC is the solution to the problem of detecting log modifications, using blockchain, even when the software generating the logs is offline.

Chapter 2

Problem Description

With the use of medical software, called Health Information Systems (HIS), retroactive security is becoming important to the theory and practice of cyber security in the medical field. Retroactive security is beiHng used to correct errors, detect fraud, and hold employees accountable for their actions [1]. In the medical world, it is difficult to give clinicians proper access control policies, as these policies can be too wide (vulnerable to misuse) or too narrow [1, 2, 3, 4]. If too narrow, it is possible that clinicians may not be able to access records when needed [2, 4], so clinicians are often given wide privileges to ensure they can access the data that they require [2]. Alternatively, Break the Glass methods of information access (where clinicians are able to access confidential information in an emergency) need to be logged to ensure that the information was accessed appropriately [1, 9].

To enforce policies used to manage wide access to HIS for clinicians, the log of actions is relied upon and analyzed to find breaches of policy and privacy violations; these logs must be accurate [2, 3, 4, 5]. Privacy violations may be the result of an internal or external malicious actor but more commonly they are caused by an authorized user abusing their credentials or acting in a careless way [10]. Data reliability can be poor and logs could be tampered with or modified to better suit a malicious user's purpose; this makes it difficult to trust log files [11]. Without logs, changes can be made to the data in the software without anyone knowing, covering up errors or malicious actions [6]. Logs can also provide a record that practitioners can use to show that they did the right thing at the right time [6].

A logging system that is forensically reliable (usable for holding people accountable for their actions) is critical in the healthcare industry. Blockchain has been used to make logs more forensically reliable, through the inherent properties of being tamper-resistant, by storing the logs on the blockchain or off the blockchain and securing them with a hash. These solutions require each member of the blockchain to be connected to the network [7, 12, 13, 14, 15, 16, 17, 18].

While most internet service providers strive for highly reliable internet connections, network outages may still occur. These outages can be caused by human factors or natural phenomena. Work on network traffic routing has been done to mitigate large-scale network outages, though small outages still occur [19]. Configuration errors may cause fragile networks to crash [19, 20], and in some cases, this can lead to widespread cascading failures, such as the Rogers outage in 2021 [21].

Use Case: Dr. Smith is a physician at a family practice clinic in Victoria, BC. The practice uses a locally hosted instance of OSCAR EMR and Dr. Smith is a power user. Dr. Smith hears that his favourite NHL hockey player has become a patient at the clinic with Dr. Jones. After an injury on the ice during playoffs, Dr. Smith's curiosity gets the better of him and he peeks into the chart, against the clinic's policy, and modifies the log entries to cover his tracks, implying that his actions were legitimate. A regular review of the EMR's logs in comparison with the information stored on the ABLOC system reveals inconsistencies. This is brought to the attention of the system administrators.

Blockchain research, up to this point, has been focused on preventing partitions in the blockchain [12, 17], which can be caused by internet outages. Generally, a loss of internet results in a blockchain not being accessible for adding transactions [17, 22] and a member could not secure their log entries on the blockchain. There has been some research into partition tolerant blockchains [23, 24, 25] to address sites with poor internet connectivity.

The goal of this thesis is to investigate partition-tolerant blockchains, that continue to function with the loss of network connections and to apply offline tolerant blockchain logging to HIS, including electronic medical record (EMR) systems. This thesis presents a scalable blockchain-based logging system that is tolerant of network out-

ages and that is able to generate summaries of log entries. In the next chapters, this thesis presents background information, describes the proposed system in detail, and then evaluates the proposed system.

Chapter 3

Background and Related Work

This chapter focuses on the background material required for discussing and implementing the ABLOC blockchain. This chapter will discuss general information about blockchain, including the types of blockchain, applications of blockchain in the medical field, general information about logging, general information about medical software, and applications of logging in the medical field. Finally, this chapter will give background on gossip protocols and present examples of gossip protocols. These topics present background information and introductions for aspects of the ABLOC system, which will be presented in Section 4.

3.1 Blockchain

Blockchain is a chain of data grouped into blocks; a block can refer to groups of transactions, other information, or executable code, called smart contracts [13, 15, 16]. Blockchain is a peer-to-peer distributed ledger or medium, where each node contains a copy of the data stored in the chain [11, 12, 13, 14, 15, 26]. The data stored on the blockchain is immutable and is considered to be tamper-proof [12, 13, 14, 15, 16]; each node stores and validates blocks that are chained together using hashes [12]. A cryptographic hash, the result of the data being fed into a one-way mathematical function, of the previous block in the chain is embedded into the new block that is being created. Any changes in the earlier data, also secured with hashes, would result in changes to the current hash, invalidating that copy of the blockchain. Therefore, peers are allowed to add new information to the end of the ledger, but no one is able

to delete or modify historic data [13, 16].

When a blockchain network grows, generally the speed of transactions decreases, this is dependent on the consensus algorithm and how nodes are rewarded [11, 13, 27]. A blockchain will always be slower than a typical centralized database; for each transaction, the regular database tasks are executed in addition to validating the transaction and running the consensus algorithm [13].

The data that is stored in each block is used to create a Merkle tree, this generates a hash of all the data to store in the block [13]. A Merkle tree is a data structure that stores hashes of each data entry as leaves in a binary tree. Pairs of the leaves are hashed and the result is stored in the root of the sub-tree [28]. Merkle trees allow for efficient verification of the data set [13, 28]. A digital fingerprint, called the Merkle root, is the final hash and is the root of the tree. An example of a Merkle tree is presented in Figure 3.1. A Merkle tree can be made from the bottom up and it can be used to prove if the data within a block has been modified [13].

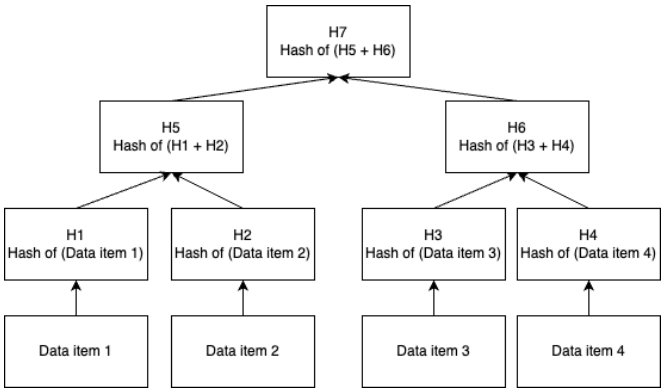

Figure 3.1: An example of a Merkle tree built from the bottom up.

3.1.1 On-Chain vs. Off-Chain Data Storage

The data managed by a blockchain can be stored on-chain or off-chain: off-chain avoids the computation costs of a traditional blockchain, where each node must store the transactions [14, 29]. Instead, data can be stored in the Interplanetary File System [26] or other storage solution where it is referenced by the blockchain [18, 30]. On-chain solutions (all transactions stored by all nodes) may be desired in some scenarios as each node contains a copy of all the data that is validated [14]. If this was applied to logging, it would allow for more transparent auditing [31] by all members of the blockchain. When data is stored on the blockchain, it is possible to define ways to keep it private and secure, yet accessible, using smart contracts [16].

3.1.2 Types of blockchain

Depending on the application, there are different types of blockchain networks and consensus algorithms; blockchain networks can be public or private [13, 16, 26].

3.1.2.1 Public blockchain

Public blockchains are open to any nodes and are transparent to all [13, 32]. Nodes may come and go at any time [14, 32]. No single entity is in control; decisions are made via the consensus algorithm [13, 14, 15] for the overall benefit of the blockchain [13]. Nodes in a public blockchain do not need to trust one another: a consensus algorithm is used to make decisions on the chain [14, 15, 24] building trust. Public blockchains are more decentralized than private blockchains; they often have a large number of nodes [13, 24, 32, 33] but the per-transaction cost tends to be higher [13, 7]. Bitcoin is an example of a public blockchain [13].

3.1.2.2 Private Blockchain

A private blockchain is an asset of an organization or individual; an entity is in charge of the network and makes decisions about how it is run [13, 29, 33]. These blockchains limit participation through identity management [25, 34], are more centralized than a public blockchain, and generally have fewer nodes [33]. Like a public blockchain, a private blockchain is still an immutable way to store data that relies on hashing [13]. Private blockchains are used by groups of people who have a common goal and relationship [33]. The participants thus have a baseline of semi-trust, as opposed to no trust, in one another. This is an advantage because less computationally intense consensus algorithms may be used [13, 25, 32]. Apart from controlling who participates, and is able to read the blockchain, private blockchains operate similarly to public blockchains.

3.1.2.3 Consortium Blockchain

Namasudra et al. state that a consortium is a compromise between a public and private blockchain that tries to remove the disadvantages of the private blockchain, in which a single entity could have all the rights. The data stored on a consortium blockchain is not public. It is closer to a private blockchain, as access to the blockchain is restricted to member organizations. Multiple groups or organizations that use the

blockchain form a consortium to manage it; a group of representatives from the organizations make decisions for the overall benefit of the network. [13].

3.1.3 Consensus Algorithms

Depending on the type of blockchain, public or private, different mechanisms are used to make decisions. These decision-making algorithms are also called consensus mechanisms; they attempt to achieve an immutable consistent registry of transactions [12]. Some consensus algorithms are seen as wasteful technology due to their massive energy consumption. The main consensus mechanism that this impacts is proof of work; which is used by Bitcoin [13]. Different strategies are used on private and public blockchains and may be specific to an implementation. Generally, each node in a public blockchain may run the consensus algorithm and compete for rewards in a process called mining [12, 17, 35]. Carrara et al. indicate that consensus algorithms are impacted by the CAP theorem; which identifies three characteristics critical to any distributed system: **C**onsistency (ensure that information is the same on each node to maintain a global view of the system), **A**vailability (give a valid response, other than an error message, to requests made by another node), and **P**artition tolerance (if the network connection is lost, the algorithm continues to prevent partitions in the distributed system). Networks can generally only provide two of the three through a consensus mechanism. Carrara et al. also claim that consensus algorithms can be categorized into probabilistic and deterministic algorithms [12].

3.1.3.1 Probabilistic Consensus Mechanisms

Probabilistic consensus mechanisms are likely to reach consensus, and deterministic proprieties are sacrificed in order to favour partition tolerance from the CAP theorem. Probabilistic consensus algorithms include proof of work and proof of stake [12].

Proof of Work The proof of work consensus algorithm is often used in public blockchains with large numbers of anonymous nodes who do not trust one another. Miners (nodes in the network) are responsible for solving a cryptographic puzzle, usually by brute force. Once the problem is solved the solution is incorporated into the block allowing it to be easily verified by the rest of the network. Once more than half of the peers have validated the block it is added to the chain [12, 13, 17]. This is a costly process as energy costs are high and is a burden on computational resources

[7, 11, 12, 13, 25]. With proof of work consensuses algorithms, the longest chain is the authoritative one [11]. Bitcoin is an example of a blockchain that uses the proof of work algorithm; it is estimated that at its peak Bitcoin mining collectively used more electricity than was being generated in Denmark [36].

Proof of Stake Proof of stake consensus was created to overcome the inefficiencies of proof of work. The lead miner is chosen through a competition and is responsible for the next block that is being added to the blockchain. The competition is driven by the amount of investment that a miner has in the blockchain and also has an element of randomness to prevent highly invested miners from exerting too much influence on the blockchain. The amount of investment made by each candidate is proportional to their chance of being chosen as the leader. The chosen leader's investment is deposited as a security deposit; if the leader does not behave honestly then this deposit is forfeited. At least half of the remaining network must approve the newly created block [12]. Proof of stake avoids the overhead of proof of work allowing a reduction in the confirmation time [11] making it more efficient. This consensus protocol is used in Ethereum [12, 37].

Other probabilistic consensus mechanisms exist, including proof of elapsed time where a specific waiting time is generated by specialized hardware to ensure that a node must wait a random amount of time before it can become the blockchain leader again. The specialized hardware creates a certificate that a node can use to become the next leader [12]. Due to having special hardware requirements, this is not as widely used.

3.1.3.2 Deterministic

Deterministic consensus mechanisms tend to sacrifice availability in favour of strong consistency. Rounds of deterministic consensus will always end, even if consensus is not reached. Examples of deterministic consensus algorithms include Raft and Byzantine Fault Tolerance. [12]

Raft Raft is the primary consensus mechanism used in Hyperledger Fabric. It uses a leader-follower model that is fault-tolerant. A leader is elected, transactions are replicated, and then secured. Any node in a Raft network must be one of a candidate, follower, or leader, with the default being a follower. If no leader is noticed, an election takes place and once a node receives a majority it is promoted to leader. A

leader communicates with the blockchain interface and receives the information that is communicated with its followers. Additionally, a leader ensures that their followers remain active. Out-of-date followers must be reintegrated and updated. The leader maintains the latest version of the ledgers [12].

Byzantine Fault Tolerance This consensus algorithm addresses the reliability of a system where up to a third of the members may be malicious. A block generator validates and collects details for the transactions for the blockchain. These transactions are grouped into blocks. Previously agreed-upon rules are applied by the nodes and blocks are signed in accordance with these rules. [13]. Three times the number of malicious nodes in the network must agree before the block can be processed [24]. Consensus is reached when the block is signed and accepted by the participants [38].

A consensus algorithm is chosen when a blockchain system is being implemented. This choice may be dictated by the blockchain or the blockchain may allow a choice of different algorithms; Hyperledger Fabric allows the implementer to choose an algorithm through their plug-able consensus architecture [39]. The consensus algorithm allows the blockchain network to make decisions and approve changes to the system, which is used to approve transactions and new smart contracts that are being stored on the blockchain. These algorithms require nodes to be connected to the network in order to participate in the process; generally consensus algorithms attempt to avoid partitions in the blockchain [25].

3.1.4 Blockchain Partitions

Most traditional blockchains aim to avoid partitions, also known as shards, in the blockchain. Partitions are considered to be Byzantine Faults and can result in the blockchain believing in multiple truths and branching into two chains. [17]. It is possible for a fork to have appeared and for only the nodes at the junction to observe it [40]. The goal of most blockchain systems has been to prevent forks in the chain; this is a significant issue faced by cryptocurrency developers [11]. Partitioning is a well-known vulnerability that impacts peer-to-peer networks, including blockchain. [17, 23] It has been reported that forks may result in less than 51% of the total network computational power being required, by malicious actors, to control a proof of work blockchain with forks in it [40]. Therefore, avoiding unwanted partitions is a priority; to achieve this nodes that are offline cannot participate or risk it causing a

partition in the blockchain [17, 23].

Research has shown that intentionally sharding blockchains can improve scalability; nodes are sorted into sub-networks, each shard is responsible for managing its own blockchain [14] and graphs of the transactions are maintained to determine inter-shard transactions [41]. This introduces the idea of replacing the linear blockchain structure with a directed acyclic graph-based blockchain (DAG).

3.1.5 DAG Blockchains

Directed acyclic graph (DAG) based blockchains use a collection of nodes and edges that make up a graph as opposed to a linear chain, see Figure 3.2. DAG blockchain graphs do not contain cycles. This structure is used to replace the underlying linear blockchain structure, and it may allow each block to have multiple children or parents on the blockchain [11, 23, 25, 27, 42].

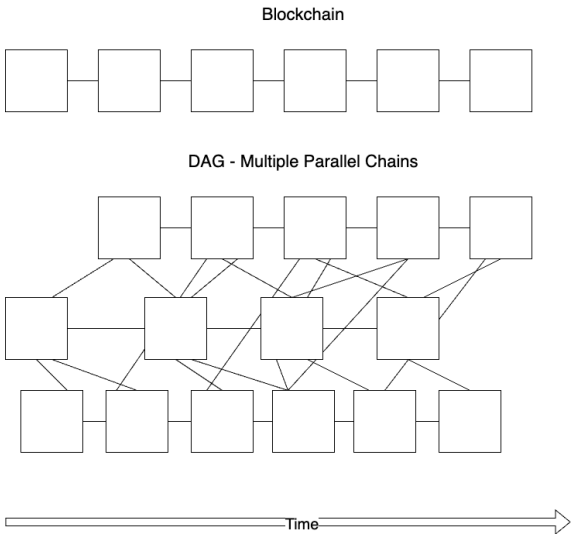

Figure 3.2: Linear blockchains (top) have a single chain and is contrasted to a DAG based blockchain with multiple parallel chains.

DAG-based blockchain implementations are currently disorganized; the basic concepts of DAGs are still maturing [27, 43]. Some DAG implementations do not support all the features of traditional blockchain [44]. Many DAG systems use a fault-tolerant consensus algorithm and message distribution protocol [11, 23]. The DAG-based blockchain is a more sophisticated structure, when compared to a linear chain, as

it stores the relationships between pieces of information [25]. Blocks in a DAG can serve as a cryptographic witness to another, or many other, validated block(s) [23, 25], proving the order of events received by the blockchain [25]. Members of the DAG blockchains witness the information that other nodes put onto the blockchain to ensure that it is immutable [27].

Depending on their approach and application, DAGs can still be prone to network inconsistencies and conflicting transactions. Speed and latency matter in some blockchain applications like payment processing [42], and DAG-based blockchains tend to be faster [23, 44]. Some DAG approaches are partition tolerant, where each user can unilaterally publish new data to the ledger using their private key. If a group of these users is cut off from the main group, they are still able to validate each other's transactions within a limited transaction scope. Due to the small scope, there is no risk for overlap [25].

DAG-based blockchains create a higher performance, lower cost way to manage data when compared to traditional blockchains [23, 27]. Using a DAG may allow blockchain to be used where traditional blockchain is not suited, such as areas of low or intermittent internet connectivity and power-constrained environments [23]. Examples of public DAG blockchains that maintain non-repudiation, and immutability are IOTA and Byteball [25]. Byteball can not tolerate partitions in its chains [45] and in these implementations disconnected nodes can't publish to the larger network until they reconnect [25]. Byteball and IOTA attempt to make blockchain more efficient at confirming transactions [44, 45].

Some DAG-based blockchains are purpose-built and have the structure of a naturally expanding graph, others anticipate that partitions will be re-merged into the main chain, and others use the idea of interconnected parallel chains [27]. The different approaches use different consensus algorithms [27] to try and create methods to store data for Internet of Things (IOT) data [23, 27].

3.2 Blockchains in Medicine

The use of blockchain has been proposed for use in the medical field to solve many problems, such as sharing data with other institutions, supporting EMR and HIS systems, and storing logs.

Data sharing in healthcare could benefit from blockchain. An example of this includes securing data from medical research and development where the blockchains are used for privilege management and to share information from clinical trials in a way that the data can not be modified [16]. It has also been proposed to use blockchain to share information with insurance companies to settle claims faster, to track prescriptions to prevent counterfeits [26], and to share data in a way that minimizes the risk of jeopardizing the user's privacy and security [16]. Blockchains could be used as a secure way to collect data from healthcare IOT devices that handle large amounts of information and can be used to track patients' compliance with their treatments. Sharing information between institutions in the medical field also could include tracking the manufacturing of medications, preventing counterfeit drugs, and increasing the reliability of product registration and tracking [16, 26].

In addition to data sharing it has been proposed to use blockchains to support EMRs and other HIS. A blockchain-based EMR was proposed for New Zealand to help address the increasing demand for care and the increased cost of delivering it. The blockchain would scale and allow the sharing of records, retain a centralized consent repository, and use smart contracts to manage access to the data. [46] Similar proposals have been made for other countries including Bangladesh. These systems aim to provide a more efficient way to share data between providers (each site has a copy of the blockchain ledger containing the data) and manage access to the data stored on the blockchains, through smart contracts, in order to improve patient privacy [30, 47, 48].

Blockchain has also been proposed as a method to secure audit logs in medicine. The proposal focuses on precision medicine where information is uploaded to a server for processing. Some of the log entries for this event would be uploaded to a permissioned blockchain. The proposal uses smart contracts to outline how transactions can change the state of the ledger. It stores some of the log information on the blockchain and the rest is stored locally, with a hash uploaded to the blockchain to secure the log [18].

These applications of blockchain rely on the sites having stable internet connections to support data sharing, medical record systems, and logging in the medical field.

3.3 Logging

Non-repudiation (being able to prove that a user did perform an action on a system) can be implemented through a logging mechanism that provides a record of user activity [49]. Logging mechanisms have been used to debug software, diagnose faults, and monitor the performance of the software; non-repudiation has often been done as an afterthought [50, 51]. Logs are used as a record of what events have occurred [49, 51], and these events are recorded as individual entries into a logging mechanism [51].

Logging is an important aspect of developing software but it may be difficult for the developer to determine, from software requirements, what activity must generate logs [49, 50]. Developers may overlook events that may require logging, misunderstand the purpose of the logging, or assume the debugging logs do the same thing as the non-repudiation logs [49]. The entries that do get logged can be used for retroactive security, allow an organization to detect fraud and errors and hold employees accountable for their actions [1]. To audit the events a user-initiated, with a piece of software, it is necessary to record sufficient information, but not too much information, and ensure that the logged data has not been tampered with [1].

Applications that generate their own logs should record requests and responses to and from their clients. This should include sign-in attempts, the use of privileges to access information, and significant actions including restarts, failures, and configuration changes [51]. Other sources indicate that it is best practice to log any Create, Update, Read, or Delete (CRUD) actions to a database [49, 52, 53]. Healthcare data should rarely be deleted, doing so must be logged. According to Kent, only the information that is considered to be the most important for the scenario needs to be stored [51].

Securely storing logs is important; logs contain sensitive information about users. Logs may capture email addresses and other sensitive information; log file data breaches could lead to serious privacy compromises [51]. Confidentiality, Integrity, and Availability - the CIA triad - need to be upheld for log files [51]. If the logs are not secured properly, it would be possible to modify or delete the log entries in order to conceal malicious actions [51].

3.3.1 Existing Solutions for Forensic Log Reliability

Infrastructure and administrators need to protect the integrity and availability of logs [51] as they are a key portion of forensic investigations [54]. According to the Open Web Application Security Project (OWASP) this is not an easy task to manage; the trustworthiness of the log storage mechanism is usually relied upon [55].

Logs that are stored at the system level are considered to be less important than logs that get sent to a central repository [51]. Users should not have access to the log files and if they must have access, they should have append-only privileges. The user should not be able to rename, delete, or modify log entries or log files [51]. Furthermore, unauthorized parties should not be able to manipulate the log system executables, or other components that can impact the logging tools [51].

3.3.1.1 Syslog

Kent and Souppaya describe Syslog, in a National Institute of Standards and Technology (NIST) standard document, as a server that provides a framework for operating systems, security software, or applications that can send their logs to the Syslog server. They go to on say that Syslog was developed in a time when the security of the logs was not as important as it is today. This is because sending logs to the server did not require authentication. Kent states that Syslog relies on the UDP protocol which can be vulnerable to a denial of service attack or intercepted, as logs may be sent in plain text. If plain text logs were intercepted they could be modified using a man-in-the-middle attack. Syslog has been adapted to require authentication and encryption of the logs in transit [51]. Kent and Souppaya's work is corroborated by Sato and Yamauchi, who add that attackers may try to compromise the Syslog program itself making the log file unreliable [8].

3.3.1.2 Security Information and Event Management Software (SIEM)

The NIST standard, authored by Kent and Souppaya describes SIEM as the amalgamation of different logs from many sources and is used to analyze the logs for potential events. They state that SIEM systems use credentials and encryption to protect the logs it stores and has tools built in to visualize the entries. These systems are complicated and expensive to deploy [51]. Using the same logic as Sato and Yamauchi did above [8], if the SIEM system were attacked the logs stored in it would be unreliable.

SIEM and Syslog are two example solutions that are central repositories for storing logs. Alternative solutions exist, including write-once-read-many memory but this is not scalable [7, 56]. It would also be possible to continually print out a copy of the logged events [56]. This was used by the Bangladesh Bank, which was impacted by the Lazarus heist [57, 58]. While this may work for major transactions at a national bank, it would not be scalable for the large number of log entries that are generated by today's software. In 2004, before blockchain technology, Snodgrass et al. proposed using hashes to prove that logs were not tampered with or corrupted. These hashes would be sent to a third-party notary service [56]; this would not be possible during an internet outage.

3.3.1.3 Blockchain Log Storage

Blockchain has been proposed to store logging information in a way that guarantees tamper resistance [7, 54] but this requires consistent internet connectivity. By storing encrypted logs on the blockchain, these peers pool resources to better protect the logs of everyone who is participating in the network [7]. If a site goes offline, it would have to rely on a backup method to secure the log files generated during the outage.

3.4 Medical Software

EMR are used in general practice clinics, have become a popular alternative to paper medical records providing a convenient and efficient way to access patient data and have become an indispensable part of the healthcare system [2, 3, 35, 59]. Data collected by the Canadian Institute for Health Information's 2022 Commonwealth Health Survey shows that 92% of family doctors use EMRs in their family practices in Canada [60]. For some clinicians, EMRs are simply a digital replacement for paper records, but EMR systems may also include user support by providing access to complete and accurate data, alerts, reminders, and other aids [59]. EMRs may be incorporated into other peices of software that may be used in larger clinics or hospitals. EMRs can be grouped under the larger umbrella of Health information Systems (HIS).

HIS may include software for different departments in a hospital. A HIS may support laboratory, radiology, pharmacy, and other areas of practice. It is possible that these pieces of software originate from multiple vendors and have different strategies for

storing logs generated by the software. These systems can be grouped under the wider umbrella of HIS.

Data stored in HIS can be accessed by multiple clinicians at the same time and HISs allow for digital searching and retrieval and other clinical supports to aid clinicians [59]. HIS may be interoperable and allow for the digital transfer of information from one hospital or clincial site to another. This would allow a patient's record to be available to multiple sites within a region and allow the record to be shared with a care team - one person having the record does not mean that others cannot access it.

Sites using HIS may or may not store all of their data onsite. It is expected that at least some of the information is stored onsite to facilitate providing care to patients in the event of a network outage. From a regional standpoint, health data is often stored at different locations and is siloed: information from one record is not accessible in others. Integration of the records is difficult because healthcare is a distributed service [4, 26].

In healthcare, the majority of threats do not involve sophisticated attacks; often the causes are misuse, human errors, or physical actions. Misuse is involved in approximately a third of breaches [54]. External threats include connectivity losses, through denial of service attacks, fraud, or social engineering, leading to device compromise with the end goal of fraud or data loss [5] for extortion or financial gain [5, 61]. The majority of internal threat actors were able to abuse their credentials; Verizon found that this occurred in over half of internal data breaches. Verizon claims that internal breaches and data mishandling make up three-quarters of all healthcare data breaches, from internal threat actors [10]. This is corroborated by Chernyshev et al., who go on to remind readers that breaches also impacted paper records [61].

3.4.1 Logging in Health Information Systems

Forensic analysis is used in the financial industry to detect fraud and transaction logs are used to support this. Reviewing access is not as common in the healthcare industry [49]. In the medical world, it is difficult to give clinicians proper access control policies. A clinician may have permissions that are too wide and vulnerable to misuse or too narrow, which could restrict [2, 3, 4] a clinician from accessing critical lifesaving information when needed [2, 4]. Clinicians are often given wide privileges or are able to override restrictions to ensure they can access the data they require in an

emergency [1, 2, 9, 23]. The access, modification, and deletion of patient information must be logged to ensure that that access was appropriate [1, 6, 9, 52, 62].

Log files may be audited to find breaches of policies so must be accurate [2, 3, 5]. If log files do not exist, it would be possible for someone to modify the data stored in the HIS without being held accountable [6]. Logs can also be used to defend clinicians, to prove that they did the right thing at the right time [6]. It is noted that keeping redundant copies of logs, to protect against modification or deletion, is a challenge [63]. Log tampering can involve adding or removing log entries, manipulating parts of a log file, or deleting the entire file's contents. This may be done to shift blame or to cover up mistakes made by employees [54].

Not all HIS adequately log events; depending on the situation this could be illegal, lead to patients mistrusting the system [6], or becoming the victims of fraud [1, 5]. It is not uncommon for employees snoop into medical records of friends, family, famous people or themselves, employees generally have no reason to access these records. many organizations use HIS that do not have adequate protections in place to detect this [52]. The logging mechanisms for HIS should note all CRUD accesses of personal health information [52].

King and Williams state that the following information should be stored in a log file from an EMR, and logically other HIS:

- IP Address [51, 64]
- Date and time of the event [52, 64]
- User Identifier [52, 64]
- Patient Identifier [52, 64]
- Type of action [52, 64]
- Identification of the patient data that was accessed [52].

In 2014, King and Williams investigated the status of the logging modules in a sample of open-source EMRs, including OSCAR and OpenEMR [52]. OSCAR and OpenEMR were found to pass the most test cases (accessing, changing, and recording patient data, and accessing and deleting log file entries) out of the four EMRs reviewed with OpenEMR recording 63% of events and OSCAR logging 38% of events

[52]. The OpenEMR logging mechanism was reported to have security weaknesses that failed to make the logs immutable [52]. King and Williams also state that the guidelines for which modules in an EMR where the logging of CRUD actions are necessary is not specific [52]. It is also noted that the data generated from the logging of EMR data is considered to be medical data, as it may contain requests and responses about the patients; this data must be protected no differently than other data stored in an EMR [62].

Tampering with logs can impede investigations and lead the investigation to a dead end. Deletion and tampering with log entries are often done by malicious actors to hide their tracks [5, 54]. Depending on the implementation of the logging software, with the EMR, clinicians may not know if the logs have been tampered with [6].

3.5 Gossip Protocols

Gossip protocols are peer-to-peer networking algorithms that allow information to be spread to the network members, often called nodes, through the epidemic spread of messages [65, 66, 67]. Each node in the network can share the messages it has received with other nodes that are members of the network [65, 66, 67], resulting in a probabilistic guarantee of a message to all group members [65] without each member receiving the message from the source directly. This generally allows gossip protocols to scale well and be fault tolerant [65]. Gossip protocols are used to broadcast messages to a network and are not used to come to a consensus or to make decisions for a network.

3.5.1 Literature Review

A non-exhaustive literature search was conducted looking for gossip algorithms that tolerate partitions or offline nodes in networks. Gossip protocols are a common way to distribute information in DAG blockchains [23]. Papers that implemented gossip protocols that support nodes in low connectivity settings or that support blockchain networks were reviewed and are presented below.

3.5.1.1 Vegvisir: A Partition-Tolerant Blockchain for the IoT

This paper looks at creating infrastructure that works well with IoT devices [23]. These devices may not be able to support traditional (Nakamoto) blockchains which require high network connectivity [23]. Karlsson et al. present a partition-tolerant DAG blockchain that relies on a gossip protocol. Each node in the protocol stores information about every node in the network and all the messages it has received. Each node also stores the frontier set, a set of blocks in the network that have no successors; they are the most recently generated blocks [23]. When gossiping, nodes exchange the frontier sets. If two nodes' frontier sets are the same, then the blockchains are also identical [23]. If two nodes' frontier sets are not the same, the new block(s) are added [23] and the blockchains are synchronized. If this operation fails because the parent blocks of a block are not known, then the node can recursively request the parent blocks from the node it is gossiping with [23]. This happens until a known block is encountered [23]. This implementation was designed for when nodes have a low bandwidth connection or the network has been split into two [23], and likely can be adapted for when a node comes back online and needs to recover.

3.5.1.2 Correctness of a Gossip-Based Membership Protocol

This paper discusses using a variation of probabilistic gossip protocols [65]. The authors choose to discard the strong reliability guarantees for probabilistic guarantees to allow protocols to deliver increased scalability [65]. Additionally, they discuss gossip protocols that use random choices of nodes and how this could be improved by storing smaller subsets of the entire network [65]. The focus of this work is to find a scalable way to manage membership and churn that may be present in a network [65]. The authors propose a local, fixed-size, subset of the network that any node manages [65]. Each node continually updates its subset [65]. Two other parameters are stored: f, the fanout and w, the weight [65]. For each round of the algorithm a node, n, constructs a list of f nodes chosen at random from its local view, and another list of nodes that requested n's local view [65]. A new view is created from these lists; w is the probability of the nodes being chosen from the list that requested n's local view [65]. This algorithm does not expect nodes to leave gracefully and manages churn well [65].

3.5.1.3 WIISARD: A Measurement Study of Network Properties and Protocol Reliability during an Emergency Response

This paper describes an emergency response system that allows responders to communicate in an emergency with minimal external infrastructure support; the authors call it WIISARD [68]. WIISARD employs a gossip-based protocol to support data dissemination and data transport [68]. It achieved 98% reliability in a drill and was shown to be a reliable communication method [68]. Each peer in the network divides time into periods; each period has W time slots [68]. Peers do not synchronize times [68]. At the start of each time period a node, N, transmits a beacon summarizing the blocks it has stored [68]. A neighbour who receives the beacon, indicating that the sending node is still alive, determines if it has blocks that N does not [68]. The peer transmits some of the missing blocks back to N [68]. One block is transmitted per time slot in W [68]. Blocks are not re-transmitted to avoid a broadcast storm [68]. The peer N can cancel the transmission of a block if it hears other identical transmissions [68]. This avoids multiple peers transmitting the block to N [68].

3.5.1.4 A Partition-tolerant many-cast Algorithm for Disaster Area Networks

Asplund and Tehrani present a many-cast algorithm that is gossip-based [67]. This algorithm is intended for use after a disaster where cellular infrastructure may not function and an ad hoc network would be needed [67]. The proposed algorithm can deal with intermittent connectivity, tolerate partitions, and contains no knowledge about the network topology; it is an efficient way to transmit data [67]. The proposed algorithm is a random walk algorithm that delivers a message, m to k nodes [67]. A node either is the custodian of m and will actively re-transmit it, or inactively hold m [67]. Neighbouring nodes request messages and will become responsible for transmitting them in place of the neighbour [67]. Inactive nodes, retaining the message, exist to become active if an uninformed neighbour is discovered [67]. When all k nodes have been informed the messages are deleted [67].

Chapter 4

Proposed Design of the ABLOC System

4.1 Overview

In the ABLOC system, each HIS implementation will be included in an ABLOC network realm. A realm is defined in Section 4.2; it is the network area where one or more HIS store and allow access to their data. Within each realm, the network should run its own blockchain to store hashes of log data generated by the HIS. Each realm will be responsible for running its own blockchain, allowing it to act independently of other realms. Each realm will allow others within the regional area to store summarized versions of its logs on the local realm's blockchain. This block will reference, via a hash, the original block(s) on the original realms' blockchain. This will result in a summary of the local realms' data being stored on other realms, using hashes to witness the original log hashes (stored on the original realm's blockchain), to provide further anti-tamper properties.

This idea could then be replicated across regions, for a second summary layer (allowing summaries of summaries), and so on. This results in the independent blockchains becoming cross-linked, forming a DAG made up of parallel blockchains that are able to function independently when a realm is offline. A similar DAG blockchain is presented in Karlsson et al.'s work Vegvisir: A Partition-Tolerant Blockchain for the Internet-of-Things [23]. This idea can be adapted to increase the forensic reliability

of logging for healthcare, making it tolerant of external network outages. While this system cannot create logs for actions that should be logged but are not, it would be able to prove that logs have not been tampered with, even if the local blockchain has been corrupted. The assumptions made when creating the ABLOC system and the details of the ABLOC system will be reviewed. This proposed method provides a forensically reliable way to protect logs that allows clinics or hospitals to go offline and still maintain logging capabilities.

4.2 ABLOC Realms

An ABLOC realm is the network area that encompasses the activities of a HIS. Each realm in the ABLOC system is intended to serve a single medical software instance with a blockchain unless another piece of software is implemented within the same networked locations (i.e. within the same data centre and hospital). In its simplest case, one ABLOC realm will have one HIS. Some physical sites may run multiple pieces of HIS that generate log data, which could result in multiple realms. If two or more HIS are resident in the same network areas (their data are stored and accessed in the same locations) then one realm can encompass multiple HIS. Some of these pieces of software may be hosted in the cloud or within a hybrid cloud-onsite model. All of these systems should also be able to use ABLOC to increase the accountability of their logs.

In the case of family practice clinics, where it is likely that the clinic is only running the EMR software, the realm would only encompass that single site's HIS. In larger organizations such as hospitals, EMRs, used for charting, may be complemented by separate HIS for laboratory, pharmacy, and radiology departments. Each HIS may have its own realm or they may share a realm, depending on where the data is physically stored and accessed. Furthermore, information from different systems (electrocardiography machines and other scanners) or personal health applications may generate logs that would need to be forensically validated. Health authorities may run a single instance of a HIS in all the hospitals across a health authority; in this case, the realm would encompass the entire implementation of the HIS.

Each realm will contain a blockchain to store the logs generated by the HIS and the summaries generated from these log entries. For system implementations that are cloud-based, the blockchain should be implemented in such a way that allows it to

process logs on the same site as they were generated if there are network outages. In this case, realm members not in the same location as the logs being generated will not be able to participate in the realm until they are brought back online. This is the typical strategy for nodes going offline in a blockchain network [22].

4.2.1 Single HIS in One Realm

If a single HIS is only used in one location, then it occupies an ABLOC realm. This definition could include a family practice clinic where the EMR data is stored locally or a family practice clinic where the data is stored in the cloud. In either case, the realm is defined to be the small area where the data is stored and used. If a network outage occurs, the realm will continue to accept logs from the HIS or EMR and process them. If the HIS is cloud-based, a local backup copy may be relied upon. Any sharing of log summaries will wait until a network connection is re-established.

4.2.2 Multiple HIS in One Realm

If multiple HIS for the same clinic or hospital store their data in the same location then they may share a realm. In this case, a single Hyperledger Fabric blockchain instance may be used for multiple instances of software with data stored in the same location. This will require each HIS to write its information to its own channel, a subchain [69], on the blockchain. A smart contract will be used to amalgamate the data from the channels forming the base ABLOC entry, defined in Section 4.5.3, from each channel's log entries. Each system in a multi-system site must have an identification system to be able to associate patient data from multiple systems with the correct profile. It should not be relied upon that the identifiers that act as the primary key for identifying patients would be the same in all these systems. Instead, the salted hash of a patient's personal health identification number could be used to synchronize log records in a way that is not reversible but also should avoid collisions. By keeping the salt secured within the realm it will be possible to generate apparently random patient identifiers that could easily be mapped to actual patients by those who know the hash algorithm and the salt. Additionally, there must be an identifier added to map log hashes back to the system that generated them. Making the log identifier field, generated by the software, into a compound key with a software identifier would resolve this issue.

If a network outage occurs, the realm will continue to function normally. In the case of a cloud-based HIS, backup data would be relied upon to continue providing care and generating logs. Any sharing of log summaries will wait until a network connection is reestablished.

4.2.3 Single Realm Spanning Multiple Sites

There may also be instances of a single system implementation that is used across physical locations of a health authority. This may be a single HIS, lab, pharmacy, or radiology system or multiple systems that store data in the same location, as described above. This system or group of systems makes up its own ABLOC realm and, therefore, its own blockchain; if a physical location that uses this system is taken offline, then the rest of the realm should be able to continue processing logs normally. The offline location should also be able to continue to process logs by becoming a subrealm.

Assuming that there is a disaster preparedness plan for an internet outage, the ABLOC realms should be aligned with that plan. If the recovery plan relies on temporary backup links, the logs can be processed by the realm once the connection is set up. If the recovery plan relies on data stored at each physical site of the multi-site instance (and therefore logs could be generated locally), the offline site could temporarily become its own realm in the ABLOC network. This operates under the assumption that a patient can only receive treatment at one given site at any given time, therefore only having their record accessed at the offline location. When the offline site rejoins the network, the logs generated by the offline site should be witnessed by the rest of the realm and the ABLOC network as log summaries (Level2 and above). This will allow the site to act as its own realm when offline, be gossiped with directly when returning online, and then rejoin its normal realm when returning online.

4.3 Assumptions

In developing this solution, it is assumed that each provider and patient only have one identifier in an HIS, that realms store a copy or summary of their medical data, and that mobile phone data may have limits or caps.

1. It is assumed that each patient and each provider will have only one identifier in each HIS; these identifiers are not assumed to be unique between HIS. The identifier for one patient could overlap with the identifier for a patient in another HIS, but this does not mean that they are referring to the same patient/provider. It is also assumed that each site in the ABLOC system has appropriately set up and backed up its HIS data and servers. This includes creating backup copies of logs that were generated.

2. If a realm relies on a cloud-based HIS it is assumed that there will be a cache, summary, or backup of the HIS information stored in a way that makes it accessible in the case of a network outage. This would allow the site to continue to provide care in the event of an outage. This would be less of an issue with an on-premise HIS system, as all data is stored onsite. An onsite implementation is typically the implementation used for the open source EMR OSCAR [41, 70]. Details about more complex sites, such as sites that rely on multiple pieces of software or have cloud implementations of medical software, are discussed in Section 4.2.

3. As an alternative, some sites may plan to use cellular mobile hotspots to access the remote HIS data if there is an internet outage. Cellular data has more limited bandwidth, so a site may not want to use this to synchronize log summaries. Furthermore, this solution of using mobile data may also fail. For example, during the 2020 Rogers outage in Canada, when a large mobile operator had a long outage [21], traffic was shifted to other providers, which caused increased strain on their networks. This has the potential to cause more network failures.

4.4 Log Data Formats from Open Source EMRs

There is no standard format for what is logged in a HIS [52]. Some open-source EMRs were chosen from King and Williams' previous work to see what information and metadata are logged. King and Williams showed that on average, only about 12% of what should have been logged is actually logged [6, 52]; thus expanding what information can be incorporated is important for future improvements of ABLOC. It was also necessary to have an idea of what formats the data was in to ensure the tool would be compatible with some open-source EMRs. Three open-source EMR systems were set up to allow their logging strategies to be noted. It was expected that each

EMR should produce some logs.

4.4.0.1 Open Source Clinical Application Resource (OSCAR)

Open Source Clinical Application Resource, more commonly known as OSCAR, is an EMR that was created by McMaster University in the early 2000s; McMaster claims that OSCAR is one of the top EMRs in Canada [71]. As OSCAR is open source, there are many forks (parallel development branches) that are available for installation; often family practices hire an OSCAR service provider to install and maintain the EMR implementation for their practice.

The specific implementation reviewed was OSCAR v15 from the OpenOSP, an OSCAR service provider's repository [72]. It was reviewed in January 2023 and was a development instance that did not contain real patient data. OSCAR stores the log information in the MySQL database table called `log`. The logs could also be accessed on the EMR administrative page. The following fields are stored in the `log` table:

- Date and time of the logged event
- Event Action, such as read, edit, REST WS, log in
- Keywords
- IP address of the provider
- Provider Identifier
- Patient Identifier
- Data (changes etc)

Creating a new prescription for a patient results in a log event with the details being stored. Recording a patient's vitals, taken during an encounter, in the EMR results in the vitals entries not being logged, instead, it is just logged that the vital page was opened. Furthermore, when a new problem is added to a patient's problem list, the details of what was added may or may not be recorded in the log table, depending on the action. This aligned with King and Williams' work indicating that not all events are logged by EMRs [6, 52].

A sample log file was extracted from the database from this EMR instance. Af-

ter twelve minutes of testing, over a thousand log events were generated in the `log` database table. The actions taken included one clinician logging into the EMR, accessing different sections of patients' records, and updating them. The entire contents of the log file from this instance of OSCAR and multiple other instances of OSCAR, containing fabricated patient data, were archived to be used as a reference to generate logs for testing ABLOC. Screenshots of OSCAR and its logging interface are provided in A.1.

4.4.0.2 Open EMR

The review of Open EMR was also completed in January 2023. Open EMR is an open-source EMR that can be downloaded and used to keep medical records, write prescriptions, view medical images, and bill patients. Open EMR also has a patient portal option [73]. Open EMR is used across the world including in India, Pakistan, the Netherlands, and Sweden; it has been translated into local languages where it is used [74]. Open EMR is certified by the U.S. Government's Office of the National Coordinator and is a fully integrated records, practice management, scheduling, and electronic billing system [73].

Open EMR saves its log information to its MySQL database's `log` table. Some of the information stored in the `log` database table is encrypted with metadata stored in the file `log_comment_encrypt`. The following fields are stored in clear text:

- Date and time of the logged event
- Event (query, update, insert, delete, replace)
- Category that the event impacted (demographic data, billing, etc.)
- Provider Identifier and group association
- Patient Identifier

Additionally, the comments field in the log is encrypted. When using the log data viewer within the EMR, the table is displayed, and decrypted, showing that the comment field contains a database query. All of the login and other security logging and patient data access/update/delete are stored in the same place. Storing all the queries allows the EMR to log each CRUD action [52, 53]. Screenshots of Open EMR and its logging interface are provided in A.2.

4.4.0.3 Open MRS

Open MRS is an open-source medical record system that is modular and is designed for use in developing countries. Users are able to expand the system with many modules available for download; modules have full system access and can modify the reference version as needed to support the clinic [75]. The base version of Open MRS does not do any logging by default. There is an Access Logging Module, but it is no longer supported (when tested, it causes the EMR to crash, requiring the whole system to be reinstalled). The Logging Access Module documentation instructs the user to install the Usage Statistics module instead [76]. With the Usage Statistics plugin installed [77], some data was available. The results obtained by King and Williams relied on the old Access Logging Module [6] which is no longer viable [76]. The Usage Statistics module allows an administrator to view usage statistics based on roles, dates, and views to track access and updates to a patient's record. Within the logging tab, the module records when a patient's record is accessed and what actions were performed with small symbols; it does not record what specific vitals were accessed or how they were potentially modified. Furthermore, it does not record what notes were added to a patient's chart. The following information is available:

- Date and time of access
- The user who accessed the record
- The patient whose information was accessed
- Actions taken (denoted by box colours)

The same information is available in the database, but there is no more than what is presented on the logging page. A screenshot of the web interface is presented in A.3. Due to the lack of information logged, Open MRS did not have a significant influence on the information that will be stored in the ABLOC system.

4.5 ABLOC Data Storage

The ABLOC system increases the forensic reliability of logged data by allowing it to detect if logged information has been tampered with. Decisions were made on how to safely store patient log data, which impacted the design of the blockchain, and how to create summaries of the data.

4.5.1 DAG Blockchain Design

Each hospital or clinic, also known as a site, in a region will be responsible for managing blockchain infrastructure for a realm that hosts a HIS for the site. Each site will have at least one realm, a network where a piece of medical software is resident, and each realm will have its own blockchain. Some of the blocks on each blockchain will also be stored on other clinics' blockchains. The shared blocks will contain hashes of the logging information stored on a local clinic's blockchain, allowing the block to link back to the original blocks. By storing blocks from other clinics on the local blockchain, the local site acts as a witness of the blocks stored on the other blockchains. This design will allow a clinic to tailor the blockchain to its local requirements while meeting the standards to participate in the overall system. This allows each realm to run a blockchain that meets the size requirements depending on the piece(s) of software that is resident in that realm. If the majority of the blockchain infrastructure is hosted locally, this would also allow the realm's local blockchain to still process logs if the internet connection goes down. The local blockchain would continue to manage the logs submitted from the local HIS. When the internet connection returns, each site will be able to get a copy of the blocks that were generated while the location was offline. In addition, the offline location is able to resynchronize with the rest of the network and receive all of the blocks created by other sites.

4.5.2 How to Store Patient Data

Log data from a HIS is considered to be health data; it could contain measurements, like blood pressure, or other sensitive information and must be protected as such [62]. Two strategies were considered for storing the log data in the ABLOC system: one option was to store the log data directly on the local realm's blockchain, and the other was to store all the data within the HIS or other log storage system and place a hash of the data on the blockchain [26].

Storing log data on the local blockchain, and thus on the larger DAG blockchain, has the advantage of storing redundant copies of log data, which allows for easy recovery of the logs and makes the logs harder to modify. The anti-tamper blockchain properties would ensure that the logs remain intact [7, 12, 17]. However, this would require finding ways to ensure that the confidential medical information, that was

generated by the HIS, remains secure. This would lead to increases in complexity in terms of storage, encryption, and access to the blockchain. Data stored on the blockchain would have to be encrypted; this leads to challenges surrounding storing and sharing the encryption keys to access the data. The keys could be split and obtained from peers when needed [9], but this would increase the complexity of the system. Additionally, storing every logged event on the DAG blockchain would greatly increase the data storage requirements. This would slow the blockchain network in processing the logs, as there is more information to sort through. Many of these challenges can be avoided by storing the logs off of the blockchain.

Storing the logs off of the blockchain maintains the existing log storage infrastructure that is in use. A hash of each log entry would be stored on the blockchain to act as a witness of each log that is generated and stored. This would reduce the amount of information being stored on the blockchain; a hash with a fixed output would be chosen to make the amount of storage needed per logged event predictable and smaller. Smart contracts can be used to validate logs with hashes stored on the blockchain when validation is required; given a log ID, the hash of the log can be compared to what is currently available. If the hashes match it confirms that the log entry has not been tampered with. Furthermore, because only a hash of the data is being stored on the blockchain it is not necessary to encrypt the data; hash algorithms should be difficult to reverse and thus the patient data is protected [30]. Fan et al. and Moghaddam et al. both took this approach for their medical blockchain implementations [30, 18].

The purpose of the ABLOC system is not to create a backup of the logged data but rather to increase the forensic reliability and accountability by allowing logs to be verified as unchanged or modified logs to be flagged. The ABLOC system would allow a realm to forensically verify its main or backup copies of logs. Therefore, it is only necessary to store the hash of the logged event on the blockchain. Only storing the hash of the logs reduces the system complexity and allows the local logs to be summarized before they are shared with the wider ABLOC network.

4.5.3 Definitions of ABLOC System Levels

The ABLOC system has been designed with multiple levels of log summarizing and storage on blockchains in mind. By designing a system with multiple granularity

levels or hierarchies, it is possible to secure and summarize individual log entries. Additionally, having multiple levels reduces the burden on a realm storing multiple other realms' logs, as there will be less data contained in the summaries. Storing summaries also abstracts the individual logs, further reducing the likelihood of patient information being mined or interpreted from the metadata of the logs [78].

The levels in the ABLOC system are created using one of the following definitions, which form a recursive definition to create as many levels of logs as required for the implementation. The data is stored in a JSON format that allows it to be both human and machine-readable. The JSON allows the log data to be processed into summaries as the data is moved up the hierarchies. The base case and following levels combine to implement this.

4.5.3.1 Level1: Local Realm Logs

Level1 is the most granular level for the log blocks in the ABLOC system. These Level1 blocks are stored on the local blockchain only. A Level1 block contains the following information:

- Start Date: The date/time of the first log entry whose hash is stored in the block.
- End date: The date/time of the last log entry whose hash is stored in the block.
- Clinic or realm universally unique identifier (UUID): A globally unique identifier for each realm. It would be possible to use a UUID here or a hash of the clinic's public key in addition to a realm identifier.
- Log_Data: This is a dictionary where the keys are patient identifiers and the values are dictionaries. This allows the logs to be grouped together by patient identifier and allows each log's hash to be found through the unique log identifier.
  - Keys: Patient Identifiers - This is the identifier used to identify a patient in the HIS. It is unique in the HIS, but not globally. The value could be used by another realm or another piece of software for the same clinic, for another patient. It is expected that there would be overlaps when mapping globally unique identifier numbers to patient identities.

- Value: This is a dictionary where the keys are the log identifiers and the values are the hashes.
    - Keys: Log identifier - This is the unique identifier used to identify a log in a realm's HIS database.
    - Hash: This is a hash of the values in the original log. Each HIS implementation may be different and thus the implementing realm would have leniency in how this hash was implemented for their HIS. It is suggested that all fields are not stored in the blockchain. Instead, the fields will be concatenated and the resulting string run through either the SHA3-256 or SHA256 algorithm.

Multiple log entries from the same piece of software are grouped together to create a Level1 entry which is saved on the realm's blockchain. When a medical centre is deploying the ABLOC system, it is up to the centre to determine how frequently Level1 blocks should be uploaded to the blockchain. This number could be a fixed value or it could have two values: a normal priority and a high priority. Patient records of well-known individuals, where people may be curious about their personal information, and providers of concern could have heightened logging status, which would set the high priority flag. This would result in the Level1 blocks being generated faster than in normal priority mode. The time intervals for normal Level1 block generation and for the high priority, faster generation, must be chosen during the implementation. The more frequently the information is added to the blockchain, the fewer logs will be stored in a block, as fewer have accumulated, and more blocks will be gener-

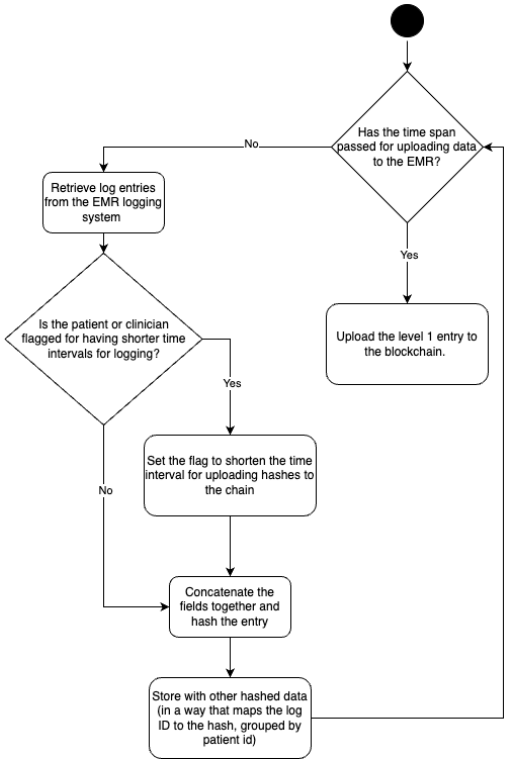

Figure 4.1: This activity diagram outlines the steps required to compile the HIS log entries for storage on the blockchain.

ated. It is important to balance the number of blocks generated by the blockchain and secure the data by storing it on the blockchain. This balance is needed to manage the overhead created by the block header - it would not be efficient to store a single log entry in a block. Depending on the particular blockchain choice there could be more metadata required for creating and storing the block than data stored in the block. Additionally, too many blocks being created could increase the confirmation time of blocks on the blockchain.

Once the time interval has passed, regardless of how many log entries have been gathered - it could be none - the Level1 block is uploaded to the local blockchain. A small, example Level1 block is provided below:


```
{
    "start_date": "4424-02-28 00:00:00",
    "end_date": "2023-02-28 00:00:00",
    "clinic_UUID": "1857587a-4bcd-496b-b3b6-2559826b3988",
    "log_data_in": {
        "system": {
            "1":
↪   "0b8554d6caaf9c7dea1ff6ab40bee7f98b01c95c35c707190f362e133aa72ab6",
            "2":
↪   "7e044da11a286a47770e17ccb0a4c1a4c00fab1b766379f4247262a3a171cd21",
            "3":
↪   "2bab6d3ece8b800bae6f75b55456d6d2816825d0a85303f02543371a2e4797f0",
            "4":
↪   "47d699f02d8109195b29b6fd9ef7d8675d9a8260a35c69bb26c7d1fe167f54e1"
        },
        "2": {
            "338":
↪   "11e3d5d6c25272440a12732431f03f4b7a0b0e4773fa187d58f62af124b4927b",
            "339":
↪   "6f712c7e2b58a4e12086ee3267ba61b356a375d8140562a79104fe55049f2dfe",
            "340":
↪   "907287981c66a97988b0775a134285a7767690348a849e601d57808c2c81bcbe",
            "341":
↪   "62e5ab1205edf36c6ce028331d203fe0037ff64275290b79db7de6fc6d3d6d05"
        }
    }
}
```


The process to create Level1 entries continues for as long as the HIS is running and

the logic to run this portion of the ABLOC system would be incorporated into the HIS software (ideally) or would pull entries from the HIS software's database for processing.

4.5.3.2 Level2 and up Blocks

LevelN blocks are summaries of the previous level's blocks that were generated within a certain time frame. These summary blocks compress the previous data so it can be shared across the block network. Depending on the implementation of ABLOC, multiple summary layers may be required. For example, realms could share Level2 blocks within a smaller geographical area and then some of these Level2 blocks would be used to create a Level3 block, which could be shared at the health authority level. An overview of this process is shown in Figure 4.2. Level4 blocks may be shared at the provincial level, etc.

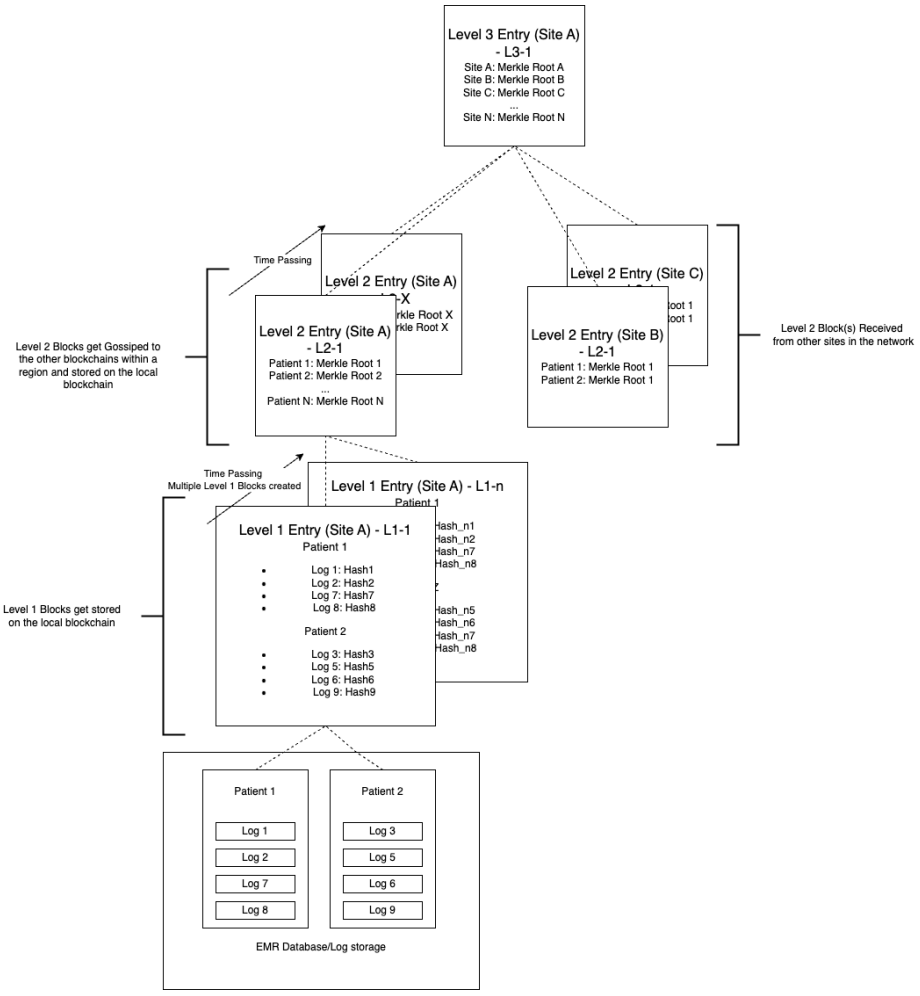

Figure 4.2: This diagram visualizes the summarization of blockchain log entries, including those from other realms, to form the next level summary. This illustrates summaries from Level1 to Level2 and from Level 2 to Level3

Each realm will be required to generate summaries for the data stored on the realm. These will then be shared with the other peers in the network. Depending on the number of levels in the hierarchy, there could be sites that store certain levels without producing the levels below, or that choose to not store or generate levels above Level3. This will allow health authorities or governments to store summaries of the log information generated by each site and remove the burden of storing the summaries created in the higher hierarchies from smaller realms.

Multiple blocks from the previous level are combined to form a block for the next level in the hierarchy, which is generated every LevelN time interval. The oldest start

time of the previous level blocks received in LevelN time interval is set as the start time, and the most recent end time within the interval is set as the end time. For each level in the hierarchy, an additional identifier must be added to the lower levels, to identify grouping. For example, in a Level2 block, there will be multiple Level1 identifiers being present in the Level2 summary. This is used to identify the blocks from which the summary was derived.

In each level, like data must be grouped together. Each entry from the previous level must be added to a Merkle tree for each sub-item (i.e. for each patient's list of log entries within a given time frame) resulting in a Merkle root for that item. This Merkle root is then added to the LevelN block, along with the identifier of what was summarized. The strategy for creating the Merkle tree must be standardized in the ABLOC network implementation.

This process of creating summaries can be used in multiple layers; it is suggested that the layers could be associated with the following abstractions:

- Level1: Logs from HIS - Stored on the realm's local blockchain
- Level2: Each patient has a Merkle root for the hashes of the logs in the block. The realm summarizes the logs that have been generated by its software. It is suggested that this could occur every 30 minutes to strike a balance between generating too many summaries, increasing storage and communication requirements, and summaries to other clinics.
- Level3: Each realm has a Merkle root for the hashes of the logs in the block. The realm summarizes the logs it has received from other realms. It is suggested that this could occur every four to eight hours.
- Level4: Each region has a Merkle root for the hashes of the logs generated that the block covers. The realm summarizes the logs it has received from the entire region. It is suggested that this occurs every one or two days.
- Level5: Continues if desired...

To create a summary block, the blocks from the lower level are collected over a time period. Once the time interval has passed, regardless of how many log entries have accumulated on the blockchain, the summary block is generated and saved back to the blockchain. The summary block is then available to be gossiped to other sites in the

network. This summary generation could be done with an external program with the lower-level data being pulled from the blockchain, summarized, and then re-uploaded to the blockchain, or it could be generated by a smart contract. It was decided that this generation should be done by a smart contract to ensure that realms that pull data from another realm's blockchain have the understanding that it was generated by a standardized smart contract. This would ensure that the log summarization is done in the same way throughout the ABLOC network and would allow summaries to be pulled from the blockchain by regulators, as opposed to trusting summaries that were pushed to the blockchain by the realm. An example Level 2 block, summarizing the Level 1 block above, is provided:


```
{
    "Level": "Level2",
    "Clinic_UUID": "1857587a-4bcd-496b-b3b6-2559826b3988",
    "Start Time": "2023-02-28 00:00:00",
    "End Time": "2023-02-28 00:30:00",
    "Blocks Hashed": {
        "1857587a-4bcd-496b-b3b6-2559826b3988--Level1--2023-02-28 00:
        00:00--2023-02-28 00:05:00":
↪   "c6f72df842174ba4be4e8665dc85bba8052300933562da0ea5fb81c03e09b0a3"
    },
    "Merkle Map": {
        "system":
↪   "e42c509065051faa30ce33b1a7d068346e3a58c0d13b4092c8de42ce9b8a937c",
        "2": "ea6aa312a59c0c0f5ff11b1c3649dab627e08d92b41aa9cde3a482761fbe7797"
    },
    "Patient To LogID List": {
        "system": ["1","2","3","4"],
        "2": ["338", "339", "340", "341"]
    },

}
```


Beyond Level2, the Merkle Map field will map other identifiers to their appropriate Merkle root and the `Patient to LogID List` field will also change to match the entity that was summarized in the Merkle map to the identifiers that the map covers. This allows for the reconstruction of the summaries if needed. For example, a Level3 block will contain an entry for each realm as a key-value pair (realm identifier, Merkle root) in the Merkle map. The last field will need to map the realm identifiers to

the patient identifiers, whose hashes were used in the Merkle map, to allow for its reconstruction.

For a successful ABLOC implementation, it is only required that a group of realms, for different clinics or hospitals, generate and share up to Level2 summaries allowing the realms to witness the summaries produced by other realms. Having multiple hierarchies technically results in redundant ways to verify a log's authenticity; it is anticipated that the lowest level required to verify logs would be used, which would reduce the computational complexity of hashing multiple logs unnecessarily. If there is a range of logs for a patient, realm, or region that all need to be verified it makes more sense to use the relevant summary level for the logs and compute the Merkle tree(s) required to verify all the logs with the fewest Merkle roots. This will authenticate all of the logs within the tree with the single Merkle root hash. If the hashes do not match, it would be possible to drill down to lower hierarchy levels (of the tree and other summaries) to discover which log(s) have been modified. The summaries facilitate the transfer of log data between the hierarchies in the ABLOC system; as the entries move up the hierarchies get summarized. The summary blocks get larger in scope, in terms of time the logs are covered and the individual logs each block stores verification information for, as the levels of the blockchain increase.

The time interval for the generation of LevelN blocks is decided at the time of implementation and should be standardized within a network. As the levels in the hierarchy increase, blocks should be generated less frequently. The minimum required number of levels for an implementation is two, where each realm generates Level1 and Level2 blocks. This assumes that every realm will store every other realm's logs and that multiple organizations are participating.

4.6 ABLOC Inter-Realm Communication – Gossip Protocol

An inter-realm blockchain communication algorithm is required to allow one realm to receive the blocks generated by another. This will facilitate the lateral movement of log data amongst peers. This algorithm must be tolerant of realms going offline. The gossip algorithm that was chosen is a modified version of the one discussed in the paper Vegvisir: A Partition-Tolerant Blockchain for the Internet of Things [2].

This gossip protocol was chosen as the starting point because it uses the regular internet infrastructure, stores messages and does not delete them once all the nodes are notified of them, and does not require beaconing or construction of new views for every iteration of the gossip protocol, like the alternative protocols discussed in Section 3.5. The modified protocol will be able to resynchronize with the other realm in the case of an internet outage.

- A copy of every LevelN block sent by the other realms [23]. The shared blocks will be stored on the blockchain.
- The frontier set [23]: this is a set of realms in the network. This was modified to store a realm's access URL, most recent connection time, and most recent message identifier, as opposed to the entire most recent message, as was proposed in Karlsson's work [23]. This way the location of each realm is also available. If the network is of size N, then there are N possible entries in the frontier set. An empty message identifier and date, from the 1300s, are used as placeholders if no blocks have been received from a realm.

When the gossip protocol is running, a smart contract has the API calls necessary to respond to gossip requests. Each realm also runs a client that connects to other realms' listening blockchains, to request gossip from peers. To start a gossip interaction, the connecting node, n_c, requests the frontier set and this is sent to n_c by the responding realm's, n_r, blockchain server. The site n_c then compares the frontier set that it received, f_r, to its own frontier set, f. For a given node, n_i, in f, if there is a newer log summary, l_{new}, in f_r then all summaries between the known log summaries, l_{known}, and l_{new}, sent by n_i, are requested by n_c from n_r. These log summaries are processed and added to n_c's blockchain. Additionally, n_c's frontier set f's entry for n_i is updated to match the new entries on the blockchain. Once this process is completed for all sites, n_c notes the connection time; n_c will not reconnect to n_r for a period of time, avoiding unnecessary traffic, as it is unlikely that n_r will have had many changes in a short time period.

Instead, n_c randomly chooses another site in the network to gossip with. Before connecting to the new site, n_c ensures that the new node has not been gossiped with recently to avoid redundant traffic. Within this choosing process, there is a pause to allow the blockchain to manage other requests. If a node n_a attempts to connect to another node and a connection can not be established, resulting in a timeout, then n_a

will choose another realm to connect to. This continues until a successful connection is made and gossip is exchanged.

When a node, n, sends a log summary, it saves the summary to the blockchain and updates its frontier set entry, for itself, to update the latest summary saved to the blockchain. When a node, n_g, tries to gossip with n it is able to retrieve a copy of the summary through the mechanism described above. This allows sent messages to be shared with the entire network.

In order to expand the network, it is necessary to be able to add new realms to the network; it would be best to achieve this through the gossip protocol. When a node n is iterating through a frontier set, FS_r, received from node n_r to check for updates to summaries, n compares its Frontier Set, FS, to FS_r. If n comes across a realm, n_n, such that $n_n \notin FS$ and $n_n \in FS_r$, then n should add n_n to its frontier set. Before finalizing, it is suggested that the administrators of site n contact and verify the administrators responsible for realm n_n out of band to confirm their identity. Once verification is complete, the new site will be added to the blockchain. The first realm to add a new node n_n would be considered a sponsoring site.

This algorithm provides the functionality of sharing information between nodes and is similar to those discussed in the literature summary Section, where some of the gossip algorithms use the probabilistic delivery reliability, relying on the epidemic spread of messages to every node [65, 66, 67]. An overview of the protocol is shown in the activity diagram, Figure 4.3 and is summarized in the algorithm formalization below. Requests to sites that are offline will go unanswered and the requesting site will eventually timeout and a new node will be chosen for gossiping. This protocol would be best implemented to interact with a blockchain with stateless connections.

Algorithm 1 Get Gossip Algorthim

Require: $|Local_{frontierset}| > 1$

1: **while** the realm has network connectivity **do**
2: Ensure local $Local_{FrontierSet}$ is up to date
3: Choose a random realm, R from $Local_{frontierset}$
4: **if** R is the local realm **then**
5: Go to 2.
6: **else**
7: **if** R has been contacted recently **then**
8: Go to 2.
9: **else**
10: Request for R's frontier set, $R_{frontierset}$
11: **if** The request times out **then**
12: Go to 2.
13: **else**
14: **for** r_i in $R_{frontierset}$ **do**
15: **if** $r_i \in Local_{FrontierSet}$ **then**
16: **if** r_i's summaries are ahead of the local blockchain **then**
17: Request all r_i's summaries $\notin$ the local realm from R
18: **if** The request times out **then**
19: Go to 2.
20: **else**
21: Save each summary to the blockchain
22: **end if**
23: **end if**
24: **else**
25: Add realm to $Local_{FrontierSet}$
26: **end if**
27: **end for**
28: **end if**
29: Update the connection time for R in $Local_{FrontierSet}$
30: Wait set time period
31: Go to 2
32: **end if**
33: **end if**
34: **end while**

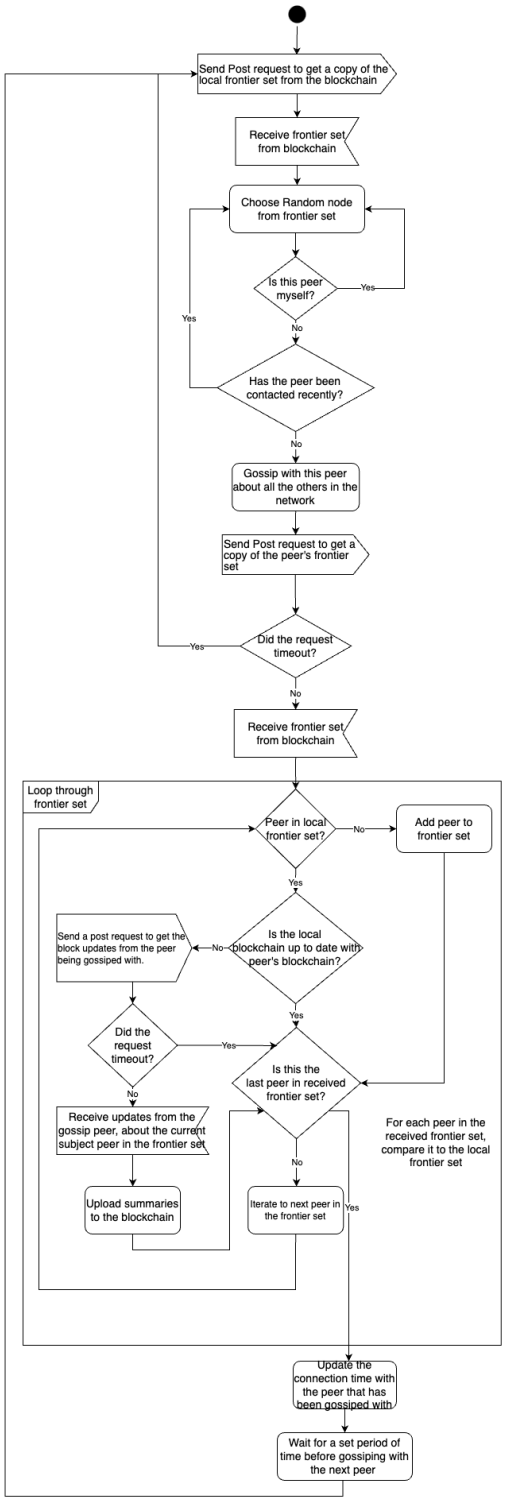

Figure 4.3: This diagram outlines the process for getting the summaries from other nodes in the ABLOC network.

This inter-chain communication protocol should be run for each level of hierarchy that the realm is part of after Level2. Therefore, there must be a frontier set or subset, and gossip APIs for each hierarchy level or entries for each level. Having a sub-frontier set for each level makes it easy to add additional levels of hierarchy to the system as needed.

4.7 Smart Contract

A smart contract is exectuable code that is stored on the blockchain. It is able to modify and interact with the blockchin in a defined way. A smart contract is needed in the ABLOC system to receive the logs that are uploaded to the blockchain and to process them in the way that is defined by the network. The smart contract will be responsible for receiving new Level1 blocks to store on the blockchain, will generate the summaries required by the gossip protocol, will store the realm's frontier set, and will respond to the gossip requests placed by other clinics in the network. These functions should be available in the form of an API. The smart contract will implement the following functionality, as shown in figure 4.4 and described below:

- Accept and store Level1 block data from the HIS software.
- Store the realms in the ABLOC network, for the appropriate levels, in a frontier set. A copy of a realm's frontier set will be available to peers on the network as part of the gossip protocol.
- Accept summaries, generated by other realms, from the gossiping in the ABLOC network and store them on the blockchain.
- Respond to requests for summaries, generated by itself or peers on the network, in response to gossip requests.
- Automatically create summaries of the logs uploaded to the local blockchain and update the frontier set for the local realm, allowing these messages to be shared with other peers.
- Disallow duplicate data to be stored on the blockchain. Each data entry on the chain should have a unique identifier and there can not be two entries, on the same level, that contain the same information about the same node.

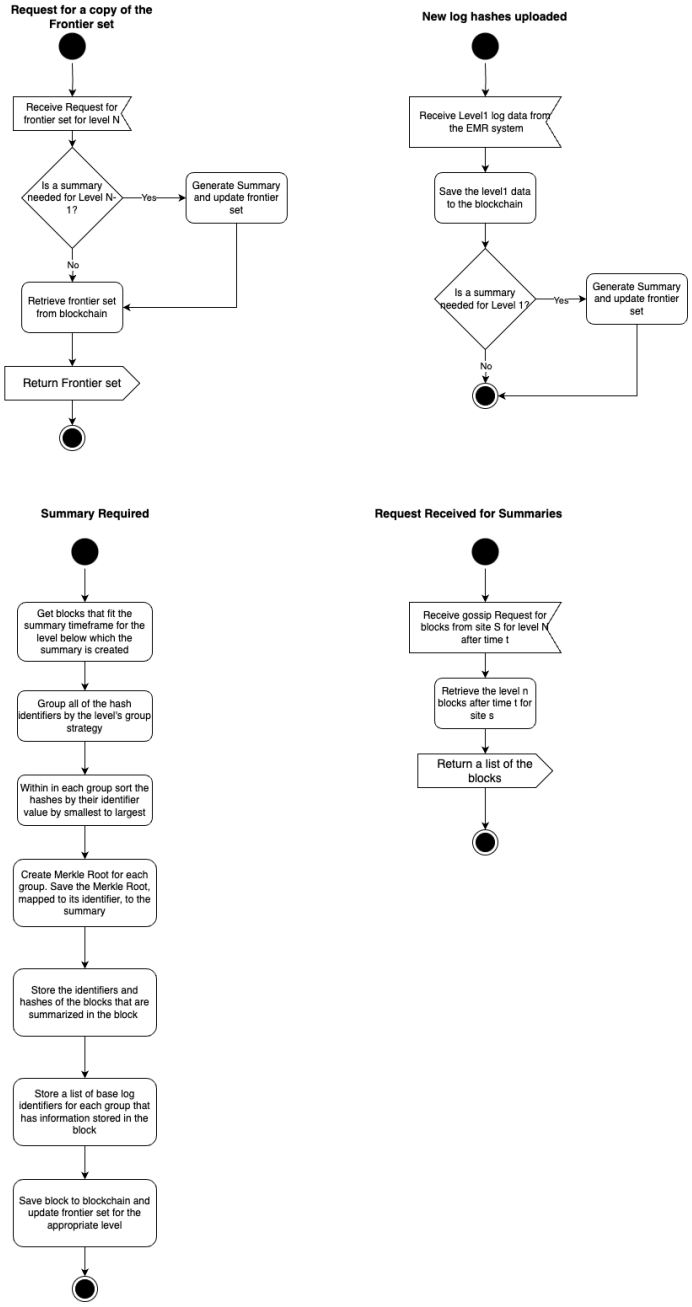

Figure 4.4: Smart Contract Primary Functions - Activity Diagram

4.7.1 Updates to the Smart Contract

Each realm is responsible for maintaining its own blockchain and, therefore, its own smart contract. However, smart contracts should be standardized to the wider ABLOC network. Updates to the smart contract should not prevent realms that

have not updated their contracts from working with the ABLOC system. This gives a larger window to allow for updates to the blockchain and a fallback if updates fail. Changes can be made and new features added, as long as the basic functions are still compatible with realms that have not updated their smart contract. Changes to the smart contract should be approved by the wider network. The smart contract could be signed by each realm's private keys to indicate approval. A consensus algorithm could also be employed to ensure that the updates to the smart contract are approved before it is deployed.

4.8 Minimum Network Size

The minimum network size needs to be considered twice for the proposed ABLOC system. In general, the more nodes that a network has the more nodes have to come to a consensus about items stored on the blockchain. Most research is concerned with the throughput of blockchains [79, 80, 81] as opposed to how small a network can be. The implementers will have to decide on the network size that is required for an ABLOC implementation depending on the application of the ABLOC system. The minimum network size needs to be considered at the local blockchain level (how many users are part of the system) and at the wider ABLOC system level during implementation.

There have been EMR blockchain implementations proposed where the equivalent of the local network only had two peers in the network - one administrator node, where smart contracts were installed, and another peer who would invoke the smart contract. The authors also stated that the number of peers in the network should grow to match the demands placed on it by the users, implying that more nodes would be better able to process the requests and data being sent to the blockchain. They also showed that the number of peers was proportional to the network's performance [79]. Therefore, it would be theoretically possible to have an ABLOC network that contains two realms. However, with a small number of realms, it would be easier for malicious actors in each realm to collude with one another, in a malicious manner, to exert control over the blockchain.

It has been discussed that if a user participates in only one smart contract on a blockchain, then the transactions processed by the smart contract can be confirmed in isolation without concern about double spending [81]. This idea can be applied to

the ABLOC system, as the logs are being generated and uploaded to a single smart contract - there is no risk of double spending because the blockchain is just witnessing the creation of the logs. Therefore, if a single smart contract is used to add the entries to the ABLOC network it should be able to process these summaries and respond to gossip. Any extra nodes on the network will witness and approve the blocks. There is a risk of falsification of the logs, as with a smaller network it would be easier to falsify the network [12, 27, 40], but broadcasting the summaries to the wider ABLOC DAG should reduce this risk.

In summary, small blockchain networks appear to be feasible [79, 81], but having more nodes in the local blockchain realm increases the reliability of the logs when a realm is offline. This is intuitive, as more nodes are witnessing the new log entries. With more witnesses, there are more people that a malicious actor would have to either collaborate with or fool to falsify the blockchain. This logic also applies to the wider ABLOC network: if more realms are available to witness the summaries shared by a given realm, it is less likely that the summaries would be able to be modified from what was originally gossiped to the other realms. It is critical to ensure that the summaries are generated using the correct information.

4.9 Retiring Realms

Realms that wish to retire or be removed from the ABLOC network should be able to do so. The realm will be able to stop running its local blockchain, assuming other realms are also storing the summaries stored by the retiring realm. However, it would not be possible to remove the summaries of the retiring realm from the blockchains that are hosted by their peers; these summary blocks make up integral parts of their blockchains. The realm that is retiring will appear to be offline and no longer respond to requests to gossip from other realms. Eventually, the realm attempting to gossip with the retiring realm would time out and choose another node to gossip with. An optimization to this is presented in Section 5.6. The information stored on the retiring realm's blockchain should be kept for as long as required to satisfy documentation laws. Once that time has passed it may be deleted. Realms leaving the ABLOC system may impact the network's ability to increase accountability. There needs to be a minimum number of realms for the system to be able to continue operating.

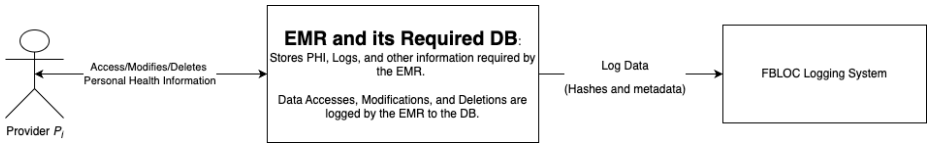

Figure 4.5: This Data Flow Diagram 0 shows how the system should act from the Clincian's point of view.

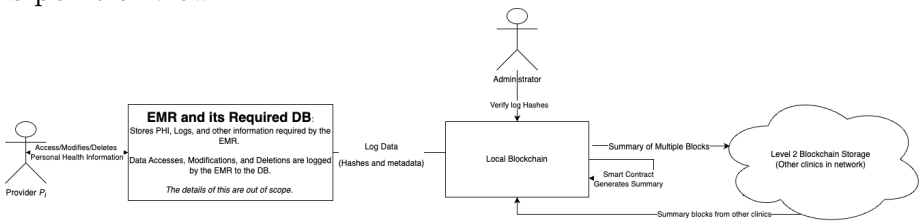

Figure 4.6: This Data Flow Diagram 1 shows how the system should act from the Clincian's and from a system admin's points of view.

4.10 System Data flow

The ABLOC system should integrate with existing HIS tools without impacting its performance. The end users should be able to continue to use the system without any discernible changes; a clinician assessing the HIS system should generate logging events, which should be processed and sent to the ABLOC blockchain. This is represented by the Data Flow Diagram (DFD) Level 0 in Figure 4.5.

The log data follows the process described above, resulting in logs that are stored on the local realm's blockchain. This data is then summarized into Level2 blocks by the smart contract and is available for sharing with the rest of the ABLOC network. The data could then optionally go through more levels of summary creation and gossip with other peers in the network, depending on the number of levels of hierarchy that are in use. This is presented in the DFD1, see Figure 4.6.

4.11 ABLOC Implementation Requirements

The following requirements are for implementing an instance of the ABLOC system. For this implementation, it is assumed that each healthcare site within the wider ABLOC network is running a single, locally hosted, piece of medical software. Each site will be made up of a single realm, making this the simplest implementation of the ABLOC system. These implementation requirements summarise the above

descriptions and provide a reference for implementing the ABLOC system; they were generated from the prototype implemented for this thesis. These requirements can be used for further research or modification of the ABLOC system.

4.11.1 Blockchain

Blockchain software will be chosen for the site that allows a custom smart contract and local blockchain to be created. This should be a private blockchain that only the realm or hospital and the remainder of the ABLOC network can use, thus creating a consortium blockchain; it is expected that other clinics joining the network would be semi-trusted peers. HyperLedger Fabric is recommended as the blockchain choice, as it allows for the sub-chains discussed in Section 4.2, is open source, and has plug-able consensus algorithms, giving it flexibility [39, 69]. Depending on the size of the realm, the blockchain network size may vary, and the flexibility of Hyperledger Fabric's plug-able consensus algorithm will let the implementer choose the most appropriate one. When considering consensus algorithms, proof of work is not recommended due to its inefficiencies, as discussed in Section 3.1.3.1. It is anticipated that each computer in a realm would be a member of the blockchain to witness/process the blocks being generated (and act as a member of the network). The blockchain must be able to run smart contracts that have a REST API.

It is expected that log block objects stored on the blockchain will use the following ID scheme: RealmIdentifier–BlockLevel–StartTime–EndTime. For example, e36543ad-e25e-4b3d-b011-f78811b23484–Level1–2023-01-01 00:00:00–2023-01-01 00:05:00. The realm may be replaced with a different value if a site goes offline, as detailed in 4.2.

Smart Contract A smart contract will be required to implement the functionality of the ABLOC system. This smart contract shall receive Level1 Blocks from the HIS software (or middle-ware generator), generate summaries of the blocks to as many levels as needed, and respond to gossip queries. This should occur from a single URL that can be referenced in the frontier set of each node. The functions of this smart contract should only be accessible to members of the ABLOC network; some functions should only be accessible to the local realm. This smart contract should have standardized functions that all the other realms in the ABLOC network are able to use to connect and access specific information on the local realm's blockchain. This allows the URLs to be standardized for the API.

Object Storage on the Blockchain The following objects should have structures or classes defined to allow them to be used in the smart contract.

- **Level1 Block**: This object stores the Level1 block on the blockchain. It requires the following fields:
  - Level: String.
  - RealmID: This is the clinic's/realm's identifier on the ABLOC network.
  - StartTime: String representing the block start date and time (this should be in the format YYYY-MM-DD HH:mm:ss.ssssss or YYYY-MM-DD HH:mm:ss)
  - EndTime: String representing the block end date and time (this should be in the format YYYY-MM-DD HH:mm:ss.ssssss or YYYY-MM-DD HH:mm:ss)
  - Data: A dictionary that maps a patient identifier (or system placeholder) to a dictionary containing log identifiers and log hashes.
- **Level2 Block**: This object stores the Level2 block on the blockchain. It requires the following fields:
  - Level: String.
  - RealmID: This is the clinic's/realm's identifier on the ABLOC network.
  - StartTime: String representing the block start date and time (this should be in the format YYYY-MM-DD HH:mm:ss.ssssss or YYYY-MM-DD HH:mm:ss)
  - EndTime: String representing the block end date and time (this should be in the format YYYY-MM-DD HH:mm:ss.ssssss or YYYY-MM-DD HH:mm:ss)
  - MerkleData: A dictionary that maps a patient identifier (or system placeholder) to the Merkle root of all the patient's logs contained in Level1 blocks between the start and end times.
  - HashedBlock: A dictionary that maps a block identifier to a hash of each Level1 block that was summarized in the Level2 block.
  - PatientToLogIDs: A dictionary that tracks the patient identifiers and maps them to lists of log identifiers that are represented by the Merkle root contained in the MerkleData.

- **LevelN Block**: This object stores the LevelN blocks that may be included. It should follow the format of the block above.
- **NetworkNode**: This is the base object to represent a realm in the smart contract's frontier set. It requires the following fields:
  - NodeID: String. This is the realm's identifier on the ABLOC network.
  - NodeUrl: String. This is the URL that is used to access the API for the realm (with the addition of the function that is being accessed).
  - LastConnection: String representing when the node has been successfully gossiped with (this should be in the format YYYY-MM-DD HH:mm:ss.ssssss or YYYY-MM-DD HH:mm:ss)
- **Frontier Set**: This is the object that will store the NetworkNode objects. It should contain a dictionary that maps a NodeID to the node in the frontier set. Alternatively, this could be stored as a dictionary.

Functions - Realm Access Only The following functions should be available to the local realm and not the wider ABLOC network; these functions are for uploading new Level1 log blocks to the network, retrieving Level1 log information, and managing the frontier set for the local realm. They could also be used to verify logs at the local level.

- **AddNodeToFrontierSet**: This function is used to add a new network node object and add it to the realm's frontier set. This will allow it to gossip with that realm. This function should accept the following as parameters, through a POST API call in the JSON format:
  - **RealmID**, type string. This is the new realm's ID number and is determined by that node. It should be a globally unique value (hash of a public key or a UUID).
  - **NodeURL**, type string. This is the base URL to access the smart contract's API for the node. This can be used to get the frontier set, summaries, or other information from the realm.

  This function should not allow duplicate values to be written to the frontier

set. If a value is already in the frontier set, a new copy of it will overwrite the previous entry.

- **CreateLevel1Block**: This function is used by the HIS to create a new entry onto the ABLOC blockchain. This function should accept the following as parameters, through a POST API call in the JSON format:
  - **RealmUUID**, type string. This is the realm's identifier in the ABLOC network.
  - **EndDate**, type string. This is the end date and time of the Level1 block that is being uploaded to ABLOC. This should be in the format YYYY-MM-DD HH:mm:ss.ssssss or YYYY-MM-DD HH:mm:ss.
  - **StartDate**, type string. This is the start date and time of the Level1 block that is being uploaded to ABLOC. This should be in the format YYYY-MM-DD HH:mm:ss.ssssss or YYYY-MM-DD HH:mm:ss.
  - **LogDataIn**, type array of dictionaries. The dictionary will use the patient identifier or a placeholder for system logs, and a dictionary for each value. The value dictionary will hold a mapping of log identifier to log hash.

  This function should save the values passed to it in a Level1 object and save them to the blockchain using the identifying scheme discussed above. This function should not allow duplicate blocks to be written to the blockchain. If the block already exists, the blockchain should return an error.

- **GetLevel1Blocks**: This function gets all of the Level1 log entries created after a given time and before another given time, and returns them in an array. This function should accept the following as parameters, through a POST API call in the JSON format:
  - **RealmUUID**, type string. This is the realm's identifier in the ABLOC network.
  - **EndDate**, type string. This is the end date and time of the Level1 block that is being uploaded to ABLOC. This should be in the format YYYY-MM-DD HH:mm:ss.ssssss.
  - **StartDate**, type string. This is the start date and time of the Level1

block that is being uploaded to ABLOC. This should be in the format YYYY-MM-DD HH:mm:ss.ssssss.

- **GetLogHashWithLogID**: This function returns a log entry's hash, given a specific log identifier from the local realm's logs. This function should accept the following as a parameter, through a POST API call in the JSON format:
  - **LogID**, type string. This is the identifier of the log from the HIS' logging database.
- **Level*N*GossipFromNode**: This function is used by the gossip server to upload the blocks from the Level*N* gossip server. It should update the Level*N* frontier set to denote the latest log from each node and the last time the realm that the gossip came from was contacted. This function should accept the following as parameters, through a POST API call in the JSON format:
  - **RealmUUID**, type string. This is the identifier of the realm, in the ABLOC network, that the gossip was received from. **NOTE:** This is not the subject of the gossip, but rather where it came from, which may, or may not be the same realm.
  - **Contact_time**, type string. This is the date and time that the realm was contacted for gossip. This should be in the format YYYY-MM-DD HH:mm:ss.ssssss or YYYY-MM-DD HH:mm:ss.
  - **Start**, type string. This is the start date and time of the Level*N* block that is being uploaded to ABLOC from the realm. This should be in the format YYYY-MM-DD HH:mm:ss.ssssss or YYYY-MM-DD HH:mm:ss.
  - **End**, type string. This is the end date and time of the Level*N* block that is being uploaded to ABLOC from the realm. This should be in the format YYYY-MM-DD HH:mm:ss.ssssss or YYYY-MM-DD HH:mm:ss.
  - **MerkleData**, type array of dictionaries, where each dictionary maps a string (the patient identifier) to a string (the Merkle root of the Level1 hashes).
  - **HashedBlocks**, type array of dictionaries, where each dictionary maps a block identifier to the hash of each Level*N-1* block summarized in the

Level$N$ block.

- **PatientLogIDs**, type array of dictionaries, where each dictionary maps the patient identifier to an array containing the log identifiers that the block covers.
- **Level**, type string. This is the level N that the blocks are associated with.

Functions - Wider ABLOC Access The following functions are required by the ABLOC network to support the gossiping between realms:

- **GetFrontierSetLevel*N***: A function will be implemented for every summary level that is available in the ABLOC network, that the realm is participating in. There will be a frontier set, or subset, for each level (Level2, Level3, etc.). This function does not take arguments and returns the frontier set for a given node at a given level. This function should be triggered by a POST API call in the JSON format.
- **GetLevel*N*BlocksAfterTime**: This function gets all of the Level$N$ log entries created after a given time and before another given time, and returns them in an array. This function should accept the following as parameters, through a POST API call in the JSON format:
  - **Level**, type string. This is the level of blocks that are being requested. This is restricted to levels that are available in the ABLOC implementation and can not access Level1 entries.
  - **RealmID**, type string. This is the identifier of the realm whose blocks should be returned.
  - **Time**, type string. This is the date and time that the required blocks come after. This should be in the format YYYY-MM-DD HH:mm:ss.ssssss or YYYY-MM-DD HH:mm:ss.
- **GetLevel*N*BlocksForRealm**: This function gets all of the Level2 log entries created after a given time and before another given time and returns them in an array. This function should accept the following as parameters, through a POST API call.

Private Blockchain Functions - Smart contract access only These functions are required to support operations in the ABLOC network and must be only accessible by the smart contract. The first letter of these functions are not capitalized to conform to the Golang programming practice for public and private functions.

- **level1SummaryNeeded**: Determines if enough time has passed and a new summary is needed to summarise the Level1 blocks created in the interim is needed. This function is triggered every time a new block is uploaded to the blockchain. This function returns a Boolean.
- **levelNSummaryNeeded**: Determines if enough time has passed and a new summary is needed to summarise the LevelN blocks created in the interim is needed. This function is triggered every time a new block is uploaded to the blockchain. This function returns a Boolean.
- **createLevel2Entry**: This function is triggered if the result of `level1SummaryNeeded` is true. This function will determine what blocks need to be included in the summary, generate it, save it to the blockchain, and update the frontier set appropriately.
- **createLevelNEntry**: This function is triggered if the result of `levelNSummaryNeeded` is true. This function will determine what blocks need to be included in the summary, generate it, save it to the blockchain, and update the frontier set appropriately.

Other helper functions may be required but are left up to the implementer to design. These functions could include but are not limited to, functions to store the frontier set on the blockchain, to parse dates, and to generate identifiers for blocks. Decisions about these functions will be left to the implementer.

Optional Functions for Debugging

- **GetAllLevel1**: This function maybe used to get all of the Level1 blocks produced by the realm. This returns every hashed log. This function takes no parameters and returns an array of Level1 block representations in JSON; this function responds to a POST request on the API. The same logic can be used to get the blocks for any level. It is not recommended to include this in a production system.

4.11.2 HIS to ABLOC interface

It is recommended that the ABLOC logging information be built into the HIS' logging module. The logs generated from the HIS will be stored in the same manner that they are now but the interface will also hash the log entry and gather and upload the logs, in the format described above, to create a Level1 block. This interface will determine the frequency at which a Level1 block needs to be generated based on the logging priority (if implemented). The logging priority will be set to high priority mode if a user is generating logs, or if a patient of concern is having their record accessed. Increasing the logging priority decreases the amount of time to create a Level1 block, reducing the window of time for a malicious actor to modify the logs without being discovered.

4.11.3 Gossip Protocol

The gossip protocol, as described above, should implement the following requirements. These requirements rely on the smart contract having functions that allow it to share the local realm's frontier set, respond to gossip requests from peers, accept gossip to be uploaded from the gossip retrieval portion of the protocol, and update the frontier set appropriately. This should complete the distribution portion of the gossip protocol.

The portion of the gossip protocol that retrieves gossip from other realms must follow the following process for gossip retrieval:

1. Connect to the realm's local blockchain and obtain an updated copy of the local frontier set by using the `GetFrontierSetLevelN` function.

2. Choose a random peer, p, from the frontier set and ensure that it has not been gossiped with recently.

3. Connect to p and obtain a copy of its frontier set.

4. Iterate through the frontier set. If updates are required, request them from p using the `GetLevelN BlocksAfterTime` with the appropriate level, realm identifier, and time after which the updates are needed. Alternatively, if this is the first time receiving gossip for a realm the the function `GetLevelN BlocksForRealm` may be used.

5. For each block received in response to the request made in item 4, upload the block to the local blockchain using `LevelNGossipFromNode` function.

The overall system can be summarized in the sequence diagram shown in Figure 4.7.

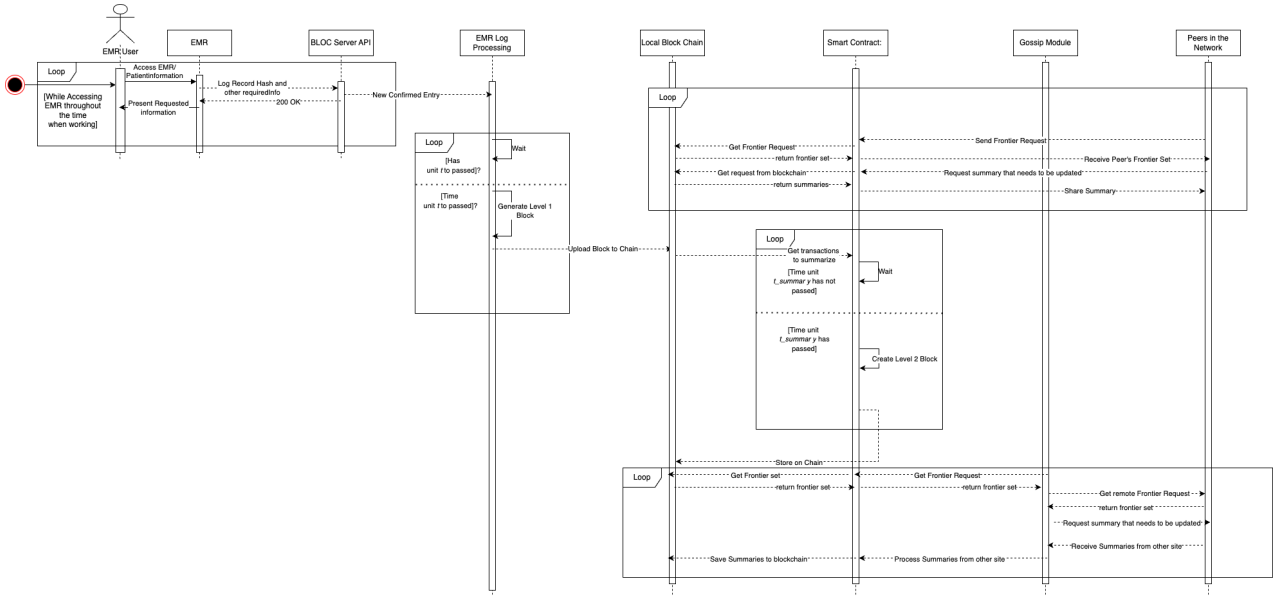

Figure 4.7: This sequence diagram provides an overview of all the functions in the ABLOC system.

Chapter 5

Evaluation and Analysis of ABLOC

5.1 Introduction

In order to evaluate the proposed ABLOC system, to ensure that it allows blockchain logging to be applied to HIS even when a realm, hospital, or clinic is offline. A theoretical evaluation was completed and a prototype was created. The theoretical evaluation and prototype assumed that a group of clinics in a small area will be using the ABLOC system, with a single level of summaries being generated. This sample implementation has one level of summarisation, though it could be modified to accommodate more. The explanations in the previous chapter provided a base for implementing the prototype. These influenced the blockchain choice, the HIS interface, and the gossip protocol. The evaluation led to some changes that were incorporated into Section 4.11 and are also discussed here.

5.2 Theoretical Evaluation

The theoretical evaluation reviews the ABLOC system from a privacy perspective, presents a case of why it increases the accountability and reliability of logs, reviews the impacts of network sharding, and finally reviews the scalability of the ABLOC system from a theoretical perspective.

5.2.1 Privacy Impact

The log data that is generated by each HIS logging mechanism is considered to be medical data; it could contain patient identifiers and other sensitive medical data [62]. The ABLOC system stores a hash of the log data generated, in addition to the patient identifier from the HIS' database and the approximate time that the event was logged. The information stored on the ABLOC system is used to increase the forensic reliability and accountability of the logs generated by the HIS system. ABLOC does not link patients between different HIS that reside in each network realm: a patient who sees a clinician at a clinic and then at a hospital does not have the visits linked in the ABLOC system. The ABLOC system was designed to protect the privacy of the patients who visit sites where ABLOC technology is used, and it avoids linking data about the patients that could be intrusive.

The log data generated by the HIS will be processed before it is added to the ABLOC system. This processing involves hashing the data using a SHA2 or, preferably, SHA3 hash function that has been approved by NIST [82]. This function will take the log as input and produce a fixed length output that is computationally difficult to reverse into the original data [82]. This results in the log data being obscured but still verifiable by recalculating the hash. The initial hash and log metadata remains within the realm's control and is not shared with the wider ABLOC network until a summary of it and other entries are created. The log hashes will be hashed again to create a Merkle tree for each patient; the Merkle root from the tree will then be placed in the Level2 block along with the patient identifier. Since the input data to the Merkle tree was created from a difficult-to-reverse hash, the hash function the Merkle tree uses does not have a negative impact on privacy, if it is less robust, since hashed data will be concatenated and hashed again. The Level2 blocks will not be published on the internet but will be communicated, in a secure manner, with other partner realms in the ABLOC system. Sites that participate in the ABLOC system will still store logs from the HIS in a database, along with other sensitive patient information, that the realm is required to protect. If the database table that maps a patient's name to the HIS' patient identifier was leaked, the approximate visit times of the patient to the healthcare site could be mapped to a specific person if the log entries were also leaked. The times that the patient visited the site also are stored in the main HIS database, which would have to be compromised to link the patient to approximate visit times available in ABLOC. The log on ABLOC would be linked

to the patient but due to the difficulty of reversing the hash, it is not likely that the original log could be found. Therefore, the confidentiality of the log information, encoded as hashes, stored at the other realms in the ABLOC remains intact.

5.2.2 Accountability Increase

The forensic reliability and accountability of the logs stored in the ABLOC system are increased by hashing the logged events and storing these hashes, or summaries of hashes, on a blockchain. The hash, stored on the blockchain, increases the forensic reliability of each log by allowing the current hash to be calculated. If the hashes are the same, it proves that the log entry has not been modified. The hash acts as a fingerprint for the log from when it was generated by the ABLOC system. Hashing the log data, when it is being used in an investigation, in the same way as the hash that was uploaded to the blockchain should result in the same hash value if the log has not been changed. If the original hash is not available, it would still be possible to verify the logs using a Level2 or higher level summary. The appropriate logs can be hashed to reconstruct a Merkle tree that is embedded in the summary blocks. This method could prove that multiple logs are unmodified at the same time. The hashes and summaries are stored on a DAG blockchain, and the underlying technology of the blockchain is relied upon to ensure that the information stored on the blockchain is not modified. Blockchain is known to be able to immutably store data [7, 12, 13, 14, 15, 16, 17, 18, 26], making it impossible to modify these hashes after they have been stored on the blockchain; it is only possible to read the hashes and append to the ledger.

5.2.2.1 Threats to Forensic Reliability

The following have been identified as threats to the forensic reliability provided by the proposed ABLOC system. These are threats to the ABLOC system not compromises to the HIS software itself.

- Man-in-the-middle attack: A malicious actor may intercept the hashes as they get sent to the blockchain. It is plausible that an attacker may attempt to intercept information that is being sent to the blockchain, modifying the data being stored on the blockchain to match changes they make to the HIS database. This risk could be mitigated by signing the information sent to the blockchain

using a private key, for authenticity, and by using an encrypted channel to communicate to the blockchain servers. This idea also applies to the gossip protocol to distribute summaries to the other ABLOC realms.

- Blockchain Sybil attack: A Sybil attack occurs when an entity has control over multiple, potentially fake, profiles on the blockchain. This can result in the malicious actor having more influence over the network or potentially recomputing the history on the chain [12, 27, 29]. This sort of attack could impact a local realm's blockchain, modifying what information it is able to store and send to the network or potentially approving a malicious smart contract. As this is a private blockchain, all new users on a local blockchain should be vetted before giving access and, therefore, should be semi-trusted. Furthermore, it is agreed that the majority of nodes in a network would be acting honestly and only a few would be compromised or malicious [29].

5.2.3 Network Sharding and Recovery

This subsection will discuss what is expected to happen when a single realm from the network goes offline, then rejoins, and if the network is divided into sub-networks. In the proposed ABLOC architecture, log data will be generated and saved to the local realm's blockchain, regardless of network connectivity status. This data is summarized for sharing with other blockchains within the ABLOC network; in the case of a network outage, this data will be shared when the connection is restored, using the normal protocol for inter-site communication.

Single Offline Node Suppose there is an ABLOC network with n nodes. If an internet networking failure occurs for a node, n_i, it is no longer able to connect to the ABLOC network. Node n_i will continue to generate logs and summaries (these will not be shared, as no node may connect to n_i to receive them). The remainder of the nodes in the network may still generate and share messages normally; no new information will be shared about the node n_i, as none has been received by the network. The network is still able to run with a node that has gone offline; if any node in the network tries to connect to n_i, it is not able to connect and chooses another to connect to.

When the network recovers and any node, n, connects to n_i, n is able to receive

any changes in n_i's frontier set. The node n will go on to distribute n_i's messages. Likewise, n_i is able to connect to other nodes and receive updates about the state of every other node in the network. Thus, any node that was offline should be able to rejoin the network, allowing the node to send and receive updates that were made while it was offline.

Network Partitioning Suppose there is an ABLOC network with n nodes and an internet or intranet networking failure occurs. This results in the network being separated into p partitions. Within each partition, the nodes are able to gossip normally. While the network outage persists, no messages from nodes outside the partition will be received; all nodes within a partition will eventually converge to a common frontier set for the nodes outside the partition. New messages will be sent and received normally by the nodes within the partition.

Once the network conditions allow the partition(s) to communicate once again, the network will recover. Any node(s) from partition p_1 could connect to a node in partition p_2 and exchange messages. This would allow the network to come back up to date; nodes communicating across the partitions and within the partitions spread the messages. It is important to remember that once a node has received a message from the originating node, it becomes a distributor of that message, allowing the recovery to happen relatively quickly for messages that have been sent and distributed to all nodes within a partition p_i.

If a partition contains only one node, the description in Section 5.2.3 applies. This reasoning can be applied to a network partition recursively until the base case of a single node, is encountered. Therefore, the inter-node gossiping employed by ABLOC satisfies the requirement of allowing a node(s) to recover from network partitioning and being offline.

5.2.3.1 Log Summarizing

To generate the summary of logs for a particular level, logs from a single entity (patient, realm, region, etc.), during the summary time frame are grouped into a Merkle tree. In the summary block, only the Merkle root is stored. This operation involves hashing each entry, as each leaf in the tree is a data point to be summarized. A Merkle tree is a full binary tree [28]; creating binary trees is not an overly intensive task and can be completed efficiently. When creating this structure from the bottom

up two values from the tree are concatenated and hashed, and the resulting hash becomes the root of the two values. Each subsequent level in the binary tree has half the number of nodes to hash resulting in an exponential decrease in the number of terms to process. As the hashes' input data is small amounts of text, the amount of time to generate the hash is fairly small. If there are concerns about the amount of time taken to create a Merkle tree, it would be possible to change the hashing algorithm that is applied to the log hashes: using SHA1 or MD5 hash algorithms would be faster. These hash algorithms have been known to be reversible, but the input data of the Merkle tree is hashed data so this is less of a concern. Additionally, it would be possible to shorten the summary interval, including fewer log events in the tree.

Merkle trees take up less space than other data structures [28], so storing only the Merkle root will speed up the confirmation time of the blocks on the chain, as there is less data to process.

5.2.3.2 Inter-Realm Gossip Protocol

The literature states that gossip protocols are scalable and send messages in an epidemic manner [65, 66, 67]; this is expected of the ABLOC protocol. Suppose there is a gossip network with n nodes. When a node, n_j, generates and sends a message, it will be gossiped to another node, n_n. Now both nodes n_j and n_n can send out the message. This results in four nodes that can gossip the message out to the rest of the network. This process continues until all nodes have the message.

Gossip Round	Informed Nodes
0	1
1	2
2	4
3	8
4	16

Table 5.1: Informed nodes per round of gossip

As more nodes join the network, more rounds of gossip result in more nodes receiving the message. Each node is then able to spread the message to other node(s). Many messages can be sent in a gossip exchange, and adding more nodes to the network will not greatly impact the scalability of this implementation. Table 5.1 shows that after n rounds of gossip, 2^n nodes have received the message, assuming no network outages. A proof of this relationship is presented in appendix A.4.

Scalability with multiple nodes offline When a node has been offline and has had its connection restored, it can reach out to the ABLOC network to receive summaries that it has missed. The offline node, n_o, will randomly choose another node, n_r, from the known ABLOC network. As long as n_r was not also offline, it is likely that n_r has the majority of the log summaries from the network that were created when n_o was offline. The node n_o can request the updates from n_r and n_o will be brought up to date by these messages. The node n_o will then connect to other peers in the ABLOC network and update further if required. Assuming that network outages are not widespread and the other nodes in the ABLOC network are sharing summaries as intended, it should not take many rounds of the gossip algorithm for n_o to be brought up to date with the network, as each node is able to share the messages it has received about its peers in the network.

As the number of nodes that go offline increases, it is more likely that a given node, n, that was offline will connect to another node that was offline n_o. Since n_o does not know about the updated state of the rest of the network it can only gossip what it knows. The node n_o may only be able to share summaries it has generated while offline resulting in n having to go through multiple rounds of gossip in order to synchronize with the network. As more nodes are able to gossip with each other after a large number of nodes are offline, each node becomes a distributor for the messages that it has received allowing the summaries to spread faster [65, 66, 67].

5.2.4 Blockchain Efficiency

Each realm in the ABLOC system is intended to serve a single medical software instance with a blockchain unless another piece of software is implemented within the same network locations (i.e. within the same data centre and hospital), allowing it to be part of the same network realm. A Hyperledger Fabric blockchain should be used for each realm. The choice of consensus mechanism will have a large impact on efficiency. As discussed in 3.1.3.1 proof of work is a computationally intensive mechanism and it is not recommended for this application as the costs to create a block would be too high. Instead, using some of the other algorithms discussed in 3.1.3 would lower the computational costs of creating blocks, making it more efficient in terms of time and energy to store blocks on the blockchain.

5.3 Proof of Concept Creation

To support the theoretical analysis of the proposed ABLOC system, a prototype was created. The prototype system implemented the major features of the proposed ABLOC system and included a single level of log summarizing. This section will discuss the assumptions used to create the proof of concept, the blockchain and smart contract solutions used, and how the inter-realm gossip protocol was created, and will outline some limitations and improvements.

5.3.1 Assumptions

The prototype was constructed under the assumption that a group of healthcare sites in the same geographical area were coming together to implement the ABLOC system. The system produces summaries at the Level2 level but could be changed to include more levels in the future. This prototype also assumes that the realms are backing up copies of their logs in a way that would allow them to be recovered if required.

5.3.2 Blockchain Choice

To develop the proof of concept, different open-source blockchain tools were considered. The Hyperledger Fabric, FireFly, and Sawtooth projects were considered, in addition to an Ethereum private network. The Hyperledger FireFly was chosen. Hyperledger FireFly is a blockchain supernode that contains the complete stack for building Web3 applications [39]. FireFly is a development tool that manages and runs the Hyperledger Fabric or a private Ethereum blockchain for the developer. The blockchain functions are available to the developer through an API; this relieves the developer of managing all of the lower-level blockchain and Docker containers that are required for a Hyperledger Fabric blockchain [83]. Additionally, the developer can choose the number of supernodes in the network. It is possible to send messages between these supernodes and it is also possible to connect to other existing blockchains.

The FireFly tool allows the user to choose to use either a Hyperledger Fabric or Ethereum blockchain, which can either be private or publicly accessible [84]. Hyperledger Fabric was the blockchain chosen for use and the supernodes were not connected to any external blockchain tools. This resulted in a private blockchain that simulated a realm for a general practice clinic. Multiple Hyperledger FireFly instances were run

on five different machines, as it was not possible to run multiple instances of FireFly on the same machine due to port collisions. Each instance was made up of at least two supernodes and ran the Hyperledger Fabric blockchain. The smart contract was deployed to the blockchain and FireFly was used to generate an API for its functions. The default IP address for accessing this API was localhost, but this was changed each time the blockchain was reset and the smart contract redeployed.

Using Hyperledger FireFly made it easier to manage the blockchain during development and scalability testing. With a single script, taking advantage of the FireFly commands, it was possible to reset the blockchain, deploy a smart contract, and create an API for the smart contract without using the FireFly GUI.

5.3.2.1 Smart Contracts

A smart contract was written in Go, using the Fabric Contract API [85], and then deployed to the FireFly Blockchain. Hyperledger Fabric refers to a smart contract as a chaincode [86]; both terms will be used interchangeably. The smart contract could have also been written in Java, or Node.js, the languages supported by Hyperledger Fabric [85]. Go was chosen as there was the documentation for the deployment of Go based smart contracts on Hyperledger FireFly blockchains. The Hyperledger Fabric smart contract API library and the Hyperledger chaincode library are used to interact with the blockchain, along with libraries to support SHA256 Hashing and the creation of Merkle trees.

The smart contract allows a realm to upload Level1 log entries to the blockchain and it will automatically, when required, generate Level2 blocks. It is also possible to retrieve a single hash with the smart contract, along with getting entire Level1 and Level2 blocks. The smart contract also allows new nodes to be added to the realm's frontier set and responds to gossip queries requesting either the frontier set or blocks needed to update the gossiping peer's realm.

In addition to the features required for the ABLOC system, some development functions were implemented. These functions allow the retrieval of specific Level1 blocks, all the Level2 blocks, all the blocks from specific realms, and the last Level2 block that was created by the realm. These features facilitated the debugging of the smart contract and likely would not need to be included in an actual implementation of ABLOC.

The smart contract was packaged and deployed to the FireFly blockchain following the steps outlined in the project wiki. FireFly requires an interface, called the FireFly Interface (FFI), which governs the interactions between the smart contract and the API generated by FireFly. The FFI must be handcrafted using structured JSON. The developer must encode and decode the values sent through the interface, as opposed to defining this in the interface. This interface is broadcast to the FireFly managed blockchain after the smart contract is packaged and deployed on the chain; this creates an HTTP API for the contract [86]. A copy of the smart contract is available in Appendix A.7, FireFly Interface is available in Appendix A.8, and a screenshot of the Swagger UI for the prototype smart contract is available in Appendix A.9.

5.3.3 Gossip Protocol

The gossip protocol was implemented twice for scalability testing: it was implemented independently to design and test the viability of the gossip protocol, itself, and then it was implemented as part of the ABLOC prototype.

5.3.3.1 Independent Gossip Protocol Implementation

The gossip protocol was first implemented to investigate if a gossip protocol would meet the requirements of ABLOC. The investigation's goal was to confirm that a peer in the gossip network could go offline and then recover, resynchronizing with the network. Initially, the protocol was implemented in Python3; each peer in the gossip network ran a single Python process that allowed the node to listen and respond to requests for gossip, and to request gossip from other nodes. This implementation required managing sockets, encoding and decoding the data sent between nodes, and managing timing out was created, which resulted in additional complications. The initial implementation relied on a class to represent peers in the network, a class to represent messages sent by each peer to the network, and a storage class to represent a blockchain or other data storage strategy. These classes can be seen in Figure 5.1, a class diagram. The main file, a multi-threaded program, used these classes and tracked each node in the network, managed the frontier set for the node, responded to gossip, and requested gossip from other nodes when appropriate. Additionally, this file generated new messages to send to the network.

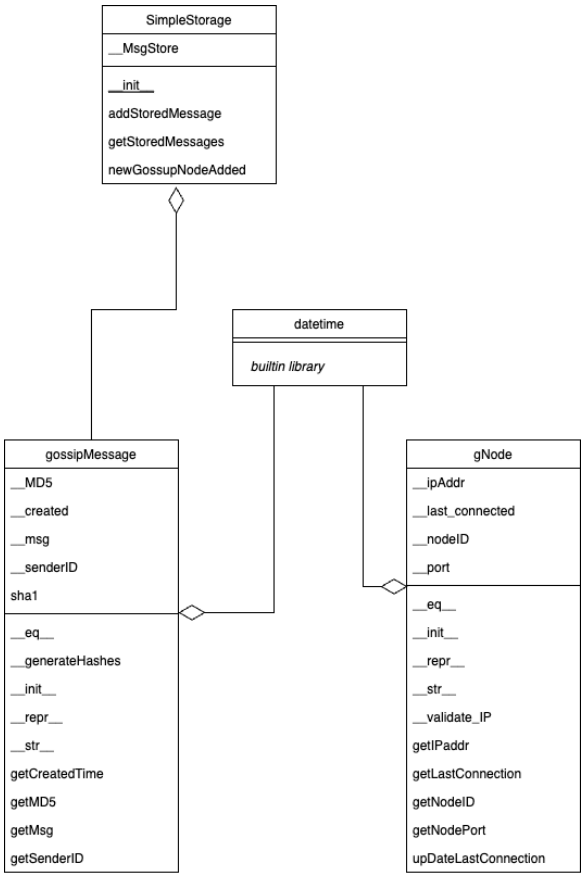

Figure 5.1: This class diagram represents the classes created for the initial Gossip protocol implementation.

The main gossip file created a gNode for each node in the test network and created a SimpleStorage object to manage the messages that it would generate. The main gossip file was multi-threaded: one thread was used to monitor the other threads, one was used to respond to gossip requests, another was used to generate new gossip messages, and the last was used to get gossip from other nodes in the network. To respond to a request for gossip, a node sends its frontier set to the node requesting gossip, allowing that node to compare the received frontier set to its own and request any messages where the responding node was ahead. The responding node shares the messages and the receiving node saves them to its SimpleStorage object. The receiving node also prints the message to the screen, indicating that a message had been received. The MD5 hash of the message and the time that the message was received are also printed. In the main gossip file, a function generates random messages to be stored as message objects. This created larger blocks of apparently gibberish data, representing

encrypted data, to be managed by the protocol. While the protocol runs, it prints out the messages that it sends and receives. When a shutdown signal is sent, the protocol prints out the frontier set and then exits.

This implementation was used to simulate the network size increasing and investigate the scalability of the gossip protocol: how does the number of nodes impact the amount of time it takes to share a message with the entire network? This setup was also used to verify the behaviour of nodes that went offline to ensure that the protocol could support the ABLOC system.

The implementation of the gossip protocol showed that the protocol allows different network nodes to communicate effectively. If internet routing problems occurred and the network was divided into subgroups, the gossip implementation was still able to run within the subgroups and resynchronize the network when the connections had been restored. Furthermore, if a single node went offline once the node recovered it was able to resynchronize with the rest of the network.

This implementation was also used for scalability analysis; creating a large number of blockchains to test the scalability of the gossip protocol was not practical, so the SimpleStorage representation was used. For each test, each node generated ten messages, waiting 60 seconds between each message. These messages simulated the ABLOC realm generating summaries and sharing them with other realms.

Mininet Offline Testing - Single Offline Node and Network Sharding For these tests, five Mininet hosts (each simulating a networked machine) and two network switches were created. Three hosts are connected to one switch and two are connected to the other. The two switches were also connected. The topology code is available in the Appendix A.5. The network was started and the gossip protocol was run for five minutes. Then the link between a host, $host_O$, and its switch was taken offline. After five minutes the link was brought back online and the network was allowed to recover. After recovering, all the nodes received all the messages. This test was run multiple times successfully.

Next, the same network topology was used to simulate a network being divided into two shards. For these tests, the link between the two switches was removed and then reinstated after five minutes. When the protocol was halted after the recovery time, all the nodes received all generated messages. This test was run multiple times and

was successful on all attempts.

Initial Gossip Protocol Scalability Testing A simple network topology was created for these tests in order to compare different network sizes. ABLOC realms will communicate over the internet using public IP addresses but this was not simulated in the scalability testing. The topology between the nodes is simplified for these tests to allow them to be run in Mininet.

The scalability of the initial ABLOC gossip protocol was tested by increasing the network size answering the question of how an increasing network size impacts message delivery times. Networks of 5, 10, 25, 50, and 100 hosts were created in Mininet. The minimum number of switches for each network was used to connect the Mininet hosts. For the largest network sizes the experiment required multiple switches, networks with more than 100 hosts are incompatible with a single Mininet switch. Instead, a single connection was made between two switches that each connected to 50 hosts. This limitation was not discovered until all but the largest tests were completed. It was decided that retaining the data outweighed the disadvantages caused by the changes in network topology.

The Python script discussed is available in Appendix A.6 was used to create the network, send the network information to each host, and start and stop the gossip protocol. Each host was configured to pipe output, a timestamp every time a message was sent or received, to a file. Each network size was run three times; no network interruptions were induced. A processing script consolidated the data for analysis.

There is little increase in the amount of time it takes, on average, for a message to be distributed throughout the network as the network size increases.

Initial Gossip Protocol Discussion The offline node and network sharding demonstrate that this algorithm can successfully recover after a network outage. All of the test messages, in each test, were synchronized to each node in the network. This is not an instantaneous process, due to the amount of time nodes wait before choosing a new node to gossip with. Messages that the network strongly agrees on (older messages) synchronize to the offline node faster; any node that has received messages is able to share it with the shard that is recovering. A newer message that every node in the main shard has not received takes longer to be merged into the offline shard because there is a chance that the first node contacted for updated gossip may

not know about the new message. This is a factor in how the messages spread. As more nodes join the network and receive messages, more nodes are able to pass on the messages to the other uninformed nodes. The relationship between the size of the network and the time it takes for a message to be distributed, without network interruptions, appears to be logarithmic, as shown in Figure 5.2. This corroborates with the analysis suggesting that the number of nodes distributing the message, as the network size increases, grows exponentially.

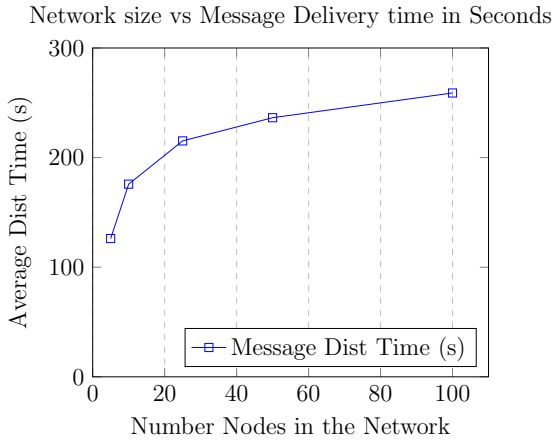

Figure 5.2: Initial Gossip Protocol: Message Delivery vs Network Size testing resulted in message delivery time logarithmic growing as the network size increases

This gossip protocol is not the fastest way to distribute a message to every node in the network, in the case of log data speed is not the most important factor. The protocol is able to recover from network outages and has been shown to scale with network size.

5.3.3.2 Final ABLOC Inter-Realm Gossip Protocol Prototype

It was anticipated that the original gossip protocol would be incorporated into the ABLOC system. However, when developing the smart contract and other aspects of the ABLOC system, another option was considered. Instead of having a server respond to gossip from peers, the smart contract was used to respond to gossip requests. This split the actions of the gossip protocol (getting gossip and responding to requests for gossip) but allowed the receiving realm to be more confident that the gossip, from a standardized smart contract, would be genuine. This required REST API calls to be created for getting the local node's frontier set, which is stored on

the blockchain, and for requesting gossip from the node. A gossip Python program, run off of the blockchain, was created to request gossip from other nodes in the network. Moving to this model required a rewrite of the portion of the gossip protocol that would retrieve gossip. This resulted in the use of REST API calls and simpler code. The overall strategy of the gossip protocol remains the same. It is noted that the FireFly blockchain tool can easily be overloaded with too many requests to store information on the blockchain. When the gossip protocol uploads new blocks to the blockchain no-ops are required between each block that is uploaded prevent this. In the prototype, this increased the amount of time required for synchronizing the blockchains, and it is recommended that this is removed in future implementations.

5.3.4 Log Generator

A log simulator was created to generate inputs to test the ABLOC system. This simulator takes log entries from multiple test instances of OSCAR EMR and processes them to have the current date and time, simulating a HIS currently running. This generator uses the original log's generation time to attempt to generate logs in the same groupings as they were generated in the EMR (i.e. if a set of five logs were generated together by the EMR, they are generated together by the generator). The generator generates new patient identifiers or chooses to reuse an existing patient identifier (if the generator has run long enough). Additionally, it can simulate multiple clinicians, the number of which is chosen by the user, using the HIS. The resulting log entries are processed into a Level1 block that is then uploaded to the blockchain. This allows the ABLOC system to be run and tested for longer, potentially infinite, periods of time.

5.4 Limitations of the Proof of Concept

The ABLOC prototype had a limited scope due to the implementation choices that were made when creating it. The scope of the prototype was limited to realms with only one piece of medical software deployed on it and the realm was run on a single computer. Each realm represented a general practice clinic that generally just runs an HIS, usually an EMR system, as opposed to a hospital, where there is the potential for multiple different HIS to support different areas of practice (pharmacy, radiology, laboratory, charting, etc.). Furthermore, this prototype used a log generator,

essentially a fuzzer, to test ABLOC instead of integrating it with an HIS.

In addition to the scope limitations, there were technical limitations. The FireFly supernode uses Docker to run the blockchain; it was not possible to run multiple clinics on a single machine due to port collisions. Some of the machines that were used to simulate a realm were limited to 16 GB of RAM and older dual-core processors. This limited the speed at which the blockchain could process new transactions being saved to the chain. Even on a newer machine, with a six-core processor and approximately 100 GB of RAM, it was still possible to overwhelm a FireFly supernode which could result in the loss of data. To minimize this, each tool that uploaded new data to the blockchain slept for a couple of seconds during each transaction to prevent the supernode from being overwhelmed. This resolved the issue but increased the amount of time required to process information. New logs being uploaded to the blockchain, from the generator, were prioritized with the shortest sleep time, and uploading gossip from other nodes was given the longest sleep time. The sleep limits were the same for all test instances of the machines used for ABLOC. These limitations of the ABLOC system were taken into account when the system was evaluated.

5.5 Evaluation of the Final Proof of Concept

The ABLOC proof of concept was evaluated by setting up five machines on a local network, each running the same ABLOC prototype. The network diagram in Figure 5.3 shows the network setup and computer specifications. This setup was used to test sharding the network, run multi-day testing, and run scalability tests. Each machine, representing a realm in the ABLOC network, was manually assigned a static IP address; this address and other required information was manually set in each machine.

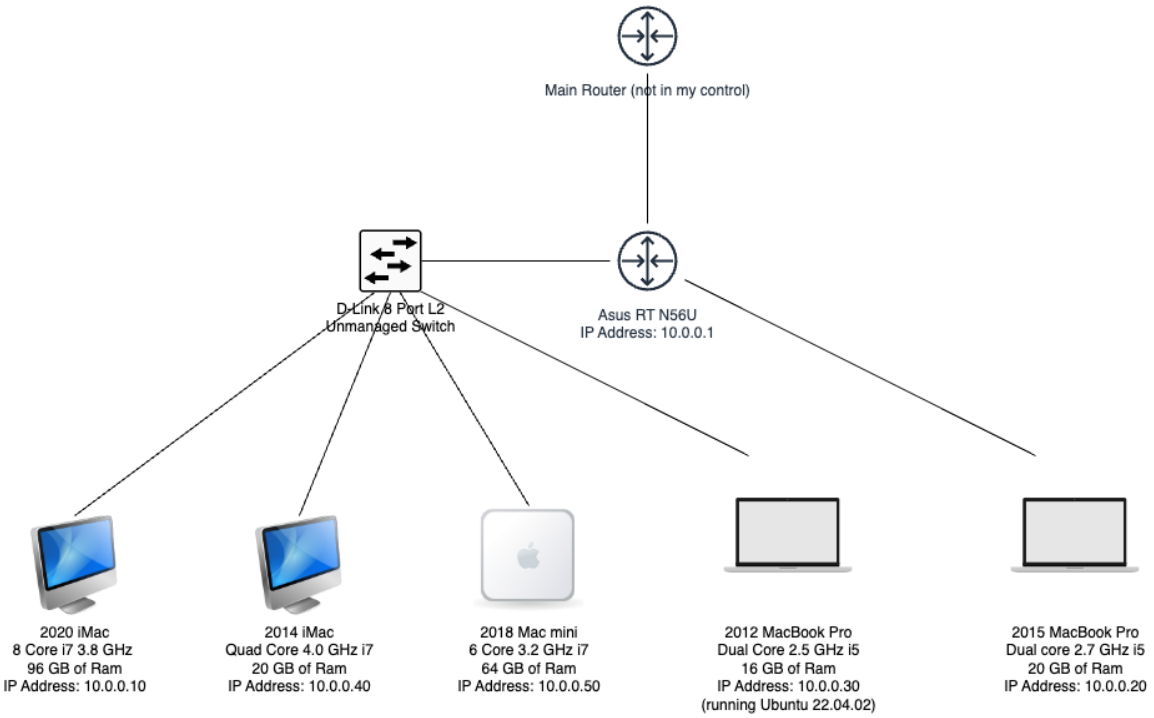

Figure 5.3: Network Layout for Prototype Evaluation five computers simulated realms in the ABLOC network. The 2012 MacBook Pro was removed from the network by unplugging its network cable.

This network was used to evaluate the proof of concept. Before each test or iteration of a test, the ABLOC system was reset. This cleared the blockchain, removing previous blocks. This also removed the smart contract, so it had to be redeployed and the REST API had to be recreated. Scripts were made to automate this task.

5.5.1 Network Loss

In addition to the sharding tests done while developing the gossip protocol, the entire ABLOC prototype was tested for a node going offline and then recovering. This was also tested multiple times in the scalability testing: nodes in the ABLOC network were taken offline and were able to resynchronize with the network when the connection was restored. Including the scalability tests, where nodes were also brought offline, network recovery was tested approximately 50 times. Each iteration involved a node being brought offline by removing the cable connecting it to the router or switch. After restoring the connection, the nodes were eventually able to resynchronize with

the network. All of the summaries were taken from each realm and compared to ensure that realms were synchronized with one another.

5.5.2 Multi-Day Testing

Using the network setup described in Figure 5.3 nodes were set up to run and generate logs. This setup was left to run for over 24 hours, resulting in over 240 summaries shared between the nodes, and the blockchains remaining synchronized. This test was run to ensure that the gossip protocol and blockchain would be able to interact, without errors, over a period of time longer than tested during development. No error messages or crashes occurred during this testing. This test was performed prior to the scalability tests to ensure that the gossip protocol was stable; bugs were found that made the gossip protocol less efficient, but still reliable. These were corrected prior to the main scalability test and made the processing of the gossiped logs more efficient.

5.5.3 Offline Scalability Tests

The scalability testing for the prototype evaluation investigates how the prototype is impacted as a realm spends more time offline; how does this impact the resynchronization time for the entire network? It is already known, from the experiments validating the gossip protocol in Section 5.3.3.1, that as the network size grows the time it will take to distribute summaries through the network grows in a $log(n)$ fashion. This section outlines an experiment used to gauge how the amount of time a node is offline impacts the network resynchronization time. The experiment will be outlined, and the results presented and discussed.

The experiment used the same setup as described in Figure 5.3. A log generator was created that generated Level2 log entries, which are all the same size. When it was started, this generator asked how many hours were being simulated and then it would generate the appropriate number of Level2 summaries for the realm. Summaries were generated on each realm, including the offline one. The gossip protocol was run on the online realms to share the summaries between realms. A script was used to access the online realms to get all of the Level2 summaries stored in each realms' blockchain. The script would then compare each set of summaries to determine if the realms were synchronized. Once the online realms had synchronized their summaries, the offline

realm was returned to the network. The offline realm was then resynchronized with the rest of the network. Every time a block was retrieved from another realm it was logged, along with the start and end time of each gossip exchange with a peer in the network. From this information, the time it took for each realm to resynchronize was measured and recorded.

The experiment was run five times at each amount of time that a realm was simulated to be offline, for a total of 30 iterations of the experiment. The gossip retrieval logs were cleared for each run in addition to resetting the blockchain. The steps to conduct an iteration of the experiment were as follows:

1. Ensure that each realm's blockchain was reset and the smart contract and REST API were redeployed.

2. Unplug the network cable from the machine simulating the offline realm. This would remove all network connections from the realm. The MacBook Pro 2012 running Linux always represented the offline realm.

3. Start the Level2 generator to generate Level2 summaries, simulating the network running for h hours with a Level2 summary being created every 30 minutes, on all machines (including the offline machine). Depending on the amount of time offline being simulated, it may take some time to generate the Level2 summaries. The generator will sleep after each uploaded block to allow the blockchain some time to process the uploaded summary.

4. While the Level2 summaries are being generated, start the gossip retrieval code on each node, except for the offline realm. This is the code that gets gossip from the other network members it can reach allowing the network to synchronize the Level2 summaries that are being created.

5. Wait until the nodes have generated all the messages, simulating the passing of h hours, and all the online nodes have synchronized their summaries.

6. Reconnect the offline realm to the network and start the gossip retrieval thread on that realm.

7. Wait until the network has been synchronized.

8. Save the output of each realm's gossip retrieval code and the Level2 blocks

stored by each blockchain.

9. Reset each realm's blockchain preparing to restart the experiment. Repeat.

This procedure was run 30 times. The amount of time that the realm was simulated as being offline started at two hours, then four, eight, 16, 32, and finally 64 hours; for each increase in the amount of time a realm was offline the experiment was run five times, for a total of thirty iterations. The number of iterations was limited by the amount of time it took to reset the blockchain after each experiment and the amount of time it took to run the experiments where the realm was offline for more than a day. This relationship is expected to be linear with a single offline node, as the summaries only need to be processed once by the receiving realm.

The data logged by the gossip retrieval was analyzed and the synchronization times were calculated for each iteration. For each network outage time simulation, the synchronization times were averaged. The data are presented in appendix A.10 and the summary data is presented in Figure 5.4. The node that took the longest to resynchronize with the network was the node that was taken offline and then allowed to recover.

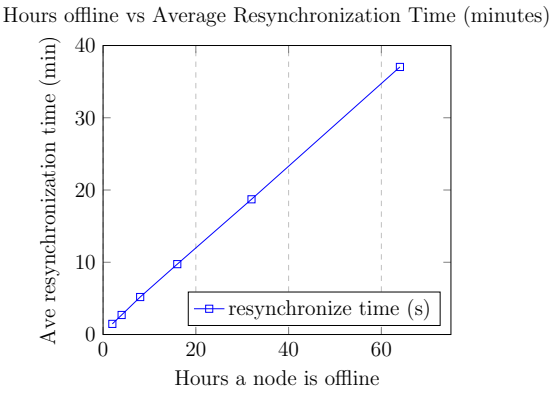

Figure 5.4: Time offline vs. Resynchronization Time. This shows the amount of time a realm was simulated being offline compared to the amount of time it took to resynchronize when brought back online.

As the amount of time a realm is offline increased, resulting in more summaries not being sent or received, there was an increase in the amount of time taken to synchronize the network. The relationship appears to be linear, as shown in Figure 5.4. This is intuitive as the amount of time a realm is offline there is a linear increase in the number of summaries it needs to receive. The slowest realm to recover was always

the one that had been taken offline. This is expected as it had the most number of summaries to process because the other online realms also generated messages during the simulated outage. The offline realm had to process four times the number of messages as the other realms, which it did in three to four times the amount of time compared to the other realms. The amount of time that it takes to process a message could be reduced using a different blockchain implementation, as more resources could be dedicated to the blockchain. This would likely reduce the likelihood of a realm being overwhelmed by the number of summaries to process and save to the blockchain, reducing the need for sleep time in the code processing gossip. This sleep time was applied across the network evenly so it did not impact the analysis; all realms were equally hampered by it. There were no patterns found indicating what machine was the fastest at synchronizing with the network. This also is expected as random choices are made by the gossip protocol as to which realm to gossip with. In the end, all the nodes were able to synchronize with the network sharing all the generated summaries successfully.

5.6 Suggested Improvements

After creating an ABLOC prototype, using Hyperledger FireFly and a single smart contract, the following improvements are suggested for future implementations of the ABLOC log system.

5.6.1 Offline Nodes

If a realm cannot be gossiped with for a longer period of time (i.e. a month), the realm should be marked as being offline in the frontier set of the realms trying to gossip with it. This could occur if the realm was being retired or going offline long-term for other reasons. If this realm was chosen to be gossiped with, it would be skipped. If gossip was requested by this realm (indicating the realm was back online) then it would be reactivated in the frontier set of the local realm, allowing it to be gossiped with once more.

5.6.2 Use of Encryption

Public/Private key encryption should be used to sign messages to show they come from genuine sources. This should be applied to the summaries and Level1 entries

from the HIS. Additionally, this should be used to encrypt the communications that are involved with the gossip protocol. It should also be used to authenticate gossip requests to ensure that they originate from authenticated users of the ABLOC system, and to prevent the messages from being understood if they are intercepted.

5.6.3 Use of Hyperledger Fabric

Using a more robust blockchain system, as opposed to Hyperledger FireFly, would allow for more control over the blockchain components during implementation. This would permit the use of channels on the local blockchain [69] which would allow for the organization of the different levels of summaries on the chain, in addition to supporting multiple pieces of software within the same realm. A channel is considered to be a sub-blockchain in Hyperledger Fabric. According to the documentation, these channels can have restricted permissions to only allow certain people to use the channel [69].

5.7 Comparison to Other Solutions

The proposed ABLOC system is a scalable way to ensure that log files from multiple pieces of software, remain secure and can be used to hold users accountable for their actions. Other proof of existence solutions that manage files or entries individually would not scale well to allow for sharing over a wider network [54]. Some of the proposed solutions to address this with blockchain solve this problem, but are vulnerable to network outages [7, 29, 54]. During a network outage, healthcare providers would still be expected to provide services and should have access to their record-keeping software to aid in providing care. This access needs to be auditable to ensure that access to patient information is appropriate. Additionally, the ABLOC system allows for multiple organizations/realms to help protect each other's log files, mutually increasing the security of the log files when compared to storing them within the organization. This property is discussed in EngraveChain [7]. ABLOC improves on the proposed EngraveChain system by allowing realms to go offline while maintaining the logging accountability capabilities. Additionally, ABLOC the entries to be summarized before they are shared with the rest of the network. These summaries can be used to authenticate the logs and are more efficient to store.

Chapter 6

Conclusions and Future Work

6.1 Summary

ABLOC, Accountable Blockchain Logging for Offline Care, is a system that can prove that the log entries generated by medical software have, or have not, been tampered with. The proposed ABLOC system meets the goal of a tamper-resistant blockchain, for log storage, that continues to function with the loss of network connections. ABLOC is made up of multiple network realms containing at least one piece of medical software that generates logs. A realm may also function as a repository for summaries of the logs. A clinic or hospital may have multiple realms to manage the logs from the medical software in use.

Each realm has a blockchain with a smart contract that allows the HIS software to upload log entries to the blockchain. The smart contract summarizes the logs, using Merkle roots, within a set time frame. This summary of the log entries is shared with other realms in the ABLOC network using a gossip protocol. The shared summaries reference the original blockchain entries with the log's information, creating a DAG that is made up of many independent, parallel blockchains. These summaries may be summarized again and shared on a wider network. The process of creating a summary shall follow the same procedures as lower levels. Summaries can be created for multiple levels, each using the lower level to generate the next summary, to create as many granularity levels as desired by the network.

In the case of a network outage, the blockchain should continue functioning, accepting

log entries and creating summaries, within the realm or sub-realm that is impacted. Once the network connections have been restored, these summaries are shared with the rest of the ABLOC network using the gossip protocol. The gossip protocol also contacts other realms and pulls summaries from those realms to resynchronize the rest of the ABLOC network. The specific gossip implementation, combined with the blockchains, was shown to be a scalable way to communicate messages between realms. Scalability testing was done to determine how long realms were offline impacts the resynchronization time with the network. Scalability tests were also done to see how long it takes a message to be delivered using the gossip protocol, with no outages, as the network size grows. These test showed that the gossip protocol would be a scalable way to connect the blockchain realms and facilitates a DAG-based blockchain to witness log summaries generated by medical software.

The proposed ABLOC system maintains the confidentiality of the log data through hashes. The blockchain infrastructure and summaries, forming a DAG, maintain the integrity of the data stored on the blockchain. This allows the log files' integrity to be verified. Finally, the ABLOC system is available to operate in the case of network outages. Therefore, the proposed ABLOC system satisfies the Confidentiality, Integrity, and Available triad of cyber security, allowing it to improve the integrity of log files to hold users accountable for their actions.

6.2 Contributions

The main contributions of this thesis are as follows

- Confirmed the work done by King and Williams [6, 49] showing that not all the actions in EMRs are logged.
- Proposed modifications to existing blockchain-based logging mechanisms that increase the forensic accountability of logging. This blockchain-based logging system was applied to the healthcare context. The proposed mechanism tolerates realms going offline but still providing care, with the clinicians' actions within the patient's record still being logged in a way that could be used to hold malicious users accountable.
- A proposed mechanism to generate summaries of the information used to prove that log files have not been modified. These summaries are shared and stored

by other peers in the ABLOC network.

- A prototype implementation of the ABLOC system, which was shown to be scalable
- A set of requirements was to aid in future implementations and improvements to the ABLOC system.

6.3 Future Work

The main features of ABLOC have been designed and a prototype has been implemented. The current implementation is a proof of concept and does not implement API security or other security features. A thorough security analysis of the components of the proposed system would be needed before deploying it in a production environment. In addition to the security hardening, it would be necessary to integrate the ABLOC system with existing EMR and other HIS infrastructure, and to build a production version of ABLOC using more robust and efficient blockchain infrastructure. Directly implementing ABLOC using Hyperledger Fabric would allow for finetuning of the resources reducing the time it takes to add new log entries to the blockchain.

In addition to hardening the security of the prototype, additional logic must be implemented and tested to allow sponsorship of new nodes and retirement of old nodes in the ABLOC network. Work also needs to be done to ensure compatibility with cloud-based infrastructure. Further improvements to the smart contract may also be explored; it may be desirable to modify the smart contract to become totally responsible for retrieving gossip from other realms in the network. There also may be value in allowing peers, from other realms, to validate the smart contract that they are using to access information from another realm's blockchain. Finally, it would be valuable to expand the application of the ABLOC system from the medical field into other fields.

Bibliography

[1] S. Amir-Mohammadian, S. Chong, and C. Skalka, "Correct Audit Logging: Theory and Practice," in *Principles of Security and Trust*, ser. Lecture Notes in Computer Science, F. Piessens and L. Viganò, Eds. Berlin, Heidelberg: Springer, 2016, pp. 139–162.

[2] H. Zhang, S. Mehotra, D. Liebovitz, C. A. Gunter, and B. Malin, "Mining Deviations from Patient Care Pathways via Electronic Medical Record System Audits," *ACM Transactions on Management Information Systems*, vol. 4, no. 4, pp. 17:1–17:20, Dec. 2013. [Online]. Available: https://doi.org/10.1145/2544102

[3] A. Boddy, W. Hurst, M. Mackay, and A. El Rhalibi, "A Hybrid Density-Based Outlier Detection Model for Privacy in Electronic Patient Record system," in *2019 5th International Conference on Information Management (ICIM)*, Mar. 2019, pp. 92–96.

[4] M. Jayabalan and T. O'Daniel, "Access control and privilege management in electronic health record: a systematic literature review," *Journal of Medical Systems*, vol. 40, no. 12, p. 261, Oct. 2016. [Online]. Available: https://doi.org/10.1007/s10916-016-0589-z

[5] M. Ahmed and M. Ahamad, "Combating Abuse of Health Data in the Age of eHealth Exchange," in *2014 IEEE International Conference on Healthcare Informatics*, Sep. 2014, pp. 109–118.

[6] J. T. King, B. Smith, and L. Williams, "Modifying without a trace: general audit guidelines are inadequate for open-source electronic health record audit mechanisms," in *Proceedings of the 2nd ACM SIGHIT International Health Informatics Symposium*, ser. IHI '12. New York, NY, USA: Association

for Computing Machinery, Jan. 2012, pp. 305–314. [Online]. Available: https://doi.org/10.1145/2110363.2110399

[7] L. Shekhtman and E. Waisbard, "EngraveChain: A Blockchain-Based Tamper-Proof Distributed Log System," *Future Internet*, vol. 13, no. 6, p. 143, Jun. 2021, number: 6 Publisher: Multidisciplinary Digital Publishing Institute. [Online]. Available: https://www.mdpi.com/1999-5903/13/6/143

[8] M. Sato and T. Yamauchi, "VMM-Based Log-Tampering and Loss Detection Scheme," *National Dong Hwa University*, 2012.

[9] J. H. Weber-Jahnke and C. Obry, "Protecting privacy during peer-to-peer exchange of medical documents," *Information Systems Frontiers*, vol. 14, no. 1, pp. 87–104, Mar. 2012. [Online]. Available: https://doi.org/10.1007/s10796-011-9304-2

[10] Verizon, "Protected Health Information Data Breach Report," Tech. Rep. [Online]. Available: https://www.verizon.com/business/resources/reports/protected_health_information_data_brea

[11] T. Zhou, X. Li, and H. Zhao, "DLattice: A Permission-Less Blockchain Based on DPoS-BA-DAG Consensus for Data Tokenization," *IEEE Access*, vol. 7, pp. 39 273–39 287, 2019, conference Name: IEEE Access.

[12] G. R. Carrara, L. M. Burle, D. S. V. Medeiros, C. V. N. de Albuquerque, and D. M. F. Mattos, "Consistency, availability, and partition tolerance in blockchain: a survey on the consensus mechanism over peer-to-peer networking," *Annals of Telecommunications*, vol. 75, no. 3, pp. 163–174, Apr. 2020. [Online]. Available: https://doi.org/10.1007/s12243-020-00751-w

[13] S. Namasudra, G. C. Deka, P. Johri, M. Hosseinpour, and A. H. Gandomi, "The Revolution of Blockchain: State-of-the-Art and Research Challenges," *Archives of Computational Methods in Engineering*, vol. 28, no. 3, pp. 1497–1515, May 2021. [Online]. Available: https://doi.org/10.1007/s11831-020-09426-0

[14] Y. Liu, J. Liu, M. A. Vaz Salles, Z. Zhang, T. Li, B. Hu, F. Henglein, and R. Lu, "Building blocks of sharding blockchain systems: Concepts, approaches, and open problems," *Computer Sci-*

ence Review, vol. 46, p. 100513, Nov. 2022. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1574013722000478

[15] H. Pervez, M. Muneeb, M. U. Irfan, and I. U. Haq, "A Comparative Analysis of DAG-Based Blockchain Architectures," in *2018 12th International Conference on Open Source Systems and Technologies (ICOSST)*, Dec. 2018, pp. 27–34.

[16] S. Khezr, M. Moniruzzaman, A. Yassine, and R. Benlamri, "Blockchain Technology in Healthcare: A Comprehensive Review and Directions for Future Research," *Applied Sciences*, vol. 9, no. 9, p. 1736, Jan. 2019, number: 9 Publisher: Multidisciplinary Digital Publishing Institute. [Online]. Available: https://www.mdpi.com/2076-3417/9/9/1736

[17] M. Kuperberg, "Towards an Analysis of Network Partitioning Prevention for Distributed Ledgers and Blockchains," in *2020 IEEE International Conference on Decentralized Applications and Infrastructures (DAPPS)*, Aug. 2020, pp. 94–99.

[18] P. Moghaddam, S. Iqbal, and I. Traore, "A Framework for Secure Logging in Precision Healthcare Cloud-based Services," in *2021 IEEE International Conference on Digital Health (ICDH)*, Sep. 2021, pp. 212–214.

[19] L. Quan, J. Heidemann, and Y. Pradkin, "Trinocular: understanding internet reliability through adaptive probing," *ACM SIGCOMM Computer Communication Review*, vol. 43, no. 4, pp. 255–266, Aug. 2013. [Online]. Available: https://dl.acm.org/doi/10.1145/2534169.2486017

[20] B. Ujcich, K.-C. Wang, B. Parker, and D. Schmiedt, "Thoughts on the Internet architecture from a modern enterprise network outage," in *2012 IEEE Network Operations and Management Symposium*, Apr. 2012, pp. 494–497, iSSN: 2374-9709.

[21] P. Evans, "ANALYSIS | What happened at Rogers? Day-long outage is over, but questions remain | CBC News," Apr. 2021. [Online]. Available: https://www.cbc.ca/news/business/rogers-outage-analysis-1.5994851

[22] K. Karlsson, D. Adams, G. Rubambiza, Z. Xian, R. Van Renesse, H. Weatherspoon, and S. Wicker, "Untethered: Deployable Blockchains for IoT Environments," in *Proceedings of the ACM Symposium on Cloud Computing.*

Carlsbad CA USA: ACM, Oct. 2018, pp. 508–508. [Online]. Available: https://dl.acm.org/doi/10.1145/3267809.3275450

[23] K. Karlsson, W. Jiang, S. Wicker, D. Adams, E. Ma, R. van Renesse, and H. Weatherspoon, "Vegvisir: A Partition-Tolerant Blockchain for the Internet-of-Things," in *2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS)*, Jul. 2018, pp. 1150–1158, iSSN: 2575-8411.

[24] M. J. Amiri, D. Agrawal, and A. El Abbadi, "On Sharding Permissioned Blockchains," in *2019 IEEE International Conference on Blockchain (Blockchain)*, Jul. 2019, pp. 282–285.

[25] A. T. Gorczyca and A. M. Decker, "Low-latency partition tolerant distributed ledger," in *Disruptive Technologies in Information Sciences*, vol. 10652. SPIE, May 2018, pp. 211–219. [Online]. Available: https://www.spiedigitallibrary.org/conference-proceedings-of-spie/10652/106520V/Low-latency-partition-tolerant-distributed-ledger/10.1117/12.2305022.full

[26] Y. Xie, J. Zhang, H. Wang, P. Liu, S. Liu, T. Huo, Y.-Y. Duan, Z. Dong, L. Lu, and Z. Ye, "Applications of Blockchain in the Medical Field: Narrative Review," *Journal of Medical Internet Research*, vol. 23, no. 10, Oct. 2021, publisher: JMIR Publications Inc. [Online]. Available: https://www.ncbi.nlm.nih.gov/pmc/articles/PMC8555946/

[27] Q. Wang, J. Yu, S. Chen, and Y. Xiang, "SoK: Diving into DAG-based Blockchain Systems," Oct. 2022, arXiv:2012.06128 [cs]. [Online]. Available: http://arxiv.org/abs/2012.06128

[28] S. Jing, X. Zheng, and Z. Chen, "Review and Investigation of Merkle Tree's Technical Principles and Related Application Fields," in *2021 International Conference on Artificial Intelligence, Big Data and Algorithms (CAIBDA)*, May 2021, pp. 86–90.

[29] J. Chen, X. Chen, K. He, R. Du, W. Chen, and Y. Xiang, "DELIA: Distributed Efficient Log Integrity Audit Based on Hierarchal Multi-Party State Channel," *IEEE Transactions on Dependable and Secure Computing*, vol. 19, no. 5, pp.

3286–3300, Sep. 2022, conference Name: IEEE Transactions on Dependable and Secure Computing.

[30] K. Fan, S. Wang, Y. Ren, H. Li, and Y. Yang, "MedBlock: Efficient and Secure Medical Data Sharing Via Blockchain," *Journal of Medical Systems*, vol. 42, no. 8, p. 136, Jun. 2018. [Online]. Available: https://doi.org/10.1007/s10916-018-0993-7

[31] Z. Xu, L. Yang, Z. Wang, S. Wen, R. Hanson, S. Chen, and Y. Xiang, "BHDA - A Blockchain-Based Hierarchical Data Access Model for Financial Services," in *2020 IEEE 19th International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom)*, Dec. 2020, pp. 530–538, iSSN: 2324-9013.

[32] H. Guo, W. Li, and M. Nejad, "A Hierarchical and Location-Aware Consensus Protocol for IoT-Blockchain Applications," *IEEE Transactions on Network and Service Management*, vol. 19, no. 3, pp. 2972–2986, Sep. 2022, conference Name: IEEE Transactions on Network and Service Management.

[33] W. Pourmajidi, L. Zhang, J. Steinbacher, T. Erwin, and A. Miranskyy, "Immutable Log Storage as a Service on Private and Public Blockchains," *IEEE Transactions on Services Computing*, pp. 1–1, 2021, conference Name: IEEE Transactions on Services Computing.

[34] Y. Wang, Z. Su, Q. Xu, R. Li, and T. H. Luan, "Lifesaving with RescueChain: Energy-Efficient and Partition-Tolerant Blockchain Based Secure Information Sharing for UAV-Aided Disaster Rescue," in *IEEE INFOCOM 2021 - IEEE Conference on Computer Communications*, May 2021, pp. 1–10, iSSN: 2641-9874.

[35] H. Wang, "Que Bian: An Electronic Medical Record Management System on Blockchain," Aug. 2021, arXiv:2105.07327 [cs]. [Online]. Available: http://arxiv.org/abs/2105.07327

[36] S. Küfeoğlu and M. Özkuran, "Bitcoin mining: A global review of energy and power demand," *Energy Research & Social Science*, vol. 58, p. 101273, Dec. 2019. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S2214629619305948

[37] "Proof-of-stake (PoS)." [Online]. Available: https://ethereum.org

[38] Q. Nguyen and A. Cronje, "ONLAY: Online Layering for scalable asynchronous BFT system," May 2019, arXiv:1905.04867 [cs]. [Online]. Available: http://arxiv.org/abs/1905.04867

[39] "Hyperledger Fabric Introduction." [Online]. Available: https://hyperledger-fabric.readthedocs.io/en/release-2.5/whatis.html

[40] J. Chen and Y. Qin, "Reducing Block Propagation Delay in Blockchain Networks via Guarantee Verification," in *2021 IEEE 29th International Conference on Network Protocols (ICNP)*, Nov. 2021, pp. 1–6, iSSN: 2643-3303.

[41] K. Lai, "*Intro: Self-Host OSCAR # - World OSCAR," Jun. 2021. [Online]. Available: https://worldoscar.org/knowledge-base/intro-self-host-oscar/

[42] P. Singhal and S. Masih, "MetaAnalysis of Methods for Scaling Blockchain Technology for Automotive Uses," Jul. 2019, arXiv:1907.02602 [cs, eess]. [Online]. Available: http://arxiv.org/abs/1907.02602

[43] H. Hellani, L. Sliman, A. E. Samhat, and E. Exposito, "Tangle the Blockchain:Towards Connecting Blockchain and DAG," in *2021 IEEE 30th International Conference on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE)*, Oct. 2021, pp. 63–68, iSSN: 2641-8169.

[44] "IOTA Wiki." [Online]. Available: https://wiki.iota.org/legacy

[45] A. Rawat, V. Daza, and M. Signorini, "Offline Scaling of IoT Devices in IOTA Blockchain," *Sensors*, vol. 22, no. 4, p. 1411, Jan. 2022, number: 4 Publisher: Multidisciplinary Digital Publishing Institute. [Online]. Available: https://www.mdpi.com/1424-8220/22/4/1411

[46] J. Huang, Y. W. Qi, M. R. Asghar, A. Meads, and Y.-C. Tu, "Sharing medical data using a blockchain-based secure EHR system for New Zealand," *IET Blockchain*, vol. 2, no. 1, pp. 13–28, 2022, _eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1049/blc2.12012. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1049/blc2.12012

[47] K. T. Akhter Md Hasib, I. Chowdhury, S. Sakib, M. Monirujjaman Khan, N. Alsufyani, A. Alsufyani, and S. Bourouis, "Electronic Health Record Moni-

toring System and Data Security Using Blockchain Technology," *Security and Communication Networks*, vol. 2022, p. e2366632, Feb. 2022, publisher: Hindawi. [Online]. Available: https://www.hindawi.com/journals/scn/2022/2366632/

[48] A. Shahnaz, U. Qamar, and A. Khalid, "Using Blockchain for Electronic Health Records," *IEEE Access*, vol. 7, pp. 147 782–147 795, 2019, conference Name: IEEE Access.

[49] J. King, "Measuring the forensic-ability of audit logs for nonrepudiation," in *2013 35th International Conference on Software Engineering (ICSE)*, May 2013, pp. 1419–1422, iSSN: 1558-1225.

[50] J. King, R. Pandita, and L. Williams, "Enabling forensics by proposing heuristics to identify mandatory log events," in *Proceedings of the 2015 Symposium and Bootcamp on the Science of Security*. Urbana Illinois: ACM, Apr. 2015, pp. 1–11. [Online]. Available: https://dl.acm.org/doi/10.1145/2746194.2746200

[51] K. Kent and M. Souppaya, "Guide to Computer Security Log Management," National Institute of Standards and Technology, Tech. Rep. NIST Special Publication (SP) 800-92, Sep. 2006. [Online]. Available: https://csrc.nist.gov/publications/detail/sp/800-92/final

[52] J. King and L. Williams, "Log your CRUD: design principles for software logging mechanisms," in *Proceedings of the 2014 Symposium and Bootcamp on the Science of Security*, ser. HotSoS '14. New York, NY, USA: Association for Computing Machinery, Apr. 2014, pp. 1–10. [Online]. Available: https://dl.acm.org/doi/10.1145/2600176.2600183

[53] "What Is CRUD? Create, Read, Update, and Delete | CrowdStrike." [Online]. Available: https://www.crowdstrike.com/cybersecurity-101/observability/crud/

[54] W. Pourmajidi and A. Miranskyy, "Logchain: Blockchain-Assisted Log Storage," in *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*, Jul. 2018, pp. 978–982, iSSN: 2159-6190.

[55] "Logging - OWASP Cheat Sheet Series." [Online]. Available: https://cheatsheetseries.owasp.org/cheatsheets/Logging_Cheat_Sheet.html

[56] R. T. Snodgrass, S. S. Yao, and C. Collberg, "Tamper detection in audit logs," in *Proceedings of the Thirtieth international conference on Very large data bases - Volume 30*, ser. VLDB '04. Toronto, Canada: VLDB Endowment, Aug. 2004, pp. 504–515.

[57] "When North Korean hackers almost pulled off a billion-dollar heist from Bangladesh Bank | The Daily Star." [Online]. Available: https://www.thedailystar.net/toggle/news/when-north-korean-hackers-almost-pulled-billion-dollar-heist-bangladesh-bank-2115317

[58] author unknown, "The Lazarus heist: How North Korea almost pulled off a billion-dollar hack," *BBC News*, Jun. 2021. [Online]. Available: https://www.bbc.com/news/stories-57520169

[59] E. Coiera, *Guide to health informatics*, 3rd ed. Boca Raton: CRC Press, Taylor & Francis Group, 2015, oCLC: 900869742.

[60] "Commonwealth Fund survey, 2022 | CIHI." [Online]. Available: https://www.cihi.ca/en/commonwealth-fund-survey-2022

[61] M. Chernyshev, S. Zeadally, and Z. Baig, "Healthcare Data Breaches: Implications for Digital Forensic Readiness," *Journal of Medical Systems*, vol. 43, no. 1, p. 7, Nov. 2018.

[62] "EHRS-Blueprint (v2) (Full) | Canada Health Infoway," Mar. 2006. [Online]. Available: https://www.infoway-inforoute.ca/en/component/edocman/resources/technical-documents/391-ehrs-blueprint-v2-full

[63] R. Gifty, R. Bharathi, and P. Krishnakumar, "Privacy and security of big data in cyber physical systems using Weibull distribution-based intrusion detection," *Neural Computing and Applications*, vol. 31, no. 1, pp. 23–34, Jan. 2019. [Online]. Available: https://doi.org/10.1007/s00521-018-3635-6

[64] ASTM, "Standard Specification for Audit and Disclosure Logs for Use in Health Information Systems," Mar. 2019. [Online]. Available: https://compass.astm.org/document/?contentCode=ASTM%7CE2147-%7Cen-US&proxycl=https%3A%2F%2Fsecure.astm.org&fromLogin=true

[65] A. Allavena, A. Demers, and J. E. Hopcroft, "Correctness of a gossip based membership protocol," in *Proceedings of the twenty-fourth annual ACM symposium on Principles of distributed computing*. Las Vegas NV USA: ACM, Jul. 2005, pp. 292–301. [Online]. Available: https://dl.acm.org/doi/10.1145/1073814.1073871

[66] A. Ghodsi, S. Haridi, and H. Weatherspoon, "Exploiting the synergy between gossiping and structured overlays," *ACM SIGOPS Operating Systems Review*, vol. 41, no. 5, pp. 61–66, Oct. 2007. [Online]. Available: https://dl.acm.org/doi/10.1145/1317379.1317389

[67] M. Asplund and S. Nadjm-Tehrani, "A Partition-Tolerant Manycast Algorithm for Disaster Area Networks," in *2009 28th IEEE International Symposium on Reliable Distributed Systems*, Sep. 2009, pp. 156–165, iSSN: 1060-9857.

[68] O. Chipara, W. G. Griswold, A. N. Plymoth, R. Huang, F. Liu, P. Johansson, R. Rao, T. Chan, and C. Buono, "WIISARD: a measurement study of network properties and protocol reliability during an emergency response," in *Proceedings of the 10th international conference on Mobile systems, applications, and services*, ser. MobiSys '12. New York, NY, USA: Association for Computing Machinery, Jun. 2012, pp. 407–420. [Online]. Available: https://dl.acm.org/doi/10.1145/2307636.2307674

[69] "Channels — hyperledger-fabricdocs main documentation." [Online]. Available: https://hyperledger-fabric.readthedocs.io/en/release-2.5/channels.html

[70] P. Hutten-Czapski, "Installation of OSCAR 19 on Ubuntu 22.04 - World OSCAR," Mar. 2022. [Online]. Available: https://worldoscar.org/knowledge-base/oscar-19-on-ubuntu-22-04/

[71] McMaster, "OSCAR-EMR." [Online]. Available: https://fammed.mcmaster.ca/oscar-emr/

[72] "openoscar / oscar — Bitbucket." [Online]. Available: https://bitbucket.org/openoscar/oscar/src/master/

[73] "OpenEMR Features - OpenEMR Project Wiki." [Online]. Available: https://www.open-emr.org/wiki/index.php/OpenEMR_Features

[74] "Welcome to OpenEMR." [Online]. Available: https://www.open-emr.org/welcome/

[75] "Introduction to OpenMRS - Documentation - OpenMRS Wiki." [Online]. Available: https://wiki.openmrs.org/display/docs/Introduction+to+OpenMRS

[76] "Access Logging Module - Documentation - OpenMRS Wiki." [Online]. Available: https://wiki.openmrs.org/display/docs/Access+Logging+Module

[77] "Usage Statistics Module - Documentation - OpenMRS Wiki." [Online]. Available: https://wiki.openmrs.org/display/docs/Usage+Statistics+Module

[78] Y.-A. d. Montjoye, L. Radaelli, V. K. Singh, and A. S. Pentland, "Unique in the shopping mall: On the reidentifiability of credit card metadata," *Science*, vol. 347, no. 6221, pp. 536–539, 2015, _eprint: https://www.science.org/doi/pdf/10.1126/science.1256297. [Online]. Available: https://www.science.org/doi/abs/10.1126/science.1256297

[79] L. Navaratna, N. Wijesinghe, and U. Pilapitiya, "Providing Electronic Health Care Services Through A Private Permissioned Blockchain," in *2020 2nd International Conference on Advancements in Computing (ICAC)*, vol. 1, Dec. 2020, pp. 144–149.

[80] O. Wu, S. Li, H. Zhang, L. Liu, Y. Wang, and H. Li, "An Optimized Scheduling Algorithm for the Multi-channel Hyperledger Fabric," in *2022 IEEE Intl Conf on Parallel & Distributed Processing with Applications, Big Data & Cloud Computing, Sustainable Computing & Communications, Social Computing & Networking (ISPA/BDCloud/SocialCom/SustainCom)*, Dec. 2022, pp. 636–643.

[81] Y. Tao, B. Li, J. Jiang, H. C. Ng, C. Wang, and B. Li, "On Sharding Open Blockchains with Smart Contracts," in *2020 IEEE 36th International Conference on Data Engineering (ICDE)*, Apr. 2020, pp. 1357–1368, iSSN: 2375-026X.

[82] I. T. L. Computer Security Division, "Hash Functions | CSRC | CSRC," Jan. 2017. [Online]. Available: https://csrc.nist.gov/projects/hash-functions

[83] "pages.introduction." [Online]. Available: https://hyperledger.github.io/firefly/v1.2.0/overview/supernode_concept.html

[84] "Public and Permissioned." [Online]. Available: https://hyperledger.github.io/firefly/v1.2.0/overview/public_vs_permissioned.html

[85] "Writing Your First Chaincode — hyperledger-fabricdocs main documentation." [Online]. Available: https://hyperledger-fabric.readthedocs.io/en/latest/chaincode4ade.html

[86] "Work with Hyperledger Fabric chaincodes." [Online]. Available: https://hyperledger.github.io/firefly/v1.2.0/tutorials/custom_contracts/fabric.html

Appendix A

Additional Information

A.1 OSCAR EMR

```
MariaDB [oscar]> DESC log
    -> ;
+----------------+--------------+------+-----+---------+----------------+
| Field          | Type         | Null | Key | Default | Extra          |
+----------------+--------------+------+-----+---------+----------------+
| id             | bigint(20)   | NO   | PRI | NULL    | auto_increment |
| dateTime       | datetime     | NO   | MUL | NULL    |                |
| provider_no    | varchar(10)  | YES  |     | NULL    |                |
| action         | varchar(100) | YES  | MUL | NULL    |                |
| content        | varchar(80)  | YES  | MUL | NULL    |                |
| contentId      | varchar(80)  | YES  | MUL | NULL    |                |
| ip             | varchar(64)  | YES  |     | NULL    |                |
| demographic_no | int(10)      | YES  | MUL | NULL    |                |
| data           | text         | YES  |     | NULL    |                |
| securityId     | int(11)      | YES  |     | NULL    |                |
+----------------+--------------+------+-----+---------+----------------+
10 rows in set (0.00 sec)
```

Figure A.1: Screenshot of the log storage for OSCAR EMR, version 15, from the openOSP repository

Sample entries in this database table, from a sample instance of OSCAR. This is a small subset of the logs that were used to create a log entry generator to simulate and test the ABLOC system.

LogID	date Time	Provider ID	Action	content	content ID	ip	Patient ID	data	security ID
33	2020-10-21 14:02:17	999998	log in	login	\N	10.0.2.2	\N	\N	\N
34	2020-10-21 14:02:17	999998	log in	login	facilityId=1	10.0.2.2	\N	\N	\N
92	2020-10-21 14:12:36	999998	log out	login	\N	10.0.2.2	\N	\N	\N
111	2020-10-21 14:17:37	N/A	REST WS: ResourceService. getLUCodeFileListFromK2A	https://127.0.0.1:8444/oscar/ws/rs/resources/luCodesList	\N	10.0.2.2	\N	{}	\N
112	2020-10-21 14:17:37	N/A	REST WS: ResourceService. getPreventionRulesListFromK2A	https://127.0.0.1:8444/oscar/ws/rs/resources/preventionRulesList	\N	10.0.2.2	\N	{}	\N
142	2020-10-21 14:45:37	\N	failed	login	oscardoc	10.0.2.2	\N	\N	\N
157	2020-10-21 15:07:42	999998	add	adminAddUser	\N	10.0.2.2	\N	\N	\N
158	2020-10-21 15:08:05	999998	add	securityRecord	drw	10.0.2.2	\N	\N	\N
159	2020-10-21 15:08:14	999998	add	role	1—admin	10.0.2.2	\N	\N	\N
160	2020-10-21 15:08:18	999998	add	role	1—doctor	10.0.2.2	\N	\N	\N
281	2020-10-21 16:22:38	1	DemographicManager. getDemographic	\N	\N	10.0.2.2	2	\N	129
282	2020-10-21 16:22:38	1	DemographicManager. getDemographic	\N	\N	10.0.2.2	2	\N	129
283	2020-10-21 16:22:38	1	DemographicManager. getDemographic	\N	\N	10.0.2.2	2	\N	129

A.2 Open EMR

The Open EMR logging database contains a table that stores the main logging information. This information is encrypted and can be viewed in a decrypted state within the EMR's log access page. Because of the encryption, no logs were exported.

```
MariaDB [openemr]> DESC log;
+--------------+--------------+------+-----+----------+----------------+
| Field        | Type         | Null | Key | Default  | Extra          |
+--------------+--------------+------+-----+----------+----------------+
| id           | bigint(20)   | NO   | PRI | NULL     | auto_increment |
| date         | datetime     | YES  |     | NULL     |                |
| event        | varchar(255) | YES  |     | NULL     |                |
| category     | varchar(255) | YES  |     | NULL     |                |
| user         | varchar(255) | YES  |     | NULL     |                |
| groupname    | varchar(255) | YES  |     | NULL     |                |
| comments     | longtext     | YES  |     | NULL     |                |
| user_notes   | longtext     | YES  |     | NULL     |                |
| patient_id   | bigint(20)   | YES  | MUL | NULL     |                |
| success      | tinyint(1)   | YES  |     | 1        |                |
| checksum     | longtext     | YES  |     | NULL     |                |
| crt_user     | varchar(255) | YES  |     | NULL     |                |
| log_from     | varchar(20)  | YES  |     | open-emr |                |
| menu_item_id | int(11)      | YES  |     | NULL     |                |
| ccda_doc_id  | int(11)      | YES  |     | NULL     |                |
+--------------+--------------+------+-----+----------+----------------+
15 rows in set (0.001 sec)
```

Figure A.2: Screenshot of the log storage for Open EMR

```
MariaDB [openemr]> DESC log_comment_encrypt;
+--------------+------------------+------+-----+---------+----------------+
| Field        | Type             | Null | Key | Default | Extra          |
+--------------+------------------+------+-----+---------+----------------+
| id           | int(11)          | NO   | PRI | NULL    | auto_increment |
| log_id       | int(11)          | NO   |     | NULL    |                |
| encrypt      | enum('Yes','No') | NO   |     | No      |                |
| checksum     | longtext         | YES  |     | NULL    |                |
| checksum_api | longtext         | YES  |     | NULL    |                |
| version      | tinyint(4)       | NO   |     | 0       |                |
+--------------+------------------+------+-----+---------+----------------+
```

Figure A.3: Screenshot of the log support table for Open EMR

Within the EMR there is an interface for accessing the logs in an unencrypted format. This allows the user to filter through and find the logs that are of interest; the EMR handles the decryption of the log entries.

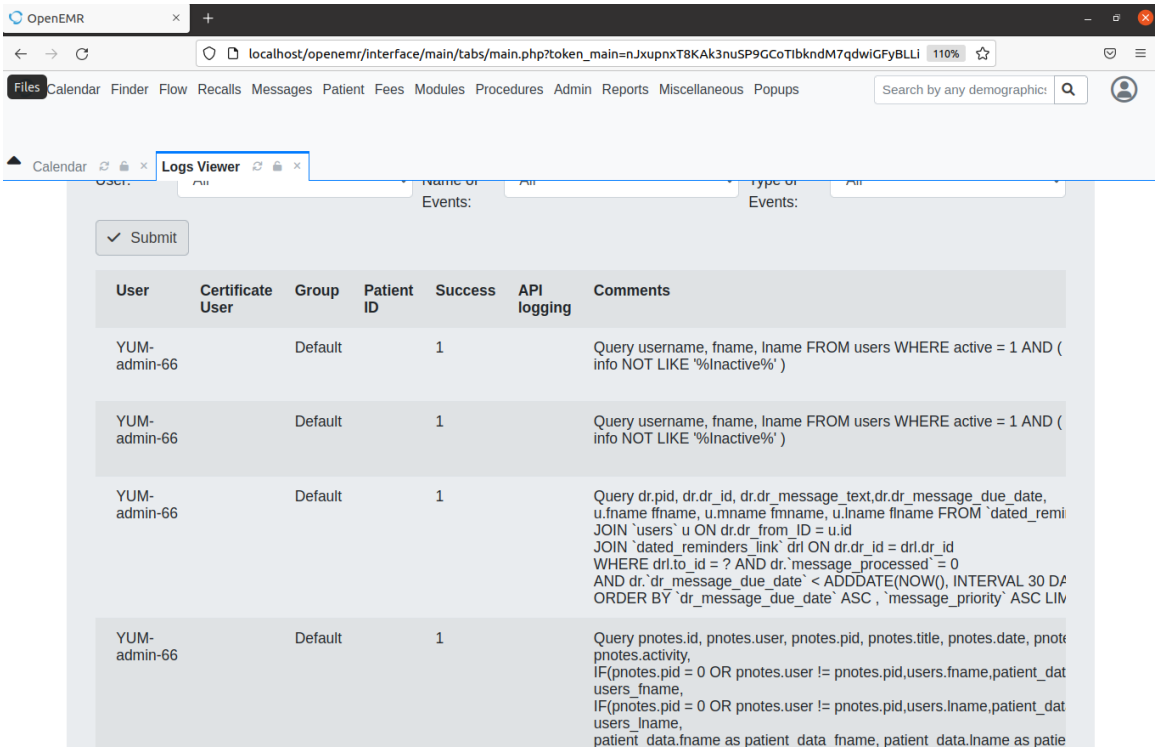

Figure A.4: Screenshot of the Log Access GUI for Open EMR

A.3 Open MRS

Open MRS has no logging by default. Adding the Usage Statistics Module [77] results in the following interface being added. No detailed logging information could be found in the database.

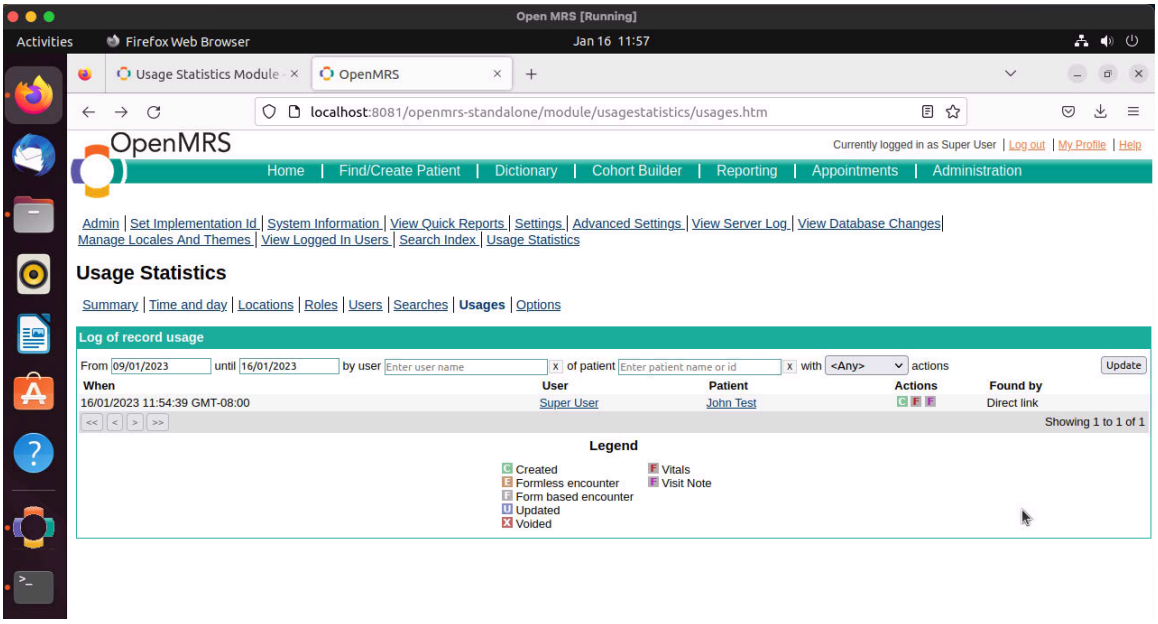

Figure A.5: A screenshot of the GUI used in Open MRS for accessing the log records.

A.4 Scalability Proof

In 5.2.3.2 the relationship between the rounds of gossip and the number of nodes who should be informed, baring network failures is discussed theoretically. This proof is to support that discussion.

Suppose we have a network running the gossip protocol, we will prove by induction that after n rounds of gossip, 2^n nodes know about the message that was sent.

Base Case

At gossip round 0, only 1 node (the sending node) knows about the message. $2^0 = 1$
At gossip round 1, only two nodes know about the message. $2^1 = 2$

Assume n = k

Assume that P(n), the number of nodes who know about the message after n rounds of gossip, equals 2^n and that this holds true for n = k. Therefore: P(k) = 2^k P(k) = $2 * (2^{k-1})$

Inductive Step

P(k+1) = 2^{k+1} P(k+1) = $2 * (2^k)$ P(k+1) = $2 * 2 * (2^{k-1})$ P(K+1) = $2 * P(k)$ where P(k) = $2 * (2^{k-1})$ from above The result follows by induction. ■

A.5 Mininet Gossip Protocol Testing

Mininet custom topology for testing a single offline node and for sharding the network.

```
from mininet.net import Mininet
from mininet.examples.clustercli import ClusterCLI as CLI
from mininet.node import OVSBridge
from mininet.topo import Topo
import csv

CSVHEADER = ['IP Addr', 'nodeID', 'port']

class NetworkTopo( Topo ):
    def build(self):

        # Add hosts and switches
        h0 = self.addHost( 'h0', ip='10.0.0.1')
        h1 = self.addHost( 'h1', ip='10.0.0.2' )
        h2 = self.addHost( 'h2', ip='10.0.0.3' )
        h3 = self.addHost( 'h3', ip='10.0.0.4' )
        h4 = self.addHost( 'h4', ip='10.0.0.5' )
        s1 = self.addSwitch( 's1', cls=OVSBridge )
        s0 = self.addSwitch( 's0', cls=OVSBridge )

        # Add links
        L0 = self.addLink( h0, s0, delay='5ms')
        L2 = self.addLink( h2, s0, delay='5ms' )
        L1 = self.addLink( h1, s1, delay='5ms' )
        L3 = self.addLink( h3, s1, delay='5ms' )
        L4 = self.addLink( h4, s0, delay='5ms' )
        LS = self.addLink( s1, s0, delay='5ms' )


        with open("network.csv", "w") as f:
            writer = csv.writer(f)
            writer.writerow(CSVHEADER)

            data = ['10.0.0.1', "0", "9876"]
            writer.writerow(data)
            data = ['10.0.0.2', "1", "9875"]
            writer.writerow(data)
            data = ['10.0.0.3', "2", "9856"]
            writer.writerow(data)
            data = ['10.0.0.4', "3", "9775"]
            writer.writerow(data)
            data = ['10.0.0.5', "4", "9777"]
            writer.writerow(data)


def run():
    topo = NetworkTopo()

    net = Mininet( topo=topo, controller=None)

    net.start()
    CLI(net)
    net.stop()

topos = { 'mytopo': ( lambda: NetworkTopo() ) }
```

A.6 Initial Gossip Protocol Mininet Scalability Topology Setup

This is the code used to generate the data for the scalability experiments.

```
from mininet.net import Mininet
from mininet.examples.clustercli import ClusterCLI as CLI
from mininet.node import OVSBridge
import frontierCompare as fc
import csv
import random
import time
import signal

CSVHEADER = ['IP Addr', 'nodeID', 'port']

net = Mininet()
hosts = []
hostSwitches = []
hCount = 0
sCount = 0

def genHst(numb = 5):
    global Spos
    global sCount
    global hCount
    global net
    global hosts
    global hostSwitches

    Spos = sCount
    Sname = "s" + str(sCount)
    sCount = sCount + 1
    hostSwitches.append(net.addSwitch(Sname, cls=OVSBridge))
    print("Added switch")

    for _ in range(numb):
        name = str(hCount)
        print(name)
        hosts.append(net.addHost(name))
        net.addLink(hostSwitches[Spos], hosts[hCount], delay='5ms')
        hCount = hCount + 1
        print("Added 1 host")

#modify this varible to control how many nodes are in the network.
toGen = 100
DIR = "./Demo/"
genHst(toGen//2)
genHst(toGen//2)

if len(hostSwitches) > 1:
    for switch1 in hostSwitches:
        for switch2 in hostSwitches:
            if switch1 != switch2 and len(net.linksBetween(switch1, switch2)) == 0:
                net.addLink(switch1, switch2, delay='5ms')
                print("Need to add link")

net.start()
# Created all needed hosts etc. Now write to file so that the information
### can be passed to all gossip nodes

with open("network.csv", "w") as f:
    writer = csv.writer(f)
    writer.writerow(CSVHEADER)

    data = ["", "", ""]
    hID = 0
    for hst in hosts:
```

```
        data[0] = str(hst.IP())
        data[1] = hID
        hID = hID +1
        data[2] = random.randint(9000, 9999)
        writer.writerow(data)

print("Done Generating file - run the test Now")

#Run the gossip protocol
for hst in hosts:
    #send command to run script in the background -> write output to file with its ipAddress
    Hid = str(hst)
    HIP = hst.IP()
    Hcmd =  "python3 ./gossip597/gossip.py " + Hid + " &>" + DIR + HIP +"_" + str(toGen) + ".txt &"
    print(Hcmd)
    hst.sendCmd(Hcmd)

waited = 0
#how long to wait in mins
STOPVALMin = 20

STOPVAL = STOPVALMin * 60
print("Starting")
while waited < STOPVAL:
    time.sleep(1)
    if waited % 30 == 0:
        print(f"Allive have done {waited/60} mins of {STOPVALMin}")
    waited = waited + 1
print("About to End - please be patient while SIGNALS are sent and handled")

for hst in hosts:
    hst.sendInt()
time.sleep(80)

print("Everything should be terminated - Now the script will check the frontier sets")
fs = fc.getData(DIR)
fc.compareData(fs)

CLI(net)
net.stop()
print("Stopped")
```

A.7 Smart Contract

The smart contract was written in Go and uses the Hyperledger Fabric contract API to interface with FireFly Fabric blockchain.

```
// This smart contract was written as part of the ABLOC system for my Thesis.
// Copyright Joseph A. Krysl
// Spring 2023

// Please note that this was written and tested before the change to use Realms instead of clinics. It is possible to use clinics and realms interchangeably in this document.

package chaincode

import (
 "crypto/sha256"
 "encoding/hex"
 "encoding/json"
 "fmt"
 "sort"
 "strings"
 "time"

 "github.com/hyperledger/fabric-contract-api-go/contractapi"
 "github.com/tendermint/tendermint/crypto/merkle"
 //"golang.org/x/crypto/sha3"
)

// Smart contract Requirements
// SimpleContract contract for handling writing and reading from the world state
type SmartContract struct {
 contractapi.Contract
}

// Structs for Level1
// map[string]map[string]string
type Level1 struct {
 Level     string                      `json:"Level"`
 ClinicID  string                      `json:"Clinic_UUID"`
 StartTime string                      `json:"Start Time"`
 EndTime   string                      `json:"End Time"`
 Data      map[string]map[string]string //`json:"log_data_in"`
}

type Level2 struct {
 Level          string              `json:"Level"`
 ClinicID       string              `json:"Clinic_UUID"`
 StartTime      string              `json:"Start Time"`
 EndTime        string              `json:"End Time"`
 Merkle_Data    map[string]string   `json:"Merkle_Map"`           // map of patient id to merkle root of all logs for that patient that were summarized. These were added to a merkle tree in order by log id
 HashedBlock    map[string]string   `json:"Blocks Hashed"`        //hash of each block's data that was summarized. Done in a JSON format with no whitespace.
 PatientToLogIDs map[string][]string `json:"Patient To LogID List"` //what logs are included for the patient
}

// ******************** Private Functions/No API access needed ******************** \\
// Private function that generates an ID string from the input criteria. This logic will be shared to other smart contracts/standardized.
func createID(start string, end string, UUID string, level string) string {
 if len(start) > 0 && len(end) > 0 && len(UUID) > 0 && len(level) > 0 {
  to_rt := UUID + "--" + level + "--" + start + "--" + end
  return to_rt
 } else {
  return ""
 }
}

// Parses time from a string in the BLOC defined format.
func parseTime(in_time string) (time.Time, error) {
```

```
 location, _ := time.LoadLocation("America/Los_Angeles")

 layout_NOms := "2006-01-02 15:04:05"
 t, err := time.ParseInLocation(layout_NOms, in_time, location)
 if err != nil {
  return t, err
 }
 return t, nil
}

// Private function to see if a block with a given ID already exists on the blockchain.
func (s *SmartContract) assetExists(ctx contractapi.TransactionContextInterface, id string) (bool, error) {
 assetJSON, err := ctx.GetStub().GetState(id)
 if err != nil {
  return false, fmt.Errorf("failed to read from world state: %v", err)
 }

 return assetJSON != nil, nil
}

// readAsset returns the asset stored in the world state with given id.
func (s *SmartContract) readL1Block(ctx contractapi.TransactionContextInterface, id string) (*Level1, error) {
 assetJSON, err := ctx.GetStub().GetState(id)
 if err != nil {
  return nil, fmt.Errorf("failed to read from world state: %v", err)
 }
 if assetJSON == nil {
  return nil, fmt.Errorf("the asset %s does not exist", id)
 }

 var asset Level1
 err = json.Unmarshal(assetJSON, &asset)
 if err != nil {
  return nil, err
 }

 return &asset, nil
}

// ********************* Level 1 Specific Functions ********************* \\
// Get all Level1 blocks with a certain clinic UUID after a time and returns them
func (s *SmartContract) getClinicLevel1BlocksAfterTime(ctx contractapi.TransactionContextInterface, endTime string, UUID string)
↪  ([]*Level1, error) {
 blocks, err := s.GetAllLevel1(ctx)

 if err != nil {
  return nil, err
 }
 //convert time to objects and compare

 end_timeOB, err := parseTime(endTime)
 if err != nil {
  return nil, err
 }

 var clinicBlocks []*Level1

 for _, b := range blocks {
  L1struc := *b
  BlockEndTime, _ := parseTime(L1struc.EndTime)
  if L1struc.ClinicID == UUID && BlockEndTime.After(end_timeOB) {
   clinicBlocks = append(clinicBlocks, b)
  }
 }

 return clinicBlocks, nil
}

// sets the UUID for the clinic
func (s *SmartContract) setUUID(ctx contractapi.TransactionContextInterface, id string) error {
```

```
 toSave, err := json.Marshal(id)
 if err != nil {
  return err
 }
 err = ctx.GetStub().PutState("LocalUUID", toSave)

 return err
}

// This function was used for development and could be changed to a private function. It was needed because this logic was being applied
↪  to multiple different clinic identifiers and being able to see which is which was advantageous
// It is left here as a reference. To use this with a firefly api you must add an entry into the FFI format document.
func (s *SmartContract) getLocalUUID(ctx contractapi.TransactionContextInterface) (string, error) {
 uuid, err := ctx.GetStub().GetState("LocalUUID")
 if err != nil {
  return "", err
 }
 var uuid_str string
 err = json.Unmarshal(uuid, &uuid_str)
 return uuid_str, err
}

// This assumes that there is at least one previous log blocks stored on the blockchain.
// For some reason I can not get firefly to accept not receiving a map of objects with the logs, thus, log_data_in is a array with one
↪  element in it.
// For the purposes of the prototype, where the clinic/realm identifier is changing we will have this to get and set the local
↪  identifiers.
func (s *SmartContract) CreateLevel1(ctx contractapi.TransactionContextInterface, start_date string, end_date string, clinic_UUID
↪  string, log_data_in []map[string]map[string]string) error {
 err := s.setUUID(ctx, clinic_UUID)
 if err != nil {
  return err
 }
 l1id := createID(start_date, end_date, clinic_UUID, "Level1")
 //check to see if the ID already exists
 exists, err := s.assetExists(ctx, l1id)
 if err != nil {
  return err
 }
 if exists {
  return fmt.Errorf("the Block %s already exists. Is this a duplicate?", l1id)
 }

 //var log_data_struct L1LogData
 //err = json.Unmarshal([]byte(log_data_in), &log_data_struct)

 if err != nil {
  fmt.Println(err)
 }

 l1 := Level1{
  Level:     "Level1",
  ClinicID:  clinic_UUID,
  StartTime: start_date,
  EndTime:   end_date,
  Data:      log_data_in[0]}

 L1JSON, err := json.Marshal(l1)
 if err != nil {
  return err
 }

 err = ctx.GetStub().PutState(l1id, L1JSON)

 if err != nil {
  return err
 }
 NeedL2Block, err := s.summaryNeeded(ctx)
 if err != nil {
  return err
 }
```

```
 if NeedL2Block {
  //create a new L2 Block
  //get all level1 not in a block up to current time
  //call the function to create a block with the above in an array

  lastBlock, err := s.getLastL2Block(ctx, clinic_UUID)

  if err != nil {
   return err
  }
  if lastBlock == nil {
   //no past block
   //create first past block with all Level 1
   all_L1, _ := s.GetAllLevel1(ctx)
   all_L1 = append(all_L1, &l1)
   s.createLevel2(ctx, clinic_UUID, all_L1)
  } else {
   //get all level1 not in a block up to current time
   //call the function to create a block with the above in an array
   to_include, _ := s.getClinicLevel1BlocksAfterTime(ctx, lastBlock.EndTime, clinic_UUID)
   to_include = append(to_include, &l1)
   s.createLevel2(ctx, clinic_UUID, to_include)
  }
 }

 return ctx.GetStub().SetEvent("Level1 Genesis Created", L1JSON)
}

// Public function
// GetAllAssets returns all assets found in world state
func (s *SmartContract) GetAllLevel1(ctx contractapi.TransactionContextInterface) ([]*Level1, error) {
 // range query with empty string for startKey and endKey does an
 // open-ended query of all assets in the chaincode namespace.
 resultsIterator, err := ctx.GetStub().GetStateByRange("", "")
 if err != nil {
  return nil, err
 }
 defer resultsIterator.Close()

 var assets []*Level1
 for resultsIterator.HasNext() {
  queryResponse, err := resultsIterator.Next()
  if err != nil {
   return nil, err
  }

  //check the id of the iterator
  id := queryResponse.Key

  if id == "LocalUUID" || strings.Contains(id, "Level2") || strings.Contains(id, "Frontier") || !strings.Contains(id, "Level1") {
   continue
  }

  var log Level1
  err = json.Unmarshal(queryResponse.Value, &log)
  if err != nil {
   return nil, err
  }
  if log.Level == "Level1" {
   assets = append(assets, &log)
  }
 }

 return assets, nil
}

// Public function to return a specific block (used for validating logs)
func (s *SmartContract) GetLevel1Block(ctx contractapi.TransactionContextInterface, start_date string, end_date string, clinic_UUID
↪  string) (*Level1, error) {
 l1id := createID(start_date, end_date, clinic_UUID, "Level1")
```

```
 log, err := s.readL1Block(ctx, l1id)

 if log != nil {
  return log, nil
 } else {
  return nil, err
 }
}

// goes through all the blocks and returns the log hash of interest.
func (s *SmartContract) GetLogHashWithLogID(ctx contractapi.TransactionContextInterface, logID string) (string, error) {
 //get all blocks that are in Level1
 blocks, err := s.GetAllLevel1(ctx)
 if err != nil {
  return "", err
 }

 //find the entry we are interested in.
 for _, val := range blocks {
  log := *val

  // need to check for level 2 here...

  for _, map_of_logs := range log.Data {
   for lID, logHash := range map_of_logs {
    if logID == lID {
     return logHash, nil
    }
   }
  }
 }
 return "", nil
}

// ***************************** Level 2 Specific Functions *****************************
// *****************************  ********************** *****************************

// Public function to return a specific block (used for validating logs)
func (s *SmartContract) GetLevel2Block(ctx contractapi.TransactionContextInterface, start_date string, end_date string, clinic_UUID
↪  string) (*Level2, error) {
 l2id := createID(start_date, end_date, clinic_UUID, "Level2")
 log, err := s.readL2Block(ctx, l2id)

 if log != nil {
  return log, nil
 } else {
  return nil, err
 }
}

// GetAllAssets returns all assets found in world state
func (s *SmartContract) GetAllLevel2(ctx contractapi.TransactionContextInterface) ([]*Level2, error) {
 // range query with empty string for startKey and endKey does an
 // open-ended query of all assets in the chaincode namespace.
 resultsIterator, err := ctx.GetStub().GetStateByRange("", "")
 if err != nil {
  return nil, err
 }
 defer resultsIterator.Close()

 var assets []*Level2
 for resultsIterator.HasNext() {
  queryResponse, err := resultsIterator.Next()
  if err != nil {
   return nil, err
  }

  //check the id of the iterator
  id := queryResponse.Key

  if id == "LocalUUID" || strings.Contains(id, "--Last Level2") || strings.Contains(id, "Frontier") || !strings.Contains(id, "Level2") {
```

```
    continue
   }

   var log Level2
   err = json.Unmarshal(queryResponse.Value, &log)
   if err != nil {
    return nil, err
   }
   if log.Level == "Level2" {
    assets = append(assets, &log)
   }
  }

  return assets, nil
 }

 // // Get all Level2 blocks with a certain UUID/realm identifier and returns them. This can be used to ensure that all realms are in sync
 ↪  with one another.
 func (s *SmartContract) GetClinicLevel2Blocks(ctx contractapi.TransactionContextInterface, UUID string) ([]*Level2, error) {
  blocks, err := s.GetAllLevel2(ctx)

  if err != nil {
   return nil, err
  }

  var clinicBlocks []*Level2

  for _, b := range blocks {
   L2struc := *b
   if L2struc.ClinicID == UUID {
    clinicBlocks = append(clinicBlocks, b)
   }
  }

  return clinicBlocks, nil
 }

 // // Get all Level2 blocks with a certain clinic/realm UUID after a time and returns them
 func (s *SmartContract) GetClinicLevel2BlocksAfterTime(ctx contractapi.TransactionContextInterface, endTime string, UUID string)
 ↪  ([]*Level2, error) {
  blocks, err := s.GetAllLevel2(ctx)

  if err != nil {
   return nil, err
  }
  //convert time to objects and compare

  end_timeOB, err := parseTime(endTime)
  if err != nil {
   return nil, err
  }

  var clinicBlocks []*Level2

  for _, b := range blocks {
   L2struc := *b
   BlockEndTime, _ := parseTime(L2struc.EndTime)
   if L2struc.ClinicID == UUID && BlockEndTime.After(end_timeOB) {
    clinicBlocks = append(clinicBlocks, b)
   }
  }

  return clinicBlocks, nil
 }

 // Gets the last Level2 block with a certain clinic/realm UUID and returns it. This is mainly used for development. It was public but was
 ↪  made private as it is also a helper function.
 func (s *SmartContract) getLastL2Block(ctx contractapi.TransactionContextInterface, clinic_UUID string) (*Level2, error) {
  LastL2_clin := clinic_UUID + "--Last Level2"

  //check if it exists
```

```
 b, _ := s.assetExists(ctx, LastL2_clin)
 if b {
  //read block address at LastL2_clin
  address, err := ctx.GetStub().GetState(LastL2_clin)
  if err != nil {
   return nil, fmt.Errorf("failed to read from world state: %v", err)
  }
  var address_str string
  json.Unmarshal(address, &address_str)
  return s.readL2Block(ctx, address_str)
 } else {
  return nil, nil
 }
}

// //determines if there is a need to create a new level 2 block. This should be done about every 30 mins for a Level2 summary.
// This would have to be modified for adding more layers to the blockchain.
func (s *SmartContract) summaryNeeded(ctx contractapi.TransactionContextInterface) (bool, error) {

 //get local UUID
 //Get last L2 block for the local clinic
 //get end time - was it more than 30 mins ago?
 //return True

 uuid, err := s.getLocalUUID(ctx)
 search := uuid + "--Last Level2"
 if err != nil {
  return false, nil //fmt.Errorf("no UUID found")
 }
 exists, _ := s.assetExists(ctx, search)
 if exists {
  last_L2, _ := s.getLastL2Block(ctx, uuid)
  L2_time, err := parseTime(last_L2.StartTime)

  if err != nil {
   return false, fmt.Errorf("error2") //err
  }

  location, _ := time.LoadLocation("America/Los_Angeles")

  timeInUTC := time.Now()
  now := timeInUTC.In(location)
  new_L2 := L2_time.Add(time.Minute * 30)

  if new_L2.Before(now) {
   return true, nil
  } else {
   return false, nil
  }

 } else {
  return true, nil
 }
}

// creates a Level2 summary block.
func (s *SmartContract) createLevel2(ctx contractapi.TransactionContextInterface, clinic_UUID string, L1_list []*Level1) error {
 //This function assumes that the array of blocks passed to it are sorted by the end date from oldest to newest.
 //set up and data values to hold the information for the block
 end_date, _ := parseTime("1970-01-01 00:00:01")
 start_date := time.Now()
 var BlockIDHash = make(map[string]string)
 var patientLogHashes = make(map[string]map[string]string) //map[PID]map[LogID]Hash
 var merkleRoots = make(map[string]string)                 //map patient id to their merkle root for the blocks
 var patientLogIDs = make(map[string][]string)
 //get information
 for _, block := range L1_list {
  //get and update times if needed
  b_e_t, _ := parseTime(block.EndTime)
  b_s_t, _ := parseTime(block.StartTime)
  if b_e_t.After(end_date) {
```

```
 end_date = b_e_t
 }
 if b_s_t.Before(start_date) {
  start_date = b_s_t
 }

 //get patient data ready for processing - get it from all the included blocks.
 for PatientID := range block.Data {
  if _, err := patientLogHashes[PatientID]; !err {
   patientLogHashes[PatientID] = make(map[string]string)
  }
  for log, hash := range block.Data[PatientID] {
   patientLogHashes[PatientID][log] = hash
   patientLogIDs[PatientID] = append(patientLogIDs[PatientID], log)
  }

 }
 //get the block's ID --> want to be able to store a hash of the block's data.
 block_ID := createID(block.StartTime, block.EndTime, block.ClinicID, block.Level)
 //Get hash of block JSON data and save it to the hash structure
 L1JSON, _ := json.Marshal(block)
 hash := sha256.New()
 hash.Write(L1JSON)
 BlockIDHash[block_ID] = hex.EncodeToString(hash.Sum(nil))

 //get the hash data from the block and make a merkle root for each patient in the block.
 //store this root for later ueses.
}

//Create ID and check it wasn't used already

//create merkle trees
for PatientID := range patientLogHashes {
 //sort data by log ID
 keys := make([]string, 0, len(patientLogHashes[PatientID]))
 for k := range patientLogHashes[PatientID] {
  keys = append(keys, k)
 }
 sort.Strings(keys)
 //create merkle tree
 var data [][]byte
 for _, k := range keys {
  str := patientLogHashes[PatientID][k]
  //fmt.Printf("%x\n", []byte(str))
  data = append(data, []byte(str))
 }
 root, _ := merkle.ProofsFromByteSlices(data)
 merkleRoots[PatientID] = hex.EncodeToString(root)
}
//convert time object to strings
startT := start_date.Format("2006-01-02 15:04:05.999999")
endT := end_date.Format("2006-01-02 15:04:05.999999")
//create ID for block
l2id := createID(startT, endT, clinic_UUID, "Level2")
//check to see if the ID already exists
exists, err := s.assetExists(ctx, l2id)
if err != nil {
 return err
}
if exists {
 return fmt.Errorf("the Block %s already exists. Is this a duplicate?", l2id)
}

//Create Block
l2 := Level2{
 Level:           "Level2",
 ClinicID:        clinic_UUID,
 StartTime:       startT,
 EndTime:         endT,
 Merkle_Data:     merkleRoots,
 HashedBlock:     BlockIDHash,
```

```
  PatientToLogIDs: patientLogIDs}

 //Store block
 L2JSON, err := json.Marshal(l2)
 if err != nil {
  return err
 }
 err = ctx.GetStub().PutState(l2id, L2JSON)

 if err != nil {
  return err
 }

 s.updateLastL2(ctx, l2id, clinic_UUID)

 //createLevel2 should only run when the a log is uploaded from the local chain. Thus make sure when it runs we update the frontier set
↪  to indicate that the blockchain has a new log for the local UUID.
 location, _ := time.LoadLocation("America/Los_Angeles")
 timeInUTC := time.Now()
 now := timeInUTC.In(location)
 FS, _ := s.GetFrontierSet(ctx)
 N := FS.Nodes[clinic_UUID]
 N.LastMessageID = l2id
 N.LastConnection = now.Format("2006-01-02 15:04:05")
 FS.Nodes[clinic_UUID] = N

 s.writeFrontierSet(ctx, FS)

 return ctx.GetStub().SetEvent("Level2 Created", L2JSON)
}

// // finds and returns a level 2 block.
func (s *SmartContract) readL2Block(ctx contractapi.TransactionContextInterface, id string) (*Level2, error) {
 assetJSON, err := ctx.GetStub().GetState(id)
 if err != nil {
  return nil, fmt.Errorf("failed to read from world state: %v", err)
 }
 if assetJSON == nil {
  return nil, fmt.Errorf("the asset %s does not exist", id)
 }

 var asset Level2
 err = json.Unmarshal(assetJSON, &asset)
 if err != nil {
  return nil, err
 }

 return &asset, nil
}

// This was a function that was used in the testing of the ABLOC system it allowed the bypassing of the summarizing process to allow the
↪  scalability tests to run and to test other level2 specific functions.
// This should be converted to be a private function but it is left as public to support the ABLOC scalability testing.
func (s *SmartContract) CreateL2fromParts(ctx contractapi.TransactionContextInterface, clinID string, start string, end string, data
↪  []map[string]string, hashedBlocks []map[string]string, PatientLogIDs []map[string][]string) error {
 l2B := Level2{
  Level:           "Level2",
  ClinicID:        clinID,
  StartTime:       start,
  EndTime:         end,
  Merkle_Data:     data[0],
  HashedBlock:     hashedBlocks[0],
  PatientToLogIDs: PatientLogIDs[0],
 }

 l2id := createID(start, end, clinID, "Level2")

 exists, err := s.assetExists(ctx, l2id)
 if err != nil {
  return err
 }
```

```
 if exists {
  return fmt.Errorf("the Block %s already exists. Is this a duplicate?", l2id)
 }

 L2JSON, err := json.Marshal(l2B)
 if err != nil {
  return err
 }
 err = ctx.GetStub().PutState(l2id, L2JSON)

 if err != nil {
  return err
 }

 s.updateLastL2(ctx, l2id, clinID)

 location, _ := time.LoadLocation("America/Los_Angeles")
 timeInUTC := time.Now()
 now := timeInUTC.In(location)
 FS, _ := s.GetFrontierSet(ctx)
 N := FS.Nodes[clinID]
 N.LastMessageID = l2id
 N.LastConnection = now.Format("2006-01-02 15:04:05")
 FS.Nodes[clinID] = N

 s.writeFrontierSet(ctx, FS)

 return ctx.GetStub().SetEvent("Level2 Created", L2JSON)
}

func (s *SmartContract) updateLastL2(ctx contractapi.TransactionContextInterface, l2id string, clinID string) error {

 L2IDJson, _ := json.Marshal(l2id)
 LastL2_clin := clinID + "--Last Level2"
 err := ctx.GetStub().PutState(LastL2_clin, L2IDJson)
 if err != nil {
  return err
 }
 return nil
}

// ****************************** GOSSIP SPECIFIC Functions ****************************** \\
type NetworkNode struct {
 NodeID         string `json:"NodeID"`
 NodeURL        string `json:"NodeURL"`
 LastConnection string `json:"LastConnection"`
 LastMessageID  string `json:"LastMessageID"`
}

type FrontierSet struct {
 Nodes map[string]NetworkNode `json:"FrontierSet"`
}

// Get and return the FrontierSet from the blockchain
func (s *SmartContract) GetFrontierSet(ctx contractapi.TransactionContextInterface) (*FrontierSet, error) {
 FSID := "FrontierSet"

 assetJSON, err := ctx.GetStub().GetState(FSID)
 if err != nil {
  return nil, fmt.Errorf("failed to read from world state: %v", err)
 }
 if assetJSON == nil {
  return nil, fmt.Errorf("the asset %s does not exist", FSID)
 }

 var asset FrontierSet
 err = json.Unmarshal(assetJSON, &asset)
 if err != nil {
  return nil, err
 }
```

```
 return &asset, nil
}

// This function takes the information about a new node that can be gossiped with and stores it on the blockchain. This function will be
↪  used to create a node for every clinic on the chain (including the local one) and will create the Frontier Set if it doesn't exist.
func (s *SmartContract) AddNodeToFrontierSet(ctx contractapi.TransactionContextInterface, NodeID string, NodeURL string) error {
 exist, _ := s.assetExists(ctx, "FrontierSet")
 var FS FrontierSet
 if exist {
  temp, err := s.GetFrontierSet(ctx)
  if err != nil {
   return err
  }
  FS = *temp
 } else {
  FS = FrontierSet{}
 }

 if FS.Nodes == nil {
  FS.Nodes = make(map[string]NetworkNode)
 }
 _, ok := FS.Nodes[NodeID]
 if !ok {
  FS.Nodes[NodeID] = NetworkNode{NodeID: NodeID, NodeURL: NodeURL, LastConnection: "", LastMessageID: ""}
 } else {
  return fmt.Errorf("node ID already exists")
 }

 //update FS
 err := s.writeFrontierSet(ctx, &FS)

 if err != nil {
  return err
 }

 return nil
}

// Writes changes to the frontier set to the blockchain.
func (s *SmartContract) writeFrontierSet(ctx contractapi.TransactionContextInterface, FS *FrontierSet) error {

 FS_JSON, _ := json.Marshal(FS)
 err := ctx.GetStub().PutState("FrontierSet", FS_JSON)
 if err != nil {
  return err
 }
 return nil
}

// This function is what processes and saves the Level2 gossip from a node to the blockchain. This would need to be modified to support
↪  multiple levels.
func (s *SmartContract) GossipFromNode(ctx contractapi.TransactionContextInterface, clinID string, contact_time string, start string,
↪  end string, data []map[string]string, hashedBlocks []map[string]string, PatientLogIDs []map[string][]string) error {
 err := s.CreateL2fromParts(ctx, clinID, start, end, data, hashedBlocks, PatientLogIDs)

 if err != nil {
  return err
 }

 FS, err := s.GetFrontierSet(ctx)

 if err != nil {
  return err
 }

 n := FS.Nodes[clinID]
 n.LastConnection = contact_time
 n.LastMessageID = createID(start, end, clinID, "Level2")
 FS.Nodes[clinID] = n

 err = s.writeFrontierSet(ctx, FS)
```

```
 if err != nil {
  return err
 }

 return nil

}
```

A.8 Firefly Interface

This is the interface that is required by Firefly to use the smart contract with an automatically created REST API. The Firefly interface is used to allow Firefly to interface with the smart contract, create a REST API, and to provide a Swagger UI documentation of the API.


```
{
    "namespace": "default",
    "name": "BLOC",
    "description": "Allows the EMR to upload logs to the blockchain, acess the blockchain and support the gossip protocol.",
    "version": "1.0",
    "methods":[
        {
            "name": "GetFrontierSet",
            "pathname": "",
            "description": "",
            "params": [],
            "returns": [
                {
                    "name": "",
                    "schema": {
                      "type": "array",
                      "details": {
                        "type": "object",
                        "properties": {
                          "type": "string"
                        }
                      }
                    }
                }
            ]
        },
        {
            "name": "AddNodeToFrontierSet",
            "pathname": "",
            "description": "",
            "params": [
                {
                    "name": "NodeID",
                    "schema":{
                        "type":"string"
                    }
                },
                {
                    "name":"NodeURL",
                    "schema":{
                        "type":"string"
                    }
                }
            ],
            "returns":[{
                "name": "",
                "schema": {
                  "type": "string"
                }
            }]
        },
        {
            "name": "GetAllLevel1",
            "pathname": "",
            "description": "",
            "params": [],
            "returns": [
                {
                    "name": "",
                    "schema": {
```

```
              "type": "array",
              "details": {
                "type": "object",
                "properties": {
                  "type": "string"
                }
              }
            }
        }
    ]
},
{
    "name": "CreateLevel1",
    "pathname": "",
    "description": "",
    "params": [
        {
            "name": "start_date",
            "schema":{
                "type":"string"
            }
        },
        {
            "name":"end_date",
            "schema":{
                "type":"string"
            }
        },
        {
            "name":"clinic_UUID",
            "schema":{
                "type":"string"
            }
        },
        {
            "name":"log_data_in",
            "schema":{
                "type":"array"
            }
        }

    ],
    "returns":[{
        "name": "",
        "schema": {
          "type": "string"
        }
    }]
},
{
    "name": "GetLevel1Block",
    "pathname": "",
    "description": "",
    "params": [
        {
            "name": "start_date",
            "schema":{
                "type":"string"
            }
        },
        {
            "name":"end_date",
            "schema":{
                "type":"string"
            }
        },
        {
            "name":"clinic_UUID",
            "schema":{
                "type":"string"
            }
```

```
        }
    ],
    "returns": [
        {
            "name": "",
            "schema": {
              "type": "array",
              "details": {
                "type": "object",
                "properties": {
                  "type": "string"
                }
              }
            }
        }
    ]
},
{
    "name": "GetLogHashWithLogID",
    "pathname": "",
    "description": "",
    "params": [
        {
            "name": "logID",
            "schema":{
                "type":"string"
            }
        }
    ],
    "returns": [
        {
            "name": "",
            "schema": {
              "type": "string"
            }
        }
    ]
},
{
    "name": "GetLevel2Block",
    "pathname": "",
    "description": "",
    "params": [
        {
            "name": "start_date",
            "schema":{
                "type":"string"
            }
        },
        {
            "name":"end_date",
            "schema":{
                "type":"string"
            }
        },
        {
            "name":"clinic_UUID",
            "schema":{
                "type":"string"
            }
        }
    ],
    "returns": [
        {
            "name": "",
            "schema": {
              "type": "array",
              "details": {
                "type": "object",
                "properties": {
                  "type": "string"
```

```
                }
              }
            }
        }
    ]
},
{
    "name": "GetAllLevel2",
    "pathname": "",
    "description": "",
    "params": [],
    "returns": [
        {
            "name": "",
            "schema": {
              "type": "array",
              "details": {
                "type": "object",
                "properties": {
                  "type": "string"
                }
              }
            }
        }
    ]
},
{
    "name": "GetClinicLevel2Blocks",
    "pathname": "",
    "description": "",
    "params": [
        {
            "name":"UUID",
            "schema":{
                "type":"string"
            }
        }
    ],
    "returns": [
        {
            "name": "",
            "schema": {
              "type": "array",
              "details": {
                "type": "object",
                "properties": {
                  "type": "string"
                }
              }
            }
        }
    ]
},
{
    "name": "GetClinicLevel2BlocksAfterTime",
    "pathname": "",
    "description": "",
    "params": [
        {
            "name":"endTime",
            "schema":{
                "type":"string"
            }
        },
        {
            "name":"clinic_UUID",
            "schema":{
                "type":"string"
            }
        }
    ],
```

```
    "returns": [
        {
            "name": "",
            "schema": {
              "type": "array",
              "details": {
                "type": "object",
                "properties": {
                  "type": "string"
                }
              }
            }
        }
    ]
},
{
    "name": "CreateL2fromParts",
    "pathname": "",
    "description": "",
    "params": [
        {
            "name":"clinID",
            "schema":{
                "type":"string"
            }
        },
        {
            "name": "start",
            "schema":{
                "type":"string"
            }
        },
        {
            "name":"end",
            "schema":{
                "type":"string"
            }
        },
        {
            "name":"data",
            "schema":{
                "type":"array"
            }
        },
        {
            "name":"hashedBlocks",
            "schema":{
                "type":"array"
            }
        },
        {
            "name":"PatientLogIDs",
            "schema":{
                "type":"array"
            }
        }
    ],
    "returns":[{
        "name": "",
        "schema": {
          "type": "string"
        }
    }]
},
{
    "name": "GossipFromNode",
    "pathname": "",
    "description": "",
    "params": [
        {
            "name":"clinID",
```

```
                    "schema":{
                        "type":"string"
                    }
                },
                {
                    "name":"contact_time",
                    "schema":{
                        "type":"string"
                    }
                },
                {
                    "name": "start",
                    "schema":{
                        "type":"string"
                    }
                },
                {
                    "name":"end",
                    "schema":{
                        "type":"string"
                    }
                },
                {
                    "name":"data",
                    "schema":{
                        "type":"array"
                    }
                },
                {
                    "name":"hashedBlocks",
                    "schema":{
                        "type":"array"
                    }
                },
                {
                    "name":"PatientLogIDs",
                    "schema":{
                        "type":"array"
                    }
                }
            ],
            "returns":[{
                "name": "",
                "schema": {
                  "type": "string"
                }
            }]
        }
    ]
}
```

A.9 Prototype Swagger UI

The Swagger UI is generated by Firefly, which processes the smart contract described in A.7, applies the Firefly Interface described in A.8, and finally generates an API with the following Swagger UI documentation.

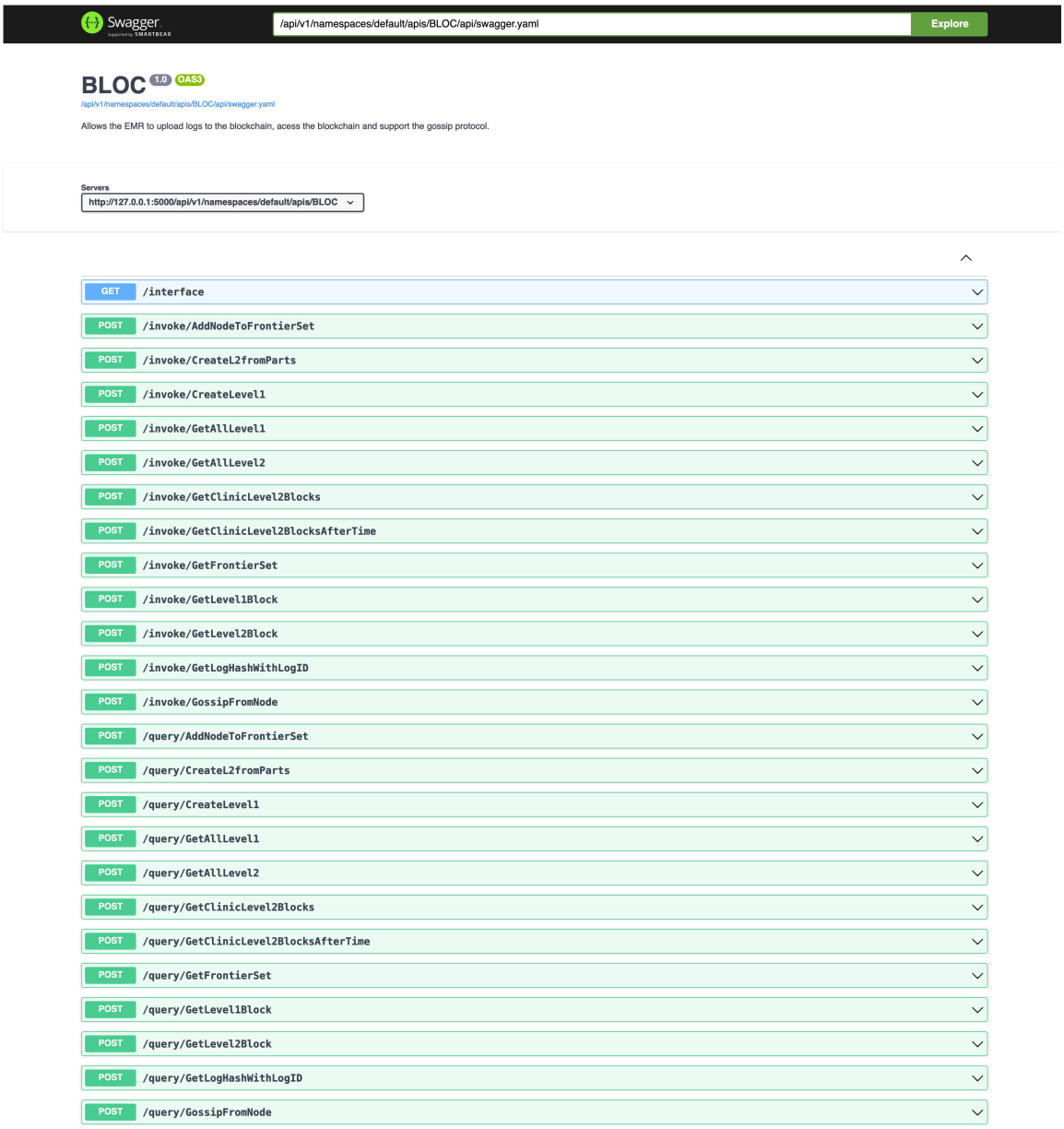

Figure A.6: Screenshot of ABLOC's SwaggerUI

```
{
    "components": {},
    "info": {
      "description": "Allows the EMR to upload logs to the blockchain, acess the blockchain and support the gossip protocol.",
      "title": "BLOC",
      "version": "1.0"
    },
    "openapi": "3.0.2",
    "paths": {
      "/interface": {
        "get": {
          "operationId": "interface",
          "parameters": [
            {
              "description": "Server-side request timeout (milliseconds, or set a custom suffix like 10s)",
              "in": "header",
              "name": "Request-Timeout",
              "schema": {
                "default": "2m0s",
                "type": "string"
              }
            }
          ],
          "responses": {
            "200": {
              "content": {
                "application/json": {
                  "schema": {
                    "properties": {
                      "description": {
                        "description": "A description of the smart contract this FFI represents",
                        "type": "string"
                      },
                      "errors": {
                        "description": "An array of smart contract error definitions",
                        "items": {
                          "description": "An array of smart contract error definitions",
                          "properties": {
                            "description": {
                              "description": "A description of the smart contract error",
                              "type": "string"
                            },
                            "id": {
                              "description": "The UUID of the FFI error definition",
                              "format": "uuid",
                              "type": "string"
                            },
                            "interface": {
                              "description": "The UUID of the FFI smart contract definition that this error is part of",
                              "format": "uuid",
                              "type": "string"
                            },
                            "name": {
                              "description": "The name of the error",
                              "type": "string"
                            },
                            "namespace": {
                              "description": "The namespace of the FFI",
                              "type": "string"
                            },
                            "params": {
                              "description": "An array of error parameter/argument definitions",
                              "items": {
                                "description": "An array of error parameter/argument definitions",
                                "properties": {
                                  "name": {
                                    "description": "The name of the parameter. Note that parameters must be ordered correctly on the
↪  FFI, according to the order in the blockchain smart contract",
                                    "type": "string"
                                  },
                                  "schema": {
```

```
                                "description": "FireFly uses an extended subset of JSON Schema to describe parameters, similar to
↪  OpenAPI/Swagger. Converters are available for native blockchain interface definitions / type systems - such as an Ethereum ABI. See
↪  the documentation for more detail"
                              }
                            },
                            "type": "object"
                          },
                          "type": "array"
                        },
                        "pathname": {
                          "description": "The unique name allocated to this error within the FFI for use on URL paths",
                          "type": "string"
                        },
                        "signature": {
                          "description": "The stringified signature of the error, as computed by the blockchain plugin",
                          "type": "string"
                        }
                      },
                      "type": "object"
                    },
                    "type": "array"
                  },
                  "events": {
                    "description": "An array of smart contract event definitions",
                    "items": {
                      "description": "An array of smart contract event definitions",
                      "properties": {
                        "description": {
                          "description": "A description of the smart contract event",
                          "type": "string"
                        },
                        "details": {
                          "additionalProperties": {
                            "description": "Additional blockchain specific fields about this event from the original smart contract.
↪  Used by the blockchain plugin and for documentation generation."
                          },
                          "description": "Additional blockchain specific fields about this event from the original smart contract.
↪  Used by the blockchain plugin and for documentation generation.",
                          "type": "object"
                        },
                        "id": {
                          "description": "The UUID of the FFI event definition",
                          "format": "uuid",
                          "type": "string"
                        },
                        "interface": {
                          "description": "The UUID of the FFI smart contract definition that this event is part of",
                          "format": "uuid",
                          "type": "string"
                        },
                        "name": {
                          "description": "The name of the event",
                          "type": "string"
                        },
                        "namespace": {
                          "description": "The namespace of the FFI",
                          "type": "string"
                        },
                        "params": {
                          "description": "An array of event parameter/argument definitions",
                          "items": {
                            "description": "An array of event parameter/argument definitions",
                            "properties": {
                              "name": {
                                "description": "The name of the parameter. Note that parameters must be ordered correctly on the
↪  FFI, according to the order in the blockchain smart contract",
                                "type": "string"
                              },
                              "schema": {
                                "description": "FireFly uses an extended subset of JSON Schema to describe parameters, similar to
↪  OpenAPI/Swagger. Converters are available for native blockchain interface definitions / type systems - such as an Ethereum ABI. See
↪  the documentation for more detail"
```

```
                                }
                              },
                              "type": "object"
                            },
                            "type": "array"
                          },
                          "pathname": {
                            "description": "The unique name allocated to this event within the FFI for use on URL paths. Supports
↪  contracts that have multiple event overrides with the same name",
                            "type": "string"
                          },
                          "signature": {
                            "description": "The stringified signature of the event, as computed by the blockchain plugin",
                            "type": "string"
                          }
                        },
                        "type": "object"
                      },
                      "type": "array"
                    },
                    "id": {
                      "description": "The UUID of the FireFly interface (FFI) smart contract definition",
                      "format": "uuid",
                      "type": "string"
                    },
                    "message": {
                      "description": "The UUID of the broadcast message that was used to publish this FFI to the network",
                      "format": "uuid",
                      "type": "string"
                    },
                    "methods": {
                      "description": "An array of smart contract method definitions",
                      "items": {
                        "description": "An array of smart contract method definitions",
                        "properties": {
                          "description": {
                            "description": "A description of the smart contract method",
                            "type": "string"
                          },
                          "details": {
                            "additionalProperties": {
                              "description": "Additional blockchain specific fields about this method from the original smart
↪  contract. Used by the blockchain plugin and for documentation generation."
                            },
                            "description": "Additional blockchain specific fields about this method from the original smart contract.
↪  Used by the blockchain plugin and for documentation generation.",
                            "type": "object"
                          },
                          "id": {
                            "description": "The UUID of the FFI method definition",
                            "format": "uuid",
                            "type": "string"
                          },
                          "interface": {
                            "description": "The UUID of the FFI smart contract definition that this method is part of",
                            "format": "uuid",
                            "type": "string"
                          },
                          "name": {
                            "description": "The name of the method",
                            "type": "string"
                          },
                          "namespace": {
                            "description": "The namespace of the FFI",
                            "type": "string"
                          },
                          "params": {
                            "description": "An array of method parameter/argument definitions",
                            "items": {
                              "description": "An array of method parameter/argument definitions",
                              "properties": {
```

```
                                "name": {
                                  "description": "The name of the parameter. Note that parameters must be ordered correctly on the
↪  FFI, according to the order in the blockchain smart contract",
                                  "type": "string"
                                },
                                "schema": {
                                  "description": "FireFly uses an extended subset of JSON Schema to describe parameters, similar to
↪  OpenAPI/Swagger. Converters are available for native blockchain interface definitions / type systems - such as an Ethereum ABI. See
↪  the documentation for more detail"
                                }
                              },
                              "type": "object"
                            },
                            "type": "array"
                          },
                          "pathname": {
                            "description": "The unique name allocated to this method within the FFI for use on URL paths. Supports
↪  contracts that have multiple method overrides with the same name",
                            "type": "string"
                          },
                          "returns": {
                            "description": "An array of method return definitions",
                            "items": {
                              "description": "An array of method return definitions",
                              "properties": {
                                "name": {
                                  "description": "The name of the parameter. Note that parameters must be ordered correctly on the
↪  FFI, according to the order in the blockchain smart contract",
                                  "type": "string"
                                },
                                "schema": {
                                  "description": "FireFly uses an extended subset of JSON Schema to describe parameters, similar to
↪  OpenAPI/Swagger. Converters are available for native blockchain interface definitions / type systems - such as an Ethereum ABI. See
↪  the documentation for more detail"
                                }
                              },
                              "type": "object"
                            },
                            "type": "array"
                          }
                        },
                        "type": "object"
                      },
                      "type": "array"
                    },
                    "name": {
                      "description": "The name of the FFI - usually matching the smart contract name",
                      "type": "string"
                    },
                    "namespace": {
                      "description": "The namespace of the FFI",
                      "type": "string"
                    },
                    "version": {
                      "description": "A version for the FFI - use of semantic versioning such as 'v1.0.1' is encouraged",
                      "type": "string"
                    }
                  },
                  "type": "object"
                }
              }
            },
            "description": "Success"
          },
          "default": {
            "description": ""
          }
        },
        "tags": [
          ""
        ]
```

```
        }
      },
      "/invoke/AddNodeToFrontierSet": {
        "post": {
          "operationId": "invoke_AddNodeToFrontierSet",
          "parameters": [
            {
              "description": "Server-side request timeout (milliseconds, or set a custom suffix like 10s)",
              "in": "header",
              "name": "Request-Timeout",
              "schema": {
                "default": "2m0s",
                "type": "string"
              }
            }
          ],
          "requestBody": {
            "content": {
              "application/json": {
                "schema": {
                  "properties": {
                    "idempotencyKey": {
                      "description": "An optional identifier to allow idempotent submission of requests. Stored on the transaction
↪  uniquely within a namespace",
                      "type": "string"
                    },
                    "input": {
                      "description": "A map of named inputs. The name and type of each input must be compatible with the FFI description
↪  of the method, so that FireFly knows how to serialize it to the blockchain via the connector",
                      "properties": {
                        "NodeID": {
                          "type": "string"
                        },
                        "NodeURL": {
                          "type": "string"
                        }
                      },
                      "type": "object"
                    },
                    "key": {
                      "description": "The blockchain signing key that will sign the invocation. Defaults to the first signing key of the
↪  organization that operates the node",
                      "type": "string"
                    },
                    "options": {
                      "description": "A map of named inputs that will be passed through to the blockchain connector",
                      "type": "object"
                    }
                  },
                  "type": "object"
                }
              }
            }
          },
          "responses": {
            "200": {
              "content": {
                "application/json": {
                  "schema": {
                    "properties": {
                      "idempotencyKey": {
                        "description": "An optional identifier to allow idempotent submission of requests. Stored on the transaction
↪  uniquely within a namespace",
                        "type": "string"
                      },
                      "input": {
                        "description": "A map of named inputs. The name and type of each input must be compatible with the FFI
↪  description of the method, so that FireFly knows how to serialize it to the blockchain via the connector",
                        "properties": {
                          "": {
                            "type": "string"
```

```
                        }
                      },
                      "type": "object"
                    },
                    "key": {
                      "description": "The blockchain signing key that will sign the invocation. Defaults to the first signing key of
↪  the organization that operates the node",
                      "type": "string"
                    },
                    "options": {
                      "description": "A map of named inputs that will be passed through to the blockchain connector",
                      "type": "object"
                    }
                  },
                  "type": "object"
                }
              }
            },
            "description": "Success"
          },
          "default": {
            "description": ""
          }
        },
        "tags": [
          ""
        ]
      }
    },
    "/invoke/CreateL2fromParts": {
      "post": {
        "operationId": "invoke_CreateL2fromParts",
        "parameters": [
          {
            "description": "Server-side request timeout (milliseconds, or set a custom suffix like 10s)",
            "in": "header",
            "name": "Request-Timeout",
            "schema": {
              "default": "2m0s",
              "type": "string"
            }
          }
        ],
        "requestBody": {
          "content": {
            "application/json": {
              "schema": {
                "properties": {
                  "idempotencyKey": {
                    "description": "An optional identifier to allow idempotent submission of requests. Stored on the transaction
↪  uniquely within a namespace",
                    "type": "string"
                  },
                  "input": {
                    "description": "A map of named inputs. The name and type of each input must be compatible with the FFI description
↪  of the method, so that FireFly knows how to serialize it to the blockchain via the connector",
                    "properties": {
                      "PatientLogIDs": {
                        "type": "array"
                      },
                      "clinID": {
                        "type": "string"
                      },
                      "data": {
                        "type": "array"
                      },
                      "end": {
                        "type": "string"
                      },
                      "hashedBlocks": {
                        "type": "array"
```

```
                    },
                    "start": {
                      "type": "string"
                    }
                  },
                  "type": "object"
                },
                "key": {
                  "description": "The blockchain signing key that will sign the invocation. Defaults to the first signing key of the
↪  organization that operates the node",
                  "type": "string"
                },
                "options": {
                  "description": "A map of named inputs that will be passed through to the blockchain connector",
                  "type": "object"
                }
              },
              "type": "object"
            }
          }
        }
      },
      "responses": {
        "200": {
          "content": {
            "application/json": {
              "schema": {
                "properties": {
                  "idempotencyKey": {
                    "description": "An optional identifier to allow idempotent submission of requests. Stored on the transaction
↪  uniquely within a namespace",
                    "type": "string"
                  },
                  "input": {
                    "description": "A map of named inputs. The name and type of each input must be compatible with the FFI
↪  description of the method, so that FireFly knows how to serialize it to the blockchain via the connector",
                    "properties": {
                      "": {
                        "type": "string"
                      }
                    },
                    "type": "object"
                  },
                  "key": {
                    "description": "The blockchain signing key that will sign the invocation. Defaults to the first signing key of
↪  the organization that operates the node",
                    "type": "string"
                  },
                  "options": {
                    "description": "A map of named inputs that will be passed through to the blockchain connector",
                    "type": "object"
                  }
                },
                "type": "object"
              }
            }
          },
          "description": "Success"
        },
        "default": {
          "description": ""
        }
      },
      "tags": [
        ""
      ]
    }
  },
  "/invoke/CreateLevel1": {
    "post": {
      "operationId": "invoke_CreateLevel1",
```

```
        "parameters": [
          {
            "description": "Server-side request timeout (milliseconds, or set a custom suffix like 10s)",
            "in": "header",
            "name": "Request-Timeout",
            "schema": {
              "default": "2m0s",
              "type": "string"
            }
          }
        ],
        "requestBody": {
          "content": {
            "application/json": {
              "schema": {
                "properties": {
                  "idempotencyKey": {
                    "description": "An optional identifier to allow idempotent submission of requests. Stored on the transaction
↪  uniquely within a namespace",
                    "type": "string"
                  },
                  "input": {
                    "description": "A map of named inputs. The name and type of each input must be compatible with the FFI description
↪  of the method, so that FireFly knows how to serialize it to the blockchain via the connector",
                    "properties": {
                      "clinic_UUID": {
                        "type": "string"
                      },
                      "end_date": {
                        "type": "string"
                      },
                      "log_data_in": {
                        "type": "array"
                      },
                      "start_date": {
                        "type": "string"
                      }
                    },
                    "type": "object"
                  },
                  "key": {
                    "description": "The blockchain signing key that will sign the invocation. Defaults to the first signing key of the
↪  organization that operates the node",
                    "type": "string"
                  },
                  "options": {
                    "description": "A map of named inputs that will be passed through to the blockchain connector",
                    "type": "object"
                  }
                },
                "type": "object"
              }
            }
          }
        },
        "responses": {
          "200": {
            "content": {
              "application/json": {
                "schema": {
                  "properties": {
                    "idempotencyKey": {
                      "description": "An optional identifier to allow idempotent submission of requests. Stored on the transaction
↪  uniquely within a namespace",
                      "type": "string"
                    },
                    "input": {
                      "description": "A map of named inputs. The name and type of each input must be compatible with the FFI
↪  description of the method, so that FireFly knows how to serialize it to the blockchain via the connector",
                      "properties": {
                        "": {
```

```
                        "type": "string"
                      }
                    },
                    "type": "object"
                  },
                  "key": {
                    "description": "The blockchain signing key that will sign the invocation. Defaults to the first signing key of
↪  the organization that operates the node",
                    "type": "string"
                  },
                  "options": {
                    "description": "A map of named inputs that will be passed through to the blockchain connector",
                    "type": "object"
                  }
                },
                "type": "object"
              }
            }
          },
          "description": "Success"
        },
        "default": {
          "description": ""
        }
      },
      "tags": [
        ""
      ]
    }
  },
  "/invoke/GetAllLevel1": {
    "post": {
      "operationId": "invoke_GetAllLevel1",
      "parameters": [
        {
          "description": "Server-side request timeout (milliseconds, or set a custom suffix like 10s)",
          "in": "header",
          "name": "Request-Timeout",
          "schema": {
            "default": "2m0s",
            "type": "string"
          }
        }
      ],
      "requestBody": {
        "content": {
          "application/json": {
            "schema": {
              "properties": {
                "idempotencyKey": {
                  "description": "An optional identifier to allow idempotent submission of requests. Stored on the transaction
↪  uniquely within a namespace",
                  "type": "string"
                },
                "input": {
                  "description": "A map of named inputs. The name and type of each input must be compatible with the FFI description
↪  of the method, so that FireFly knows how to serialize it to the blockchain via the connector",
                  "type": "object"
                },
                "key": {
                  "description": "The blockchain signing key that will sign the invocation. Defaults to the first signing key of the
↪  organization that operates the node",
                  "type": "string"
                },
                "options": {
                  "description": "A map of named inputs that will be passed through to the blockchain connector",
                  "type": "object"
                }
              },
              "type": "object"
            }
```

```
            }
          }
        },
        "responses": {
          "200": {
            "content": {
              "application/json": {
                "schema": {
                  "properties": {
                    "idempotencyKey": {
                      "description": "An optional identifier to allow idempotent submission of requests. Stored on the transaction
↪  uniquely within a namespace",
                      "type": "string"
                    },
                    "input": {
                      "description": "A map of named inputs. The name and type of each input must be compatible with the FFI
↪  description of the method, so that FireFly knows how to serialize it to the blockchain via the connector",
                      "properties": {
                        "": {
                          "details": {
                            "type": "object",
                            "properties": {
                              "type": "string"
                            }
                          },
                          "type": "array"
                        }
                      },
                      "type": "object"
                    },
                    "key": {
                      "description": "The blockchain signing key that will sign the invocation. Defaults to the first signing key of
↪  the organization that operates the node",
                      "type": "string"
                    },
                    "options": {
                      "description": "A map of named inputs that will be passed through to the blockchain connector",
                      "type": "object"
                    }
                  },
                  "type": "object"
                }
              }
            },
            "description": "Success"
          },
          "default": {
            "description": ""
          }
        },
        "tags": [
          ""
        ]
      }
    },
    "/invoke/GetAllLevel2": {
      "post": {
        "operationId": "invoke_GetAllLevel2",
        "parameters": [
          {
            "description": "Server-side request timeout (milliseconds, or set a custom suffix like 10s)",
            "in": "header",
            "name": "Request-Timeout",
            "schema": {
              "default": "2m0s",
              "type": "string"
            }
          }
        ],
        "requestBody": {
          "content": {
```

```
            "application/json": {
              "schema": {
                "properties": {
                  "idempotencyKey": {
                    "description": "An optional identifier to allow idempotent submission of requests. Stored on the transaction
↪  uniquely within a namespace",
                    "type": "string"
                  },
                  "input": {
                    "description": "A map of named inputs. The name and type of each input must be compatible with the FFI description
↪  of the method, so that FireFly knows how to serialize it to the blockchain via the connector",
                    "type": "object"
                  },
                  "key": {
                    "description": "The blockchain signing key that will sign the invocation. Defaults to the first signing key of the
↪  organization that operates the node",
                    "type": "string"
                  },
                  "options": {
                    "description": "A map of named inputs that will be passed through to the blockchain connector",
                    "type": "object"
                  }
                },
                "type": "object"
              }
            }
          }
        },
        "responses": {
          "200": {
            "content": {
              "application/json": {
                "schema": {
                  "properties": {
                    "idempotencyKey": {
                      "description": "An optional identifier to allow idempotent submission of requests. Stored on the transaction
↪  uniquely within a namespace",
                      "type": "string"
                    },
                    "input": {
                      "description": "A map of named inputs. The name and type of each input must be compatible with the FFI
↪  description of the method, so that FireFly knows how to serialize it to the blockchain via the connector",
                      "properties": {
                        "": {
                          "details": {
                            "type": "object",
                            "properties": {
                              "type": "string"
                            }
                          },
                          "type": "array"
                        }
                      },
                      "type": "object"
                    },
                    "key": {
                      "description": "The blockchain signing key that will sign the invocation. Defaults to the first signing key of
↪  the organization that operates the node",
                      "type": "string"
                    },
                    "options": {
                      "description": "A map of named inputs that will be passed through to the blockchain connector",
                      "type": "object"
                    }
                  },
                  "type": "object"
                }
              }
            },
            "description": "Success"
          },
```

```
          "default": {
            "description": ""
          }
        },
        "tags": [
          ""
        ]
      }
    },
    "/invoke/GetClinicLevel2Blocks": {
      "post": {
        "operationId": "invoke_GetClinicLevel2Blocks",
        "parameters": [
          {
            "description": "Server-side request timeout (milliseconds, or set a custom suffix like 10s)",
            "in": "header",
            "name": "Request-Timeout",
            "schema": {
              "default": "2m0s",
              "type": "string"
            }
          }
        ],
        "requestBody": {
          "content": {
            "application/json": {
              "schema": {
                "properties": {
                  "idempotencyKey": {
                    "description": "An optional identifier to allow idempotent submission of requests. Stored on the transaction
↪  uniquely within a namespace",
                    "type": "string"
                  },
                  "input": {
                    "description": "A map of named inputs. The name and type of each input must be compatible with the FFI description
↪  of the method, so that FireFly knows how to serialize it to the blockchain via the connector",
                    "properties": {
                      "UUID": {
                        "type": "string"
                      }
                    },
                    "type": "object"
                  },
                  "key": {
                    "description": "The blockchain signing key that will sign the invocation. Defaults to the first signing key of the
↪  organization that operates the node",
                    "type": "string"
                  },
                  "options": {
                    "description": "A map of named inputs that will be passed through to the blockchain connector",
                    "type": "object"
                  }
                },
                "type": "object"
              }
            }
          }
        },
        "responses": {
          "200": {
            "content": {
              "application/json": {
                "schema": {
                  "properties": {
                    "idempotencyKey": {
                      "description": "An optional identifier to allow idempotent submission of requests. Stored on the transaction
↪  uniquely within a namespace",
                      "type": "string"
                    },
                    "input": {
                      "description": "A map of named inputs. The name and type of each input must be compatible with the FFI
↪  description of the method, so that FireFly knows how to serialize it to the blockchain via the connector",
```

```
                    "properties": {
                      "": {
                        "details": {
                          "type": "object",
                          "properties": {
                            "type": "string"
                          }
                        },
                        "type": "array"
                      }
                    },
                    "type": "object"
                  },
                  "key": {
                    "description": "The blockchain signing key that will sign the invocation. Defaults to the first signing key of
↪  the organization that operates the node",
                    "type": "string"
                  },
                  "options": {
                    "description": "A map of named inputs that will be passed through to the blockchain connector",
                    "type": "object"
                  }
                },
                "type": "object"
              }
            }
          },
          "description": "Success"
        },
        "default": {
          "description": ""
        }
      },
      "tags": [
        ""
      ]
    }
  },
  "/invoke/GetClinicLevel2BlocksAfterTime": {
    "post": {
      "operationId": "invoke_GetClinicLevel2BlocksAfterTime",
      "parameters": [
        {
          "description": "Server-side request timeout (milliseconds, or set a custom suffix like 10s)",
          "in": "header",
          "name": "Request-Timeout",
          "schema": {
            "default": "2m0s",
            "type": "string"
          }
        }
      ],
      "requestBody": {
        "content": {
          "application/json": {
            "schema": {
              "properties": {
                "idempotencyKey": {
                  "description": "An optional identifier to allow idempotent submission of requests. Stored on the transaction
↪  uniquely within a namespace",
                  "type": "string"
                },
                "input": {
                  "description": "A map of named inputs. The name and type of each input must be compatible with the FFI description
↪  of the method, so that FireFly knows how to serialize it to the blockchain via the connector",
                  "properties": {
                    "clinic_UUID": {
                      "type": "string"
                    },
                    "endTime": {
                      "type": "string"
```

```
                    }
                  },
                  "type": "object"
                },
                "key": {
                  "description": "The blockchain signing key that will sign the invocation. Defaults to the first signing key of the
↪  organization that operates the node",
                  "type": "string"
                },
                "options": {
                  "description": "A map of named inputs that will be passed through to the blockchain connector",
                  "type": "object"
                }
              },
              "type": "object"
            }
          }
        }
      },
      "responses": {
        "200": {
          "content": {
            "application/json": {
              "schema": {
                "properties": {
                  "idempotencyKey": {
                    "description": "An optional identifier to allow idempotent submission of requests. Stored on the transaction
↪  uniquely within a namespace",
                    "type": "string"
                  },
                  "input": {
                    "description": "A map of named inputs. The name and type of each input must be compatible with the FFI
↪  description of the method, so that FireFly knows how to serialize it to the blockchain via the connector",
                    "properties": {
                      "": {
                        "details": {
                          "type": "object",
                          "properties": {
                            "type": "string"
                          }
                        },
                        "type": "array"
                      }
                    },
                    "type": "object"
                  },
                  "key": {
                    "description": "The blockchain signing key that will sign the invocation. Defaults to the first signing key of
↪  the organization that operates the node",
                    "type": "string"
                  },
                  "options": {
                    "description": "A map of named inputs that will be passed through to the blockchain connector",
                    "type": "object"
                  }
                },
                "type": "object"
              }
            }
          },
          "description": "Success"
        },
        "default": {
          "description": ""
        }
      },
      "tags": [
        ""
      ]
    }
  },
```

```
    "/invoke/GetFrontierSet": {
      "post": {
        "operationId": "invoke_GetFrontierSet",
        "parameters": [
          {
            "description": "Server-side request timeout (milliseconds, or set a custom suffix like 10s)",
            "in": "header",
            "name": "Request-Timeout",
            "schema": {
              "default": "2m0s",
              "type": "string"
            }
          }
        ],
        "requestBody": {
          "content": {
            "application/json": {
              "schema": {
                "properties": {
                  "idempotencyKey": {
                    "description": "An optional identifier to allow idempotent submission of requests. Stored on the transaction
↪  uniquely within a namespace",
                    "type": "string"
                  },
                  "input": {
                    "description": "A map of named inputs. The name and type of each input must be compatible with the FFI description
↪  of the method, so that FireFly knows how to serialize it to the blockchain via the connector",
                    "type": "object"
                  },
                  "key": {
                    "description": "The blockchain signing key that will sign the invocation. Defaults to the first signing key of the
↪  organization that operates the node",
                    "type": "string"
                  },
                  "options": {
                    "description": "A map of named inputs that will be passed through to the blockchain connector",
                    "type": "object"
                  }
                },
                "type": "object"
              }
            }
          }
        },
        "responses": {
          "200": {
            "content": {
              "application/json": {
                "schema": {
                  "properties": {
                    "idempotencyKey": {
                      "description": "An optional identifier to allow idempotent submission of requests. Stored on the transaction
↪  uniquely within a namespace",
                      "type": "string"
                    },
                    "input": {
                      "description": "A map of named inputs. The name and type of each input must be compatible with the FFI
↪  description of the method, so that FireFly knows how to serialize it to the blockchain via the connector",
                      "properties": {
                        "": {
                          "details": {
                            "type": "object",
                            "properties": {
                              "type": "string"
                            }
                          },
                          "type": "array"
                        }
                      },
                      "type": "object"
                    },
```

```
                    "key": {
                      "description": "The blockchain signing key that will sign the invocation. Defaults to the first signing key of
↪  the organization that operates the node",
                      "type": "string"
                    },
                    "options": {
                      "description": "A map of named inputs that will be passed through to the blockchain connector",
                      "type": "object"
                    }
                  },
                  "type": "object"
                }
              }
            },
            "description": "Success"
          },
          "default": {
            "description": ""
          }
        },
        "tags": [
          ""
        ]
      }
    },
    "/invoke/GetLevel1Block": {
      "post": {
        "operationId": "invoke_GetLevel1Block",
        "parameters": [
          {
            "description": "Server-side request timeout (milliseconds, or set a custom suffix like 10s)",
            "in": "header",
            "name": "Request-Timeout",
            "schema": {
              "default": "2m0s",
              "type": "string"
            }
          }
        ],
        "requestBody": {
          "content": {
            "application/json": {
              "schema": {
                "properties": {
                  "idempotencyKey": {
                    "description": "An optional identifier to allow idempotent submission of requests. Stored on the transaction
↪  uniquely within a namespace",
                    "type": "string"
                  },
                  "input": {
                    "description": "A map of named inputs. The name and type of each input must be compatible with the FFI description
↪  of the method, so that FireFly knows how to serialize it to the blockchain via the connector",
                    "properties": {
                      "clinic_UUID": {
                        "type": "string"
                      },
                      "end_date": {
                        "type": "string"
                      },
                      "start_date": {
                        "type": "string"
                      }
                    },
                    "type": "object"
                  },
                  "key": {
                    "description": "The blockchain signing key that will sign the invocation. Defaults to the first signing key of the
↪  organization that operates the node",
                    "type": "string"
                  },
                  "options": {
```

```
                    "description": "A map of named inputs that will be passed through to the blockchain connector",
                    "type": "object"
                  }
                },
                "type": "object"
              }
            }
          }
        },
        "responses": {
          "200": {
            "content": {
              "application/json": {
                "schema": {
                  "properties": {
                    "idempotencyKey": {
                      "description": "An optional identifier to allow idempotent submission of requests. Stored on the transaction
↪  uniquely within a namespace",
                      "type": "string"
                    },
                    "input": {
                      "description": "A map of named inputs. The name and type of each input must be compatible with the FFI
↪  description of the method, so that FireFly knows how to serialize it to the blockchain via the connector",
                      "properties": {
                        "": {
                          "details": {
                            "type": "object",
                            "properties": {
                              "type": "string"
                            }
                          },
                          "type": "array"
                        }
                      },
                      "type": "object"
                    },
                    "key": {
                      "description": "The blockchain signing key that will sign the invocation. Defaults to the first signing key of
↪  the organization that operates the node",
                      "type": "string"
                    },
                    "options": {
                      "description": "A map of named inputs that will be passed through to the blockchain connector",
                      "type": "object"
                    }
                  },
                  "type": "object"
                }
              }
            },
            "description": "Success"
          },
          "default": {
            "description": ""
          }
        },
        "tags": [
          ""
        ]
      }
    },
    "/invoke/GetLevel2Block": {
      "post": {
        "operationId": "invoke_GetLevel2Block",
        "parameters": [
          {
            "description": "Server-side request timeout (milliseconds, or set a custom suffix like 10s)",
            "in": "header",
            "name": "Request-Timeout",
            "schema": {
              "default": "2m0s",
```

```
              "type": "string"
            }
          }
        ],
        "requestBody": {
          "content": {
            "application/json": {
              "schema": {
                "properties": {
                  "idempotencyKey": {
                    "description": "An optional identifier to allow idempotent submission of requests. Stored on the transaction
↪  uniquely within a namespace",
                    "type": "string"
                  },
                  "input": {
                    "description": "A map of named inputs. The name and type of each input must be compatible with the FFI description
↪  of the method, so that FireFly knows how to serialize it to the blockchain via the connector",
                    "properties": {
                      "clinic_UUID": {
                        "type": "string"
                      },
                      "end_date": {
                        "type": "string"
                      },
                      "start_date": {
                        "type": "string"
                      }
                    },
                    "type": "object"
                  },
                  "key": {
                    "description": "The blockchain signing key that will sign the invocation. Defaults to the first signing key of the
↪  organization that operates the node",
                    "type": "string"
                  },
                  "options": {
                    "description": "A map of named inputs that will be passed through to the blockchain connector",
                    "type": "object"
                  }
                },
                "type": "object"
              }
            }
          }
        },
        "responses": {
          "200": {
            "content": {
              "application/json": {
                "schema": {
                  "properties": {
                    "idempotencyKey": {
                      "description": "An optional identifier to allow idempotent submission of requests. Stored on the transaction
↪  uniquely within a namespace",
                      "type": "string"
                    },
                    "input": {
                      "description": "A map of named inputs. The name and type of each input must be compatible with the FFI
↪  description of the method, so that FireFly knows how to serialize it to the blockchain via the connector",
                      "properties": {
                        "": {
                          "details": {
                            "type": "object",
                            "properties": {
                              "type": "string"
                            }
                          },
                          "type": "array"
                        }
                      },
                      "type": "object"
```

```
                    },
                    "key": {
                      "description": "The blockchain signing key that will sign the invocation. Defaults to the first signing key of
↪  the organization that operates the node",
                      "type": "string"
                    },
                    "options": {
                      "description": "A map of named inputs that will be passed through to the blockchain connector",
                      "type": "object"
                    }
                  },
                  "type": "object"
                }
              }
            },
            "description": "Success"
          },
          "default": {
            "description": ""
          }
        },
        "tags": [
          ""
        ]
      }
    },
    "/invoke/GetLogHashWithLogID": {
      "post": {
        "operationId": "invoke_GetLogHashWithLogID",
        "parameters": [
          {
            "description": "Server-side request timeout (milliseconds, or set a custom suffix like 10s)",
            "in": "header",
            "name": "Request-Timeout",
            "schema": {
              "default": "2m0s",
              "type": "string"
            }
          }
        ],
        "requestBody": {
          "content": {
            "application/json": {
              "schema": {
                "properties": {
                  "idempotencyKey": {
                    "description": "An optional identifier to allow idempotent submission of requests. Stored on the transaction
↪  uniquely within a namespace",
                    "type": "string"
                  },
                  "input": {
                    "description": "A map of named inputs. The name and type of each input must be compatible with the FFI description
↪  of the method, so that FireFly knows how to serialize it to the blockchain via the connector",
                    "properties": {
                      "logID": {
                        "type": "string"
                      }
                    },
                    "type": "object"
                  },
                  "key": {
                    "description": "The blockchain signing key that will sign the invocation. Defaults to the first signing key of the
↪  organization that operates the node",
                    "type": "string"
                  },
                  "options": {
                    "description": "A map of named inputs that will be passed through to the blockchain connector",
                    "type": "object"
                  }
                },
                "type": "object"
```

```
            }
           }
          }
         },
         "responses": {
          "200": {
           "content": {
            "application/json": {
             "schema": {
              "properties": {
               "idempotencyKey": {
                "description": "An optional identifier to allow idempotent submission of requests. Stored on the transaction
↪  uniquely within a namespace",
                "type": "string"
               },
               "input": {
                "description": "A map of named inputs. The name and type of each input must be compatible with the FFI
↪  description of the method, so that FireFly knows how to serialize it to the blockchain via the connector",
                "properties": {
                 "": {
                  "type": "string"
                 }
                },
                "type": "object"
               },
               "key": {
                "description": "The blockchain signing key that will sign the invocation. Defaults to the first signing key of
↪  the organization that operates the node",
                "type": "string"
               },
               "options": {
                "description": "A map of named inputs that will be passed through to the blockchain connector",
                "type": "object"
               }
              },
              "type": "object"
             }
            }
           },
           "description": "Success"
          },
          "default": {
           "description": ""
          }
         },
         "tags": [
          ""
         ]
        }
       },
       "/invoke/GossipFromNode": {
        "post": {
         "operationId": "invoke_GossipFromNode",
         "parameters": [
          {
           "description": "Server-side request timeout (milliseconds, or set a custom suffix like 10s)",
           "in": "header",
           "name": "Request-Timeout",
           "schema": {
            "default": "2m0s",
            "type": "string"
           }
          }
         ],
         "requestBody": {
          "content": {
           "application/json": {
            "schema": {
             "properties": {
              "idempotencyKey": {
               "description": "An optional identifier to allow idempotent submission of requests. Stored on the transaction
↪  uniquely within a namespace",
```

```
                  "type": "string"
                },
                "input": {
                  "description": "A map of named inputs. The name and type of each input must be compatible with the FFI description
↪  of the method, so that FireFly knows how to serialize it to the blockchain via the connector",
                  "properties": {
                    "PatientLogIDs": {
                      "type": "array"
                    },
                    "clinID": {
                      "type": "string"
                    },
                    "contact_time": {
                      "type": "string"
                    },
                    "data": {
                      "type": "array"
                    },
                    "end": {
                      "type": "string"
                    },
                    "hashedBlocks": {
                      "type": "array"
                    },
                    "start": {
                      "type": "string"
                    }
                  },
                  "type": "object"
                },
                "key": {
                  "description": "The blockchain signing key that will sign the invocation. Defaults to the first signing key of the
↪  organization that operates the node",
                  "type": "string"
                },
                "options": {
                  "description": "A map of named inputs that will be passed through to the blockchain connector",
                  "type": "object"
                }
              },
              "type": "object"
            }
          }
        }
      },
      "responses": {
        "200": {
          "content": {
            "application/json": {
              "schema": {
                "properties": {
                  "idempotencyKey": {
                    "description": "An optional identifier to allow idempotent submission of requests. Stored on the transaction
↪  uniquely within a namespace",
                    "type": "string"
                  },
                  "input": {
                    "description": "A map of named inputs. The name and type of each input must be compatible with the FFI
↪  description of the method, so that FireFly knows how to serialize it to the blockchain via the connector",
                    "properties": {
                      "": {
                        "type": "string"
                      }
                    },
                    "type": "object"
                  },
                  "key": {
                    "description": "The blockchain signing key that will sign the invocation. Defaults to the first signing key of
↪  the organization that operates the node",
                    "type": "string"
                  },
```

```
                    "options": {
                      "description": "A map of named inputs that will be passed through to the blockchain connector",
                      "type": "object"
                    }
                  },
                  "type": "object"
                }
              }
            },
            "description": "Success"
          },
          "default": {
            "description": ""
          }
        },
        "tags": [
          ""
        ]
      }
    },
    "/query/AddNodeToFrontierSet": {
      "post": {
        "operationId": "query_AddNodeToFrontierSet",
        "parameters": [
          {
            "description": "Server-side request timeout (milliseconds, or set a custom suffix like 10s)",
            "in": "header",
            "name": "Request-Timeout",
            "schema": {
              "default": "2m0s",
              "type": "string"
            }
          }
        ],
        "requestBody": {
          "content": {
            "application/json": {
              "schema": {
                "properties": {
                  "idempotencyKey": {
                    "description": "An optional identifier to allow idempotent submission of requests. Stored on the transaction
↪  uniquely within a namespace",
                    "type": "string"
                  },
                  "input": {
                    "description": "A map of named inputs. The name and type of each input must be compatible with the FFI description
↪  of the method, so that FireFly knows how to serialize it to the blockchain via the connector",
                    "properties": {
                      "NodeID": {
                        "type": "string"
                      },
                      "NodeURL": {
                        "type": "string"
                      }
                    },
                    "type": "object"
                  },
                  "key": {
                    "description": "The blockchain signing key that will sign the invocation. Defaults to the first signing key of the
↪  organization that operates the node",
                    "type": "string"
                  },
                  "options": {
                    "description": "A map of named inputs that will be passed through to the blockchain connector",
                    "type": "object"
                  }
                },
                "type": "object"
              }
            }
          }
```

```
        },
        "responses": {
          "200": {
            "content": {
              "application/json": {
                "schema": {
                  "properties": {
                    "idempotencyKey": {
                      "description": "An optional identifier to allow idempotent submission of requests. Stored on the transaction
↪  uniquely within a namespace",
                      "type": "string"
                    },
                    "input": {
                      "description": "A map of named inputs. The name and type of each input must be compatible with the FFI
↪  description of the method, so that FireFly knows how to serialize it to the blockchain via the connector",
                      "properties": {
                        "": {
                          "type": "string"
                        }
                      },
                      "type": "object"
                    },
                    "key": {
                      "description": "The blockchain signing key that will sign the invocation. Defaults to the first signing key of
↪  the organization that operates the node",
                      "type": "string"
                    },
                    "options": {
                      "description": "A map of named inputs that will be passed through to the blockchain connector",
                      "type": "object"
                    }
                  },
                  "type": "object"
                }
              }
            },
            "description": "Success"
          },
          "default": {
            "description": ""
          }
        },
        "tags": [
          ""
        ]
      }
    },
    "/query/CreateL2fromParts": {
      "post": {
        "operationId": "query_CreateL2fromParts",
        "parameters": [
          {
            "description": "Server-side request timeout (milliseconds, or set a custom suffix like 10s)",
            "in": "header",
            "name": "Request-Timeout",
            "schema": {
              "default": "2m0s",
              "type": "string"
            }
          }
        ],
        "requestBody": {
          "content": {
            "application/json": {
              "schema": {
                "properties": {
                  "idempotencyKey": {
                    "description": "An optional identifier to allow idempotent submission of requests. Stored on the transaction
↪  uniquely within a namespace",
                    "type": "string"
                  },
```

```
                "input": {
                  "description": "A map of named inputs. The name and type of each input must be compatible with the FFI description
↪  of the method, so that FireFly knows how to serialize it to the blockchain via the connector",
                  "properties": {
                    "PatientLogIDs": {
                      "type": "array"
                    },
                    "clinID": {
                      "type": "string"
                    },
                    "data": {
                      "type": "array"
                    },
                    "end": {
                      "type": "string"
                    },
                    "hashedBlocks": {
                      "type": "array"
                    },
                    "start": {
                      "type": "string"
                    }
                  },
                  "type": "object"
                },
                "key": {
                  "description": "The blockchain signing key that will sign the invocation. Defaults to the first signing key of the
↪  organization that operates the node",
                  "type": "string"
                },
                "options": {
                  "description": "A map of named inputs that will be passed through to the blockchain connector",
                  "type": "object"
                }
              },
              "type": "object"
            }
          }
        }
      },
      "responses": {
        "200": {
          "content": {
            "application/json": {
              "schema": {
                "properties": {
                  "idempotencyKey": {
                    "description": "An optional identifier to allow idempotent submission of requests. Stored on the transaction
↪  uniquely within a namespace",
                    "type": "string"
                  },
                  "input": {
                    "description": "A map of named inputs. The name and type of each input must be compatible with the FFI
↪  description of the method, so that FireFly knows how to serialize it to the blockchain via the connector",
                    "properties": {
                      "": {
                        "type": "string"
                      }
                    },
                    "type": "object"
                  },
                  "key": {
                    "description": "The blockchain signing key that will sign the invocation. Defaults to the first signing key of
↪  the organization that operates the node",
                    "type": "string"
                  },
                  "options": {
                    "description": "A map of named inputs that will be passed through to the blockchain connector",
                    "type": "object"
                  }
                },
```

```
                "type": "object"
              }
            }
          },
          "description": "Success"
        },
        "default": {
          "description": ""
        }
      },
      "tags": [
        ""
      ]
    }
  },
  "/query/CreateLevel1": {
    "post": {
      "operationId": "query_CreateLevel1",
      "parameters": [
        {
          "description": "Server-side request timeout (milliseconds, or set a custom suffix like 10s)",
          "in": "header",
          "name": "Request-Timeout",
          "schema": {
            "default": "2m0s",
            "type": "string"
          }
        }
      ],
      "requestBody": {
        "content": {
          "application/json": {
            "schema": {
              "properties": {
                "idempotencyKey": {
                  "description": "An optional identifier to allow idempotent submission of requests. Stored on the transaction
↪  uniquely within a namespace",
                  "type": "string"
                },
                "input": {
                  "description": "A map of named inputs. The name and type of each input must be compatible with the FFI description
↪  of the method, so that FireFly knows how to serialize it to the blockchain via the connector",
                  "properties": {
                    "clinic_UUID": {
                      "type": "string"
                    },
                    "end_date": {
                      "type": "string"
                    },
                    "log_data_in": {
                      "type": "array"
                    },
                    "start_date": {
                      "type": "string"
                    }
                  },
                  "type": "object"
                },
                "key": {
                  "description": "The blockchain signing key that will sign the invocation. Defaults to the first signing key of the
↪  organization that operates the node",
                  "type": "string"
                },
                "options": {
                  "description": "A map of named inputs that will be passed through to the blockchain connector",
                  "type": "object"
                }
              },
              "type": "object"
            }
          }
```

```
          }
        },
        "responses": {
          "200": {
            "content": {
              "application/json": {
                "schema": {
                  "properties": {
                    "idempotencyKey": {
                      "description": "An optional identifier to allow idempotent submission of requests. Stored on the transaction
↪  uniquely within a namespace",
                      "type": "string"
                    },
                    "input": {
                      "description": "A map of named inputs. The name and type of each input must be compatible with the FFI
↪  description of the method, so that FireFly knows how to serialize it to the blockchain via the connector",
                      "properties": {
                        "": {
                          "type": "string"
                        }
                      },
                      "type": "object"
                    },
                    "key": {
                      "description": "The blockchain signing key that will sign the invocation. Defaults to the first signing key of
↪  the organization that operates the node",
                      "type": "string"
                    },
                    "options": {
                      "description": "A map of named inputs that will be passed through to the blockchain connector",
                      "type": "object"
                    }
                  },
                  "type": "object"
                }
              }
            },
            "description": "Success"
          },
          "default": {
            "description": ""
          }
        },
        "tags": [
          ""
        ]
      }
    },
    "/query/GetAllLevel1": {
      "post": {
        "operationId": "query_GetAllLevel1",
        "parameters": [
          {
            "description": "Server-side request timeout (milliseconds, or set a custom suffix like 10s)",
            "in": "header",
            "name": "Request-Timeout",
            "schema": {
              "default": "2m0s",
              "type": "string"
            }
          }
        ],
        "requestBody": {
          "content": {
            "application/json": {
              "schema": {
                "properties": {
                  "idempotencyKey": {
                    "description": "An optional identifier to allow idempotent submission of requests. Stored on the transaction
↪  uniquely within a namespace",
                    "type": "string"
```

```
                    },
                    "input": {
                      "description": "A map of named inputs. The name and type of each input must be compatible with the FFI description
↪  of the method, so that FireFly knows how to serialize it to the blockchain via the connector",
                      "type": "object"
                    },
                    "key": {
                      "description": "The blockchain signing key that will sign the invocation. Defaults to the first signing key of the
↪  organization that operates the node",
                      "type": "string"
                    },
                    "options": {
                      "description": "A map of named inputs that will be passed through to the blockchain connector",
                      "type": "object"
                    }
                  },
                  "type": "object"
                }
              }
            }
          },
          "responses": {
            "200": {
              "content": {
                "application/json": {
                  "schema": {
                    "properties": {
                      "idempotencyKey": {
                        "description": "An optional identifier to allow idempotent submission of requests. Stored on the transaction
↪  uniquely within a namespace",
                        "type": "string"
                      },
                      "input": {
                        "description": "A map of named inputs. The name and type of each input must be compatible with the FFI
↪  description of the method, so that FireFly knows how to serialize it to the blockchain via the connector",
                        "properties": {
                          "": {
                            "details": {
                              "type": "object",
                              "properties": {
                                "type": "string"
                              }
                            },
                            "type": "array"
                          }
                        },
                        "type": "object"
                      },
                      "key": {
                        "description": "The blockchain signing key that will sign the invocation. Defaults to the first signing key of
↪  the organization that operates the node",
                        "type": "string"
                      },
                      "options": {
                        "description": "A map of named inputs that will be passed through to the blockchain connector",
                        "type": "object"
                      }
                    },
                    "type": "object"
                  }
                }
              },
              "description": "Success"
            },
            "default": {
              "description": ""
            }
          },
          "tags": [
            ""
          ]
```

```
        }
      },
      "/query/GetAllLevel2": {
        "post": {
          "operationId": "query_GetAllLevel2",
          "parameters": [
            {
              "description": "Server-side request timeout (milliseconds, or set a custom suffix like 10s)",
              "in": "header",
              "name": "Request-Timeout",
              "schema": {
                "default": "2m0s",
                "type": "string"
              }
            }
          ],
          "requestBody": {
            "content": {
              "application/json": {
                "schema": {
                  "properties": {
                    "idempotencyKey": {
                      "description": "An optional identifier to allow idempotent submission of requests. Stored on the transaction
↪  uniquely within a namespace",
                      "type": "string"
                    },
                    "input": {
                      "description": "A map of named inputs. The name and type of each input must be compatible with the FFI description
↪  of the method, so that FireFly knows how to serialize it to the blockchain via the connector",
                      "type": "object"
                    },
                    "key": {
                      "description": "The blockchain signing key that will sign the invocation. Defaults to the first signing key of the
↪  organization that operates the node",
                      "type": "string"
                    },
                    "options": {
                      "description": "A map of named inputs that will be passed through to the blockchain connector",
                      "type": "object"
                    }
                  },
                  "type": "object"
                }
              }
            }
          },
          "responses": {
            "200": {
              "content": {
                "application/json": {
                  "schema": {
                    "properties": {
                      "idempotencyKey": {
                        "description": "An optional identifier to allow idempotent submission of requests. Stored on the transaction
↪  uniquely within a namespace",
                        "type": "string"
                      },
                      "input": {
                        "description": "A map of named inputs. The name and type of each input must be compatible with the FFI
↪  description of the method, so that FireFly knows how to serialize it to the blockchain via the connector",
                        "properties": {
                          "": {
                            "details": {
                              "type": "object",
                              "properties": {
                                "type": "string"
                              }
                            },
                            "type": "array"
                          }
                        },
```

```
                    "type": "object"
                  },
                  "key": {
                    "description": "The blockchain signing key that will sign the invocation. Defaults to the first signing key of
↪  the organization that operates the node",
                    "type": "string"
                  },
                  "options": {
                    "description": "A map of named inputs that will be passed through to the blockchain connector",
                    "type": "object"
                  }
                },
                "type": "object"
              }
            }
          },
          "description": "Success"
        },
        "default": {
          "description": ""
        }
      },
      "tags": [
        ""
      ]
    }
  },
  "/query/GetClinicLevel2Blocks": {
    "post": {
      "operationId": "query_GetClinicLevel2Blocks",
      "parameters": [
        {
          "description": "Server-side request timeout (milliseconds, or set a custom suffix like 10s)",
          "in": "header",
          "name": "Request-Timeout",
          "schema": {
            "default": "2m0s",
            "type": "string"
          }
        }
      ],
      "requestBody": {
        "content": {
          "application/json": {
            "schema": {
              "properties": {
                "idempotencyKey": {
                  "description": "An optional identifier to allow idempotent submission of requests. Stored on the transaction
↪  uniquely within a namespace",
                  "type": "string"
                },
                "input": {
                  "description": "A map of named inputs. The name and type of each input must be compatible with the FFI description
↪  of the method, so that FireFly knows how to serialize it to the blockchain via the connector",
                  "properties": {
                    "UUID": {
                      "type": "string"
                    }
                  },
                  "type": "object"
                },
                "key": {
                  "description": "The blockchain signing key that will sign the invocation. Defaults to the first signing key of the
↪  organization that operates the node",
                  "type": "string"
                },
                "options": {
                  "description": "A map of named inputs that will be passed through to the blockchain connector",
                  "type": "object"
                }
              },
```

```
              "type": "object"
            }
          }
        }
      },
      "responses": {
        "200": {
          "content": {
            "application/json": {
              "schema": {
                "properties": {
                  "idempotencyKey": {
                    "description": "An optional identifier to allow idempotent submission of requests. Stored on the transaction
↪  uniquely within a namespace",
                    "type": "string"
                  },
                  "input": {
                    "description": "A map of named inputs. The name and type of each input must be compatible with the FFI
↪  description of the method, so that FireFly knows how to serialize it to the blockchain via the connector",
                    "properties": {
                      "": {
                        "details": {
                          "type": "object",
                          "properties": {
                            "type": "string"
                          }
                        },
                        "type": "array"
                      }
                    },
                    "type": "object"
                  },
                  "key": {
                    "description": "The blockchain signing key that will sign the invocation. Defaults to the first signing key of
↪  the organization that operates the node",
                    "type": "string"
                  },
                  "options": {
                    "description": "A map of named inputs that will be passed through to the blockchain connector",
                    "type": "object"
                  }
                },
                "type": "object"
              }
            }
          },
          "description": "Success"
        },
        "default": {
          "description": ""
        }
      },
      "tags": [
        ""
      ]
    }
  },
  "/query/GetClinicLevel2BlocksAfterTime": {
    "post": {
      "operationId": "query_GetClinicLevel2BlocksAfterTime",
      "parameters": [
        {
          "description": "Server-side request timeout (milliseconds, or set a custom suffix like 10s)",
          "in": "header",
          "name": "Request-Timeout",
          "schema": {
            "default": "2m0s",
            "type": "string"
          }
        }
      ],
```

```
        "requestBody": {
          "content": {
            "application/json": {
              "schema": {
                "properties": {
                  "idempotencyKey": {
                    "description": "An optional identifier to allow idempotent submission of requests. Stored on the transaction
↪  uniquely within a namespace",
                    "type": "string"
                  },
                  "input": {
                    "description": "A map of named inputs. The name and type of each input must be compatible with the FFI description
↪  of the method, so that FireFly knows how to serialize it to the blockchain via the connector",
                    "properties": {
                      "clinic_UUID": {
                        "type": "string"
                      },
                      "endTime": {
                        "type": "string"
                      }
                    },
                    "type": "object"
                  },
                  "key": {
                    "description": "The blockchain signing key that will sign the invocation. Defaults to the first signing key of the
↪  organization that operates the node",
                    "type": "string"
                  },
                  "options": {
                    "description": "A map of named inputs that will be passed through to the blockchain connector",
                    "type": "object"
                  }
                },
                "type": "object"
              }
            }
          }
        },
        "responses": {
          "200": {
            "content": {
              "application/json": {
                "schema": {
                  "properties": {
                    "idempotencyKey": {
                      "description": "An optional identifier to allow idempotent submission of requests. Stored on the transaction
↪  uniquely within a namespace",
                      "type": "string"
                    },
                    "input": {
                      "description": "A map of named inputs. The name and type of each input must be compatible with the FFI
↪  description of the method, so that FireFly knows how to serialize it to the blockchain via the connector",
                      "properties": {
                        "": {
                          "details": {
                            "type": "object",
                            "properties": {
                              "type": "string"
                            }
                          },
                          "type": "array"
                        }
                      },
                      "type": "object"
                    },
                    "key": {
                      "description": "The blockchain signing key that will sign the invocation. Defaults to the first signing key of
↪  the organization that operates the node",
                      "type": "string"
                    },
                    "options": {
```

```
                    "description": "A map of named inputs that will be passed through to the blockchain connector",
                    "type": "object"
                  }
                },
                "type": "object"
              }
            }
          },
          "description": "Success"
        },
        "default": {
          "description": ""
        }
      },
      "tags": [
        ""
      ]
    }
  },
  "/query/GetFrontierSet": {
    "post": {
      "operationId": "query_GetFrontierSet",
      "parameters": [
        {
          "description": "Server-side request timeout (milliseconds, or set a custom suffix like 10s)",
          "in": "header",
          "name": "Request-Timeout",
          "schema": {
            "default": "2m0s",
            "type": "string"
          }
        }
      ],
      "requestBody": {
        "content": {
          "application/json": {
            "schema": {
              "properties": {
                "idempotencyKey": {
                  "description": "An optional identifier to allow idempotent submission of requests. Stored on the transaction
↪  uniquely within a namespace",
                  "type": "string"
                },
                "input": {
                  "description": "A map of named inputs. The name and type of each input must be compatible with the FFI description
↪  of the method, so that FireFly knows how to serialize it to the blockchain via the connector",
                  "type": "object"
                },
                "key": {
                  "description": "The blockchain signing key that will sign the invocation. Defaults to the first signing key of the
↪  organization that operates the node",
                  "type": "string"
                },
                "options": {
                  "description": "A map of named inputs that will be passed through to the blockchain connector",
                  "type": "object"
                }
              },
              "type": "object"
            }
          }
        }
      },
      "responses": {
        "200": {
          "content": {
            "application/json": {
              "schema": {
                "properties": {
                  "idempotencyKey": {
                    "description": "An optional identifier to allow idempotent submission of requests. Stored on the transaction
↪  uniquely within a namespace",
```

```
                    "type": "string"
                  },
                  "input": {
                    "description": "A map of named inputs. The name and type of each input must be compatible with the FFI
↪  description of the method, so that FireFly knows how to serialize it to the blockchain via the connector",
                    "properties": {
                      "": {
                        "details": {
                          "type": "object",
                          "properties": {
                            "type": "string"
                          }
                        },
                        "type": "array"
                      }
                    },
                    "type": "object"
                  },
                  "key": {
                    "description": "The blockchain signing key that will sign the invocation. Defaults to the first signing key of
↪  the organization that operates the node",
                    "type": "string"
                  },
                  "options": {
                    "description": "A map of named inputs that will be passed through to the blockchain connector",
                    "type": "object"
                  }
                },
                "type": "object"
              }
            }
          },
          "description": "Success"
        },
        "default": {
          "description": ""
        }
      },
      "tags": [
        ""
      ]
    }
  },
  "/query/GetLevel1Block": {
    "post": {
      "operationId": "query_GetLevel1Block",
      "parameters": [
        {
          "description": "Server-side request timeout (milliseconds, or set a custom suffix like 10s)",
          "in": "header",
          "name": "Request-Timeout",
          "schema": {
            "default": "2m0s",
            "type": "string"
          }
        }
      ],
      "requestBody": {
        "content": {
          "application/json": {
            "schema": {
              "properties": {
                "idempotencyKey": {
                  "description": "An optional identifier to allow idempotent submission of requests. Stored on the transaction
↪  uniquely within a namespace",
                  "type": "string"
                },
                "input": {
                  "description": "A map of named inputs. The name and type of each input must be compatible with the FFI description
↪  of the method, so that FireFly knows how to serialize it to the blockchain via the connector",
                  "properties": {
```

```
                    "clinic_UUID": {
                      "type": "string"
                    },
                    "end_date": {
                      "type": "string"
                    },
                    "start_date": {
                      "type": "string"
                    }
                  },
                  "type": "object"
                },
                "key": {
                  "description": "The blockchain signing key that will sign the invocation. Defaults to the first signing key of the
↪  organization that operates the node",
                  "type": "string"
                },
                "options": {
                  "description": "A map of named inputs that will be passed through to the blockchain connector",
                  "type": "object"
                }
              },
              "type": "object"
            }
          }
        }
      },
      "responses": {
        "200": {
          "content": {
            "application/json": {
              "schema": {
                "properties": {
                  "idempotencyKey": {
                    "description": "An optional identifier to allow idempotent submission of requests. Stored on the transaction
↪  uniquely within a namespace",
                    "type": "string"
                  },
                  "input": {
                    "description": "A map of named inputs. The name and type of each input must be compatible with the FFI
↪  description of the method, so that FireFly knows how to serialize it to the blockchain via the connector",
                    "properties": {
                      "": {
                        "details": {
                          "type": "object",
                          "properties": {
                            "type": "string"
                          }
                        },
                        "type": "array"
                      }
                    },
                    "type": "object"
                  },
                  "key": {
                    "description": "The blockchain signing key that will sign the invocation. Defaults to the first signing key of
↪  the organization that operates the node",
                    "type": "string"
                  },
                  "options": {
                    "description": "A map of named inputs that will be passed through to the blockchain connector",
                    "type": "object"
                  }
                },
                "type": "object"
              }
            }
          },
          "description": "Success"
        },
        "default": {
```

```
            "description": ""
          }
        },
        "tags": [
          ""
        ]
      }
    },
    "/query/GetLevel2Block": {
      "post": {
        "operationId": "query_GetLevel2Block",
        "parameters": [
          {
            "description": "Server-side request timeout (milliseconds, or set a custom suffix like 10s)",
            "in": "header",
            "name": "Request-Timeout",
            "schema": {
              "default": "2m0s",
              "type": "string"
            }
          }
        ],
        "requestBody": {
          "content": {
            "application/json": {
              "schema": {
                "properties": {
                  "idempotencyKey": {
                    "description": "An optional identifier to allow idempotent submission of requests. Stored on the transaction
↪  uniquely within a namespace",
                    "type": "string"
                  },
                  "input": {
                    "description": "A map of named inputs. The name and type of each input must be compatible with the FFI description
↪  of the method, so that FireFly knows how to serialize it to the blockchain via the connector",
                    "properties": {
                      "clinic_UUID": {
                        "type": "string"
                      },
                      "end_date": {
                        "type": "string"
                      },
                      "start_date": {
                        "type": "string"
                      }
                    },
                    "type": "object"
                  },
                  "key": {
                    "description": "The blockchain signing key that will sign the invocation. Defaults to the first signing key of the
↪  organization that operates the node",
                    "type": "string"
                  },
                  "options": {
                    "description": "A map of named inputs that will be passed through to the blockchain connector",
                    "type": "object"
                  }
                },
                "type": "object"
              }
            }
          }
        },
        "responses": {
          "200": {
            "content": {
              "application/json": {
                "schema": {
                  "properties": {
                    "idempotencyKey": {
                      "description": "An optional identifier to allow idempotent submission of requests. Stored on the transaction
↪  uniquely within a namespace",
```

```
                    "type": "string"
                  },
                  "input": {
                    "description": "A map of named inputs. The name and type of each input must be compatible with the FFI
↪  description of the method, so that FireFly knows how to serialize it to the blockchain via the connector",
                    "properties": {
                      "": {
                        "details": {
                          "type": "object",
                          "properties": {
                            "type": "string"
                          }
                        },
                        "type": "array"
                      }
                    },
                    "type": "object"
                  },
                  "key": {
                    "description": "The blockchain signing key that will sign the invocation. Defaults to the first signing key of
↪  the organization that operates the node",
                    "type": "string"
                  },
                  "options": {
                    "description": "A map of named inputs that will be passed through to the blockchain connector",
                    "type": "object"
                  }
                },
                "type": "object"
              }
            }
          },
          "description": "Success"
        },
        "default": {
          "description": ""
        }
      },
      "tags": [
        ""
      ]
    }
  },
  "/query/GetLogHashWithLogID": {
    "post": {
      "operationId": "query_GetLogHashWithLogID",
      "parameters": [
        {
          "description": "Server-side request timeout (milliseconds, or set a custom suffix like 10s)",
          "in": "header",
          "name": "Request-Timeout",
          "schema": {
            "default": "2m0s",
            "type": "string"
          }
        }
      ],
      "requestBody": {
        "content": {
          "application/json": {
            "schema": {
              "properties": {
                "idempotencyKey": {
                  "description": "An optional identifier to allow idempotent submission of requests. Stored on the transaction
↪  uniquely within a namespace",
                  "type": "string"
                },
                "input": {
                  "description": "A map of named inputs. The name and type of each input must be compatible with the FFI description
↪  of the method, so that FireFly knows how to serialize it to the blockchain via the connector",
                  "properties": {
```

```
                    "logID": {
                      "type": "string"
                    }
                  },
                  "type": "object"
                },
                "key": {
                  "description": "The blockchain signing key that will sign the invocation. Defaults to the first signing key of the
↪  organization that operates the node",
                  "type": "string"
                },
                "options": {
                  "description": "A map of named inputs that will be passed through to the blockchain connector",
                  "type": "object"
                }
              },
              "type": "object"
            }
          }
        }
      },
      "responses": {
        "200": {
          "content": {
            "application/json": {
              "schema": {
                "properties": {
                  "idempotencyKey": {
                    "description": "An optional identifier to allow idempotent submission of requests. Stored on the transaction
↪  uniquely within a namespace",
                    "type": "string"
                  },
                  "input": {
                    "description": "A map of named inputs. The name and type of each input must be compatible with the FFI
↪  description of the method, so that FireFly knows how to serialize it to the blockchain via the connector",
                    "properties": {
                      "": {
                        "type": "string"
                      }
                    },
                    "type": "object"
                  },
                  "key": {
                    "description": "The blockchain signing key that will sign the invocation. Defaults to the first signing key of
↪  the organization that operates the node",
                    "type": "string"
                  },
                  "options": {
                    "description": "A map of named inputs that will be passed through to the blockchain connector",
                    "type": "object"
                  }
                },
                "type": "object"
              }
            }
          },
          "description": "Success"
        },
        "default": {
          "description": ""
        }
      },
      "tags": [
        ""
      ]
    }
  },
  "/query/GossipFromNode": {
    "post": {
      "operationId": "query_GossipFromNode",
      "parameters": [
```

```
          {
            "description": "Server-side request timeout (milliseconds, or set a custom suffix like 10s)",
            "in": "header",
            "name": "Request-Timeout",
            "schema": {
              "default": "2m0s",
              "type": "string"
            }
          }
        ],
        "requestBody": {
          "content": {
            "application/json": {
              "schema": {
                "properties": {
                  "idempotencyKey": {
                    "description": "An optional identifier to allow idempotent submission of requests. Stored on the transaction
↪  uniquely within a namespace",
                    "type": "string"
                  },
                  "input": {
                    "description": "A map of named inputs. The name and type of each input must be compatible with the FFI description
↪  of the method, so that FireFly knows how to serialize it to the blockchain via the connector",
                    "properties": {
                      "PatientLogIDs": {
                        "type": "array"
                      },
                      "clinID": {
                        "type": "string"
                      },
                      "contact_time": {
                        "type": "string"
                      },
                      "data": {
                        "type": "array"
                      },
                      "end": {
                        "type": "string"
                      },
                      "hashedBlocks": {
                        "type": "array"
                      },
                      "start": {
                        "type": "string"
                      }
                    },
                    "type": "object"
                  },
                  "key": {
                    "description": "The blockchain signing key that will sign the invocation. Defaults to the first signing key of the
↪  organization that operates the node",
                    "type": "string"
                  },
                  "options": {
                    "description": "A map of named inputs that will be passed through to the blockchain connector",
                    "type": "object"
                  }
                },
                "type": "object"
              }
            }
          }
        },
        "responses": {
          "200": {
            "content": {
              "application/json": {
                "schema": {
                  "properties": {
                    "idempotencyKey": {
                      "description": "An optional identifier to allow idempotent submission of requests. Stored on the transaction
↪  uniquely within a namespace",
```

```
                    "type": "string"
                  },
                  "input": {
                    "description": "A map of named inputs. The name and type of each input must be compatible with the FFI
↪  description of the method, so that FireFly knows how to serialize it to the blockchain via the connector",
                    "properties": {
                      "": {
                        "type": "string"
                      }
                    },
                    "type": "object"
                  },
                  "key": {
                    "description": "The blockchain signing key that will sign the invocation. Defaults to the first signing key of
↪  the organization that operates the node",
                    "type": "string"
                  },
                  "options": {
                    "description": "A map of named inputs that will be passed through to the blockchain connector",
                    "type": "object"
                  }
                },
                "type": "object"
              }
            }
          },
          "description": "Success"
        },
        "default": {
          "description": ""
        }
      },
      "tags": [
        ""
      ]
    }
  }
},
"servers": [
  {
    "url": "http://127.0.0.1:5000/api/v1/namespaces/default/apis/BLOC"
  }
]
}
```

A.10 Offline hours Scalability Raw Sync Times

Hours offline	Average Sync time (min)	Fastest Time (min)	Fastest Machine	Slowest Time (min)	Slowest Machine
2	1.472	0.865	iMac2014	2.933	MBP2015
4	2.703	1.575	Mini	5.463	MBP2015
8	5.189	3.354	Mini	10.814	MBP2015
16	9.739	6.268	iMac2020	21.437	MBP2015
32	18.715	12.162	Mini	42.954	MBP2015
64	37.032	22.165	iMac2014	91.776	MBP2015

Table A.1: Average network recovery and synchronization time data for all realms in the ABLOC prototype

Hours offline	MBP2015 sync time (mins)	Mini sync time (mins)	iMac2020 sync time (mins)	iMac 2014 sync time (mins)	MBPBCS sync time (mins)
2	2.93328	1.0015	1.2796	0.8653	1.2782
4	5.4630	1.5747	2.5464	1.7042	2.2245
8	10.8141	3.3536	3.4605	3.7033	4.6114
16	21.4369	6.70652	6.2684	7.2501	7.0315
32	42.9543	12.1622	13.5171	12.5949	12.3475
64	91.7762	24.0284	23.4355	22.1646	23.7547

Table A.2: Average Machine recovery and synchronization time (minutes)