AFRL-RI-RS-TR-2020-069

SENSOR-CHAIN: A DEMO OF A LIGHTWEIGHT BLOCKCHAIN BASED INTERNET OF MILITARY THINGS SECURITY SCHEME

FLORIDA INTERNATIONAL UNIVERSITY

APRIL 2020

FINAL TECHNICAL REPORT

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED

STINFO COPY

AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE

■ **AIR FORCE MATERIEL COMMAND** ■ **UNITED STATES AIR FORCE** ■ **ROME, NY 13441**

NOTICE AND SIGNATURE PAGE

Using Government drawings, specifications, or other data included in this document for any purpose other than Government procurement does not in any way obligate the U.S. Government. The fact that the Government formulated or supplied the drawings, specifications, or other data does not license the holder or any other person or corporation; or convey any rights or permission to manufacture, use, or sell any patented invention that may relate to them.

This report is the result of contracted fundamental research deemed exempt from public affairs security and policy review in accordance with SAF/AQR memorandum dated 10 Dec 08 and AFRL/CA policy clarification memorandum dated 16 Jan 09. This report is available to the general public, including foreign nations. Copies may be obtained from the Defense Technical Information Center (DTIC) (http://www.dtic.mil).

AFRL-RI-RS-TR-2020-069 HAS BEEN REVIEWED AND IS APPROVED FOR PUBLICATION IN ACCORDANCE WITH ASSIGNED DISTRIBUTION STATEMENT.

FOR THE CHIEF ENGINEER:

/ S /
TODD N. CUSHMAN
Work Unit Manager

/ S /
JAMES S. PERRETTA
Chief, Information Exploitation
& Operations Division
Information Directorate

This report is published in the interest of scientific and technical information exchange, and its publication does not constitute the Government's approval or disapproval of its ideas or findings.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. REPORT DATE *(DD-MM-YYYY)*	2. REPORT TYPE	3. DATES COVERED *(From - To)*
APRIL 2020	FINAL TECHNICAL REPORT	MAY 2019 – DEC 2019

4. TITLE AND SUBTITLE

SENSOR-CHAIN: A DEMO OF A LIGHTWEIGHT BLOCKCHAIN BASED INTERNET OF MILITARY THINGS SECURITY SCHEME

5a. CONTRACT NUMBER
N/A

5b. GRANT NUMBER
FA8750-19-1-0022

5c. PROGRAM ELEMENT NUMBER
62788F

6. AUTHOR(S)

Niki Pissinou

5d. PROJECT NUMBER
BC2S

5e. TASK NUMBER
CI

5f. WORK UNIT NUMBER
MT

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Florida International University
11200 SW 8th St
Miami, FL 33199

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Air Force Research Laboratory/RIGA
525 Brooks Road
Rome NY 13441-4505

10. SPONSOR/MONITOR'S ACRONYM(S)
AFRL/RI

11. SPONSOR/MONITOR'S REPORT NUMBER
AFRL-RI-RS-TR-2020-069

12. DISTRIBUTION AVAILABILITY STATEMENT
Approved for Public Release; Distribution Unlimited. This report is the result of contracted fundamental research deemed exempt from public affairs security and policy review in accordance with SAF/AQR memorandum dated 10 Dec 08 and AFRL/CA policy clarification memorandum dated 16 Jan 09.

13. SUPPLEMENTARY NOTES

14. ABSTRACT

The Internet of Military Things (IoMT) consists of an increasing number of ubiquitous sensing and computing devices worn by military personnel and embedded within military equipment that are capable of acquiring a variety of static and dynamic biometrics and collecting operational context data that can be used to perform context-adaptive authentication in-the-wild and in a dedicated edge computing architecture. This demo, coined "Sensor-Chain", promises a new generation of lightweight blockchain management with superior reduction in resource consumption, and, at the same time capable of retaining critical information about IoMTs.

15. SUBJECT TERMS
Block chain, Internet of Things, cybersecurity, distributed ledger, embedded systems, hyperledger, internet of military things, partition tolerance

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			**TODD N. CUSHMAN**
U	U	U	UU	82	19b. TELEPHONE NUMBER (*Include area code*) **N/A**

TABLE OF CONTENTS

List of Figures

List of Tables

1.0 SUMMARY

The Internet of Military Things (IoMT) consists of an increasing number of ubiquitous sensing and computing devices worn by military personnel and embedded within military equipment that are capable of acquiring a variety of static and dynamic biometrics and collecting operational context data that can be used to perform context-adaptive authentication in-the-wild and in a dedicated edge computing architecture. To secure IoMT deployments, blockchain technology has emerged as a way of recording digital interactions in a way that is designed to be secure, transparent, highly resistant to outages, auditable, and efficient. Although blockchains are considered as the key to redesign IoMT systems, they cannot be directly integrated into IoMT systems. In particular, since the chain is always growing, IoMT nodes require more and more resources. Thus, an oversized chain poses storage and scalability problems. With this in mind, the overall goal of this proposal is to design, develop and demonstrate a lightweight blockchain-based Internet of Military Things (IoMT) Security scheme. This demo, coined “Sensor-Chain”, promises a new generation of lightweight blockchain management with superior reduction in resource consumption, and, at the same time capable of retaining critical information about IoMTs.

2.0 INTRODUCTION

The overall goal of this technical report is to present and demonstrate the design and development details of a lightweight blockchain-based Internet of Military Things (IoMT) Security scheme. This demo, coined "Sensor-Chain", is depicted in Figure 1 and promises a new generation of lightweight blockchain management with superior reduction in resource consumption, and, at the same time capable of retaining critical information about IoMTs.

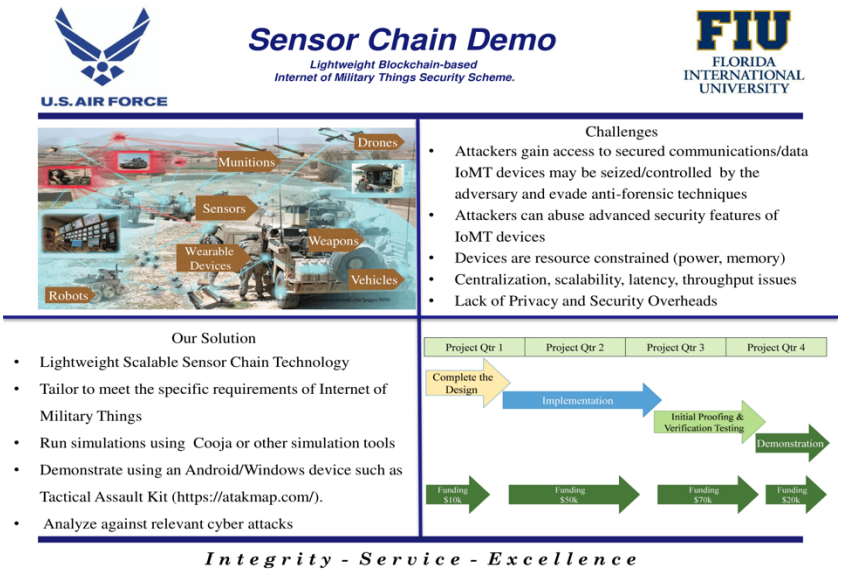

Figure 1. Sensor-Chain demo. PI: Niki Pissinou

Over the past several years, there has been a surge of interest on systems that connect physical infrastructures with machine intelligence, information and communication technologies based on sensors and Wireless Sensor Networks (WSN). Today, these uniquely identifiable objects and their virtual representations in an internet like structure are referred to as the "Internet of Things (IoT)" [1]. While the new capabilities of IoT technology promise to impact many commercial and civilian applications, its machine intelligence and networked communications will also have a tremendous influence on military applications [2]. For example, research conducted at the US Army Research Laboratory predicted a scale on the order of a million of things per square kilometer of IoT [2]. Coined as the "Internet of Military Things(IoMT)" [3],"the IoMT (or Internet of Battlefield Things), consists of an increasing number of ubiquitous sensing and computing devices worn by military personnel and embedded within military equipment that are capable of acquiring a variety of static and dynamic biometrics and collecting operational context data that can be used to perform context-adaptive authentication in-the-wild and in a dedicated edge computing architecture [3]."

Before realizing the new promise of the IoMT (figure 2), there are significant challenges to address. First, we must overcome the limitations of its centralized model. Depending on the use, this can be done by using a Peer-to-Peer network paradigm. Since the IoMT network topology is prone to frequent changes due to node failure, damage, energy depletion, or channel fading, the peer-to-peer paradigm can adapt to change faster. Also, as peer-to peer networks allow direct communication between devices, it can lead to faster communication among devices. As such, the peer-to-peer approach has gained significant attention in IoMT applications, but tapping into the benefits of a ubiquitous peer-to-peer connectivity is not without challenges. The first challenge is the scale issue around data collection, storage, and analytic. Since IoMT devices possess limited

computational power and storage capabilities, and are characterized by lossy, low communication channels, these characterizes affect the design of protocols for the IoMT domain. Often, to reduce memory requirements, the size and number of messages is minimized. To reduce the memory and resource consumption of security protocols the use of resource intensive cryptography primitives is limited. In this situation, the adversary can seize or control IoMT devices or networks and gain access to secured communications and data. Devices seized in an attack can be controlled by an adversary because the adversary can evade anti-forensic techniques and abuse advanced security features of IoMT devices. Thus, despite the potential advantages of IoMTs, such as greater and faster tactical-level situational awareness, improved command and control of combined operations logistics support, monitoring vehicle and soldier status, trust management security, privacy and and transparency remain a concern.

2.1 Background

To secure IoMT deployments, blockchain technology has emerged as a way of recording digital interactions in a way that is designed to be secure, transparent, highly resistant to outages, auditable, and efficient [4]. Blockchain is essentially a data structure or public ledger of sequence of blocks that constantly grows as newly created blocks are added to record the up-to-date transactions [5], [6]. It provides built-in integrity of information, and security of immutability by design, making it very useful to ensure trust, security, and transparency in Peer-to-Peer (P2P) trustless networks of huge number of devices. Although blockchains are considered as the key to redesign IoMT systems, they cannot be directly integrated into IoMT systems. Since the chain is always growing, IoMT nodes require more and more resources. Thus, an oversized chain poses storage and scalability problems.

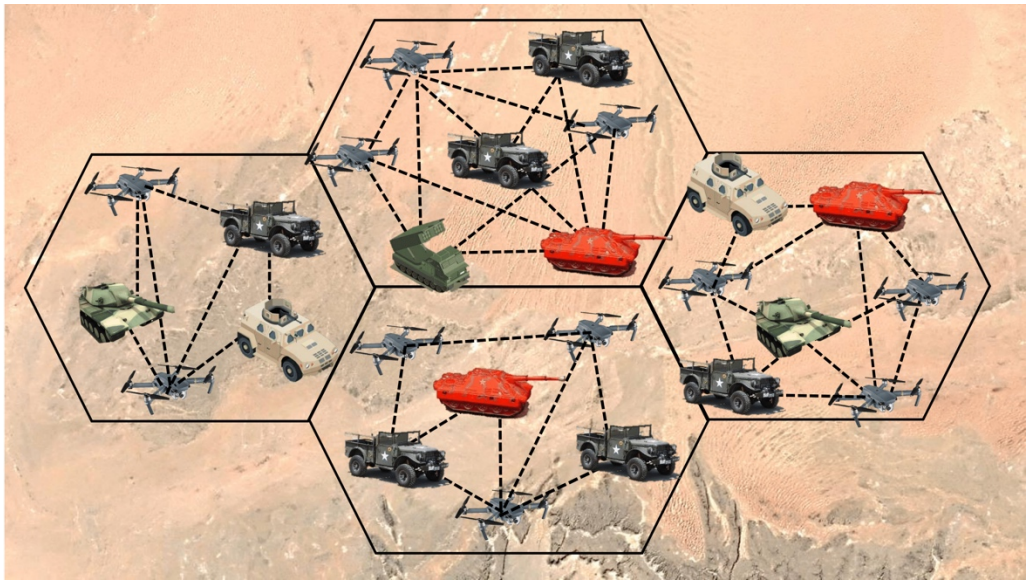

Figure 2. Future IoMT architecture

First, managing blockchain with limited storage space remains a challenge. Sensors, unlike high-end computing devices, used for cryptocurrencies, are equipped with very limited storage space. Consequently, an ever-growing blockchain will take over the whole storage space of sensors in a very short period of time after the creation of genesis block [7]. Similarly, scalability with constrained computing power and battery also poses a challenge. In a distributed IoT system, each sensor node is assumed to be connected to all other nodes in the network. Thus, with an integration of blockchain, each node needs to perform large number of tasks at different stages of the blockchain with their constrained computing power and battery life. The growth of the network further aggravates the problem. These two issues are rooted to the problem of managing the number of transactions required to be stored and processed by a single sensor. If we take a look at the different elements of a block (figure 3), we observe that it contains some elements which take constant storage space (the elements of block header and transaction counter). While each transaction size can be within a limit, it's their number which is the only dominating variable in a block. Thus, we can express the size of a blockchain as a function of number of transactions stored in it. For better understanding, let us consider a conventional blockchain for a network of n number of nodes where each node performs a transaction with every other node at each timestamp in a worst-case scenario. That is, there are $\frac{n(n-1)}{2}$ number of transactions are happening every timestamp. Then the size of the blockchain,

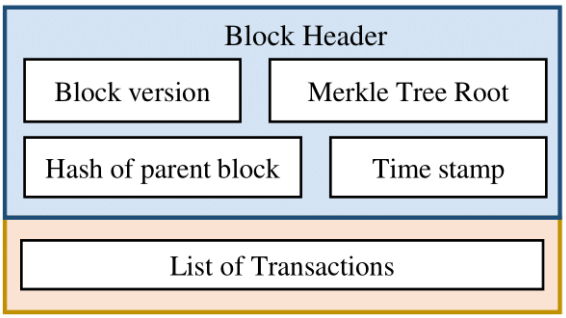

Figure 3. A block structure.

$$\begin{aligned}\text{Size of a blockchain} &= TX^1 + TX^2 + \cdots + TX^T \\ &= \frac{n(n-1)}{2} + \frac{n(n-1)}{2} + \cdots + \frac{n(n-1)}{2} \\ &= \sum_{t=1}^{t=T} \frac{n(n-1)}{2} \\ &= T \times \frac{n(n-1)}{2}\end{aligned} \tag{1}$$

Where, T is the lifetime of the blockchain and TX^t refers to the total number of transactions happened at time t. Let us understand the issue with an example. If $n = 1000$, the nodes perform transactions at every 10 seconds, and each transaction takes $.01$ KB space to be stored in a blockchain; then the size of the blockchain after 10 hours $1000 \times \frac{(1000-1)}{2} \times 6 \times 60 \times 10 \times 0.1$ GB = 17.982 GB, which is far beyond the storage and processing capabilities of sensors. Understandably, in some IoT applications, the transactions may happen at a longer interval (e.g. 30 minutes) and every timestamp a smaller number of nodes perform transactions than $\frac{n(n-1)}{2}$.

However, it is not going to change the fact that, existing blockchains are not feasible for resource-constrained sensors in the long-run.

Severity of the problem of managing blockchain with limited resources can go beyond the mere "out of storage space" issue to critical cyberattack on IoMT network. For instance, adversaries can take over few IoMT nodes in the network and continue performing frequent transactions with other nodes without violating any protocol of the system. Eventually, the blockchain will be large enough such that some nodes will be out of the network due to their lack of power and storage space, allowing attackers to compromise the network. All these problems highlight that the storage and scalability problems of blockchain for resource-constrained sensors must be addressed before moving forward with blockchain-based IoMT.

Understanding the limitations of existing blockchain solutions, recent focus has been shifted to developing lightweight decentralized architecture based on blockchain. Large number of the research works in this area [7], [8] discussed the impact of blockchain on IoT and important research issues that are required to be addressed to fully realize the benefit of blockchain. The existing research efforts can be categorized into devising approach to integrate blockchain with IoT [9]-[12], node authentication and access control [13]-[16], trust management [8], [17]-[19], and security and privacy [20]-[22]. These different research works have one thing in common: they either simply considered that IoMT devices are equipped with enough storage and computing resources to hold and process blockchains, or utilized high end edge computing devices to manage the blockchain. The assumptions of having enough resources is hard to get on with IoMT devices, making the applicability of the research works, based on such assumptions, questionable. For instance, trust and authentication management for wireless sensor networks using blockchain was proposed in [17] without hinting how the sensors will manage the blockchain on their own local space. Likewise, the Block-VN architecture for distributed transport management system [23], based on a permissioned blockchain, considered that at least some portion of the vehicles are capable of storing and processing an ever-growing blockchain. Another example is the IoT-based Machine-to-Machine payment system, known as IOTA [24]. IOTA uses proof-of-work consensus protocol, which makes the new block creation task both computationally expensive and time consuming. Thus, in IOTA the hardware requirement is too high and it is hard to meet such requirement for IoMT sensor nodes.

Realizing the resource issues of the IoT devices, many researches proposed to offload the blockchain onto edge computing devices. The SpeedyChain [25] data sharing framework for intelligent vehicles suggested to use roadside infrastructure units (RSIs) and service providers (SPs) to maintain blockchain. In SpeedyChain, RSIs are responsible for trust and authentication management and trusted vehicles, verified by the RSIs, can append block to the blockchain. In a similar way, a Roadside Units (RSU) based blockchain trust management for vehicular network was proposed in [26]. In this work, each vehicle generates a rating for its neighboring vehicles and share the rating with nearby RSU. With all most recently received ratings, RSUs calculate the trust value offsets of involved vehicles and gather these data into a block. In order to insert the new block into the blockchain, the authors proposed a combination of proof-of-work and proof-of-stake, improving each other. In contemporary works, Xiong et al. [9], [27] proposed to deploy multiple access mobile edge computing devices to carry out the computationally expensive proof-

of-work and introduced game theoretic approach for edge computing resource management. In these works, the sensors are considered as ordinary nodes, and the edge devices are responsible to carry out the blockchain operations. The "EdgeChain" framework [10] extended this idea by introducing credit-based resource management system to control the edge server resource consumption by an individual IoT device. In [13], a smart contract-based access mechanism was put forward with the aim of simplifying the process of blockchain management and reducing the communication overhead between the nodes. In this mechanism, the IoT devices are kept out of the blockchain as they cannot hold a large blockchain. Rather, a special node called management hub is proposed to put as a link between IoT devices and blockchain. A blockchain framework was proposed for smart homes [28], where the information produced by smart home devices are stored in the blockchain. In this architecture, the blockchain is maintained in the gateways and is isolated from the devices. Similar to the other works on blockchain based Internet of Vehicles, kang et al. [29] also considered RSUs as edge computing infrastructures for blockchain management. This approach utilized a modified high-efficiency Delegated Proof-of-Stake (DPoS) consensus scheme where instead of stake-based voting, reputation is used for miner RSU selection.

Through a careful observation of all these approaches, one can figure it out that that they all tried to solve the storing and processing heavyweight blockchain problems by employing more powerful computing devices in the architecture. Besides being costly, these devices are mostly static and are deployed in a predefined structured way. Such structured deployment of static devices is hard to be acceptable in IoMT, as the network topology is prone to changes faster than in many other IoT scenarios (e.g. smart home, IoV with predefined road network, and so on). Furthermore, the management of blockchain using fixed positioned edge devices naturally makes the system more vulnerable, as compromise of few edge devices will affect a large portion of the sensors nodes in the IoMT network.

One viable solution to make blockchain "manageable" for sensors without using any edge or other devices is limiting the size of the blockchain within the resource capacity of sensors. The "temporal blockchain" framework proposed a solution based on such concept [30]. It was proposed to delete all the blocks older than a preset period (e.g. 30 days old) from the blockchain. While this approach can reduce the size of the blockchain, it still lacks in guaranteeing limited storage capacity with the growth of the network in the long-run in IoT scenario. Moreover, how to deal with the loss of information due to the deletion of blocks was not addressed.

This investigation highlights that existing blockchain frameworks for IoMT lack a clear understanding of resource management issues for blockchain in IoMT scenario. Lack of such understanding not only makes the frameworks impractical for IoMT, but also allow adversaries to dismantle a system through a variety of denial-of-service (DoS) attacks [31], such as sleep deprivation attack [32] and buffer overflow attack [33]. Equally important, the absence of a proper transaction validation method in those frameworks can allow an adversary to inject false data in an IoMT system in a legitimate way [34]. The research on blockchain and IoMT has a long way to go, and we emphasize that before taking further steps, we must have an efficient approach to make blockchain lightweight and scalable for IoMT sensors in a secure way.

In light of this, this investigation develops a lightweight scalable practical blockchain security scheme for IoMT and demonstrates the framework in a simulated environment with field spatiotemporal data.

3.0 METHODS, ASSUMPTIONS, AND PROCEDURES

This section presents the Sensor-Chain framework. We first discuss 3 different frameworks: Conventional and our proposed improved temporal, and spatial blockchains. We analyze their strength and limitations to highlight the motivation behind the design of Sensor-Chain framework.

3.1 System Model and Assumptions

The proposed system model has two major entities: 1) a region, divided into a set of smaller cells, and 2) a set of sensor nodes. Some of the sensors are static and others are mobile. The mobile nodes are moving over the region based on Random Waypoint Mobility model [34]. Each sensor node is capable of performing lightweight aggregate operations, such as e.g. max, mean, min, weighted average [35] and so on. Furthermore, the proposed system does not require any additional resources. We assume that the distribution of the sensed data within a cell is approximately same. The proposed blockchain can be either a public blockchain or a permissioned-blockchain. If it is public blockchain, there is no authority in the blockchain and nodes can join and leave the network with random cryptographic key pairs. In such a blockchain, we assume that the nodes are using a lightweight consensus algorithm, such as Proof-of-Stake (PoS). Our work is also applicable in permissioned-blockchain where an authority assigns each IoT Node a private key and a private key and to join a network a node needs to reveal its identity to all the other nodes in the network. In order to achieve conditional privacy from the peers, an IoT node can anytime request the authority for new key pairs. In such a case, we assume that the nodes are using a Byzantine Fault tolerant algorithm for reaching consensus in the network. Devising novels mechanisms for Key management and authentication are beyond the scope of this work.

3.2 Methodology

In the conventional blockchain frameworks [9], [17] a blockchain is managed by all the nodes in the network and continues to grow with the lifespan of the network. Thus, with a $T = \infty$ lifespan, according to our discussion on the size of blockchain, the size of a conventional blockchain becomes,

$$size(conventional) = \sum_{t=1}^{\infty} \frac{n(n-1)}{2}$$

Obviously, this blockchain will impose a high storage requirement which cannot be met by sensor nodes. To improve this, we then design an improved version of temporal blockchain [30] in the context of mobile IoT.

In the original temporal blockchain [30], it was proposed to keep a portion of the blockchain after certain time period. However, we propose to replace the blockchain with an aggregated version of it after certain a time period. In detail, in the preprocessing step of our scheme, we consider a specific time at the "genesis time", and a time period is set as the temporal constraint for blockchain deletion. For example, if 00:00 in 24-hour format is taken as the genesis time and the temporal constraint is 2 hours, then the deletion operation will take place at 02:00, 04:00, 06:00, ... of each day. This genesis time information and temporal constraint are preset onto the IoT devices. Another way to set this information is to have smart contract on the blockchain. We leave this for our future research. Every time the lifetime of the blockchain meets the temporal constraints, through the consensus mechanism, a node will be selected as an aggregator node which performs aggregation over the whole blockchain and creates an aggregated block. This block includes the ID of the aggregator node. This block is then broadcasted over the network by the aggregator. This aggregation could be anything lightweight for IoT sensor devices to perform (e.g. min, max, mean, weight average [35]). Upon receiving this block, the nodes in the network replaces the whole existing blockchain with this block on their local storage. That is, it will be considered as the genesis block of a new blockchain. Even though as a consequence the newly restarted blockchain's size becomes relatively small, we still need to look into the size of the blockchain between two consecutive restarts so as to ensure that it is within the storage space capacity of the IoT sensor node. If the temporal constraint is T_{chain}, then in the the worst-case scenario, the maximum size of the blockchain can be,

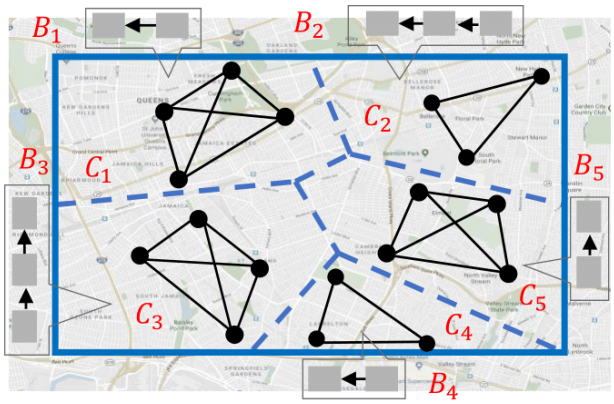

Figure 4. A Voronoi diagram of a region with local networks and local blockchains

$$size(improved - temporal) = \sum_{t=1}^{t=T_{chain}} \frac{n(n-1)}{2}$$

Clearly this scheme outperforms the conventional blockchain schemes. However, with higher T_{chain} and a large number of nodes in a network, the nodes still need to hold a large blockchain, making it quite impractical for IoT devices. Thus, despite the fact that a temporal blockchain can reduce the size of a chain, the size of a chain must be further improved when dealing with IoT nodes. This is done using the following spatial blockchain technique.

In our spatial blockchain framework, a global blockchain is broken down into smaller disjoint *local blockchains* with the aim of reducing the number of transactions performed by a node at any given time than in conventional blockchain frameworks. To achieve this objective, we translate a region into a Voronoi diagram [36]. Voronoi diagram $\mathbb{C}$, is a partitioning of a plane into non-overlapping smaller convex regions, called Voronoi cells $\mathbb{C}$. Based on this partitioning of the plane, we define two different structures: local networks and local blockchains (figure 4 depicts

these structures). A *local network* refers to the graph $G_i^t = (V_i^t, E_i^t)$ formed by the nodes in the cell $C_{i \in \mathbb{C}}$ at time t. Here, V_i^t and E_i^t are the set of the nodes and the edges between them. Any two local networks of two different cells at the same time are disjoint. That is,

$$V_i^t \cap V_j^t = \emptyset, \quad E_i^t \cap E_j^t = \emptyset \tag{2}$$

A *local blockchain* B_i, is the blockchain managed by the nodes in cell C_i and B_i^t is the snapshot of B_i at time t. Any two local blockchains from two different cells have the following property: a block of a local blockchain in a cell is neither a parent nor a child of a block of another local blockchain in another cell at any time instance. That is,

$$\left(\exists b_i^x \in B_i \middle| b_i^x \text{ is a parent of a block in } B_j\right) \cup$$

$$\left(\exists b_j^y \in B_j \middle| b_j^y \text{ is a parent of a block in } B_i\right) = \emptyset; \forall t \tag{3}$$

The two properties imply that a sensor node in G_i works only on local blockchain B_i. Hence, it needs to store only the copy of B_i at any given time as long as it remains in G_i.

While this definitely improves the storage issue than in conventional blockchain, this scheme further enhances its efficacy considering mobility of the nodes. In case of mobility, if a node moves from cell C_i to C_j, at first it deletes the copy of local blockchain B_i from its memory and then, after joining G_j, it downloads the copy of B_j from its peers. Thus, a node is required to store only one local blockchain at any time instance, which significantly reduces the required space to store a blockchain. We quantify the storage requirement of this scheme as follows. Let us consider that at any time instance, there could be at most m number of nodes in a cell, where $m < n$ and the time difference between the creation of genesis block and current time is $\approx \infty$. Let us also assume that a mobile node's permanence in a cell is at most T_{per}. At the first glance, it seems $size(spatial) = \sum_{t=1}^{t=t_{par}} \frac{m(m-1)}{2}$. However, consider the worst-case scenario where there exists at least one node in a particular cell C_i all the time (if some nodes are static or the cell is never empty). That is, the local blockchain continues to expand forever. In that case,

$$size(spatial) = \sum_{t=1}^{t=\infty} \frac{m(m-1)}{2} \; ; \quad m < n \tag{4}$$

From the analysis of temporal and spatial blockchains, it is not clear which one offers the best solution. For static nodes, the temporal blockchain with a small temporal constraint could be the better solution in the long run. On the other hand, in mobile environment, the spatial blockchain will be the winner. To address the limitations of both approaches, we propose Sensor-Chain approach.

Sensor-Chain is a fusion of both temporal and spatial blockchain approaches. Similar to spatial blockchain, in this framework, a complete region is first divided into a number of Voronoi cells. Using those cells, the nodes in a cell form a local network and maintain a local blockchain. All the local networks and local blockchains follow the properties defined for spatial blockchain. Among different information, each nodes holds the following tuple: $\{current\ cell\ id, C_{cur}, copy\ of\ the\ local\ blockcahin\ B_{cur}^{t}\}$. In order to manage the size of a blockchain, this framework has two important constraints: temporal constraint T_{chain} and block creation time constraint T_{block}. The storage management of blockchain is done in two ways: spatiotemporal and mobility-based.

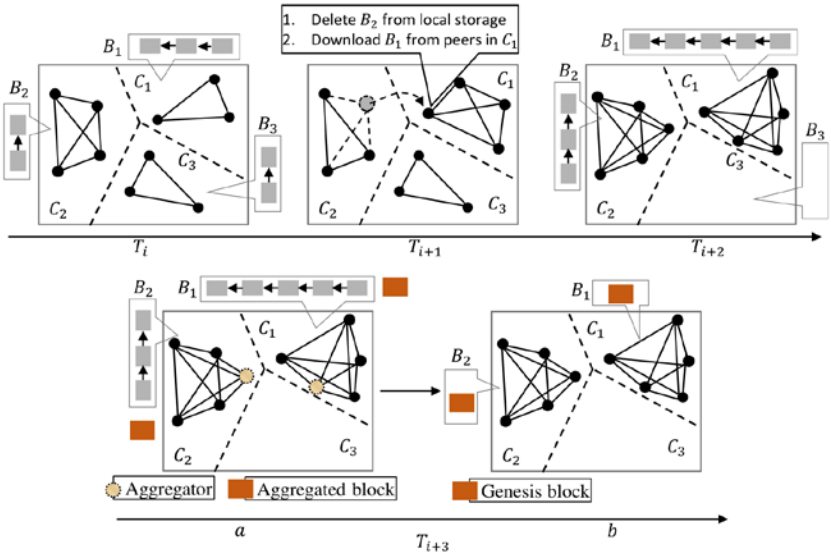

Figure 5. Illustrated Sensor-Chain

Spatiotemporal-based blockchain management is detailed in algorithm 1. In this framework, the block creation and insertion are done at a fixed time interval (lines 1-6), a similar approach of bitcoin. At first, in each local network G_i^t a $Miner$ is selected through consensus. Then the $Miner$ gathers all the recent transactions and creates $NewBlock$. Upon verification, the new block is inserted into B_i^t. The temporal constraint is used to reset the local blockchains at a fixed time interval. Every time the temporal constraint is met (line 8), an $Aggregator$ node is selected from each local network. This $Aggregator$ node computes aggregation of its local blockchain, creates an $AggregatedBlock$, and broadcasts it over its local network (lines 9-13). Upon receiving the $AggregatedBlock$, the nodes in the local network first delete their copy of the existing local blockchain (line 14) and then regenerate the local blockchain using the aggregated block as the genesis block (line 15).

Algorithm 5 presents the mobility-based blockchain management. Every time a node moves from one cell C_{cur} to another C_{new_cell} (line 1), it deletes the copy of the local blockchain B_{cur} of previous cell from its memory. Then it joins the network in C_{new_cell}. The work flow of Sensor-Chain is illustrated in figure 5. For figure 5, T_(i+1): A mobile node moves from cell C_2 toC_1. First, it deletes the copy of local blockchain b_2 from its memory and then downloads B_1 from its peers in G_1^(i+1). T_(i+2) : local blockchain B_3 does not exist anymore as C_3 is empty.

T_(i+3): as temporal constraint is met, (a) aggregator node from each local network is selected. The selected nodes compute aggregation over their respective local blockchains and generate aggregated blocks. (b) using the aggregated blocks as the genesis, the local blockchains are regenerated

Algorithm 1: Spatiotemporal Blockchain Management

Input : Current time T_t, set of all local networks $\mathcal{G}$ at T_t, set of all local blockchains $\mathcal{B}^t$, genesis time T_{gen}, temporal constraint T_{chain}, block creation time constraint T_{block}

Output: Updated local blockchains $\mathcal{B}^t$

if $(T_{gen} - T_t)\%T_{block} == 0$ **then**
  for *each* $G_i^t \in \mathcal{G}^t$ **do**
    $Miner \leftarrow$ Select-Miner(V_i^t)
    $NewBlock \leftarrow$ Create-Block$(Miner)$
    Insert-Block$(NewBlock)$
  end
end
if $(T_{gen} - T_t)\%T_{chain} == 0$ **then**
  for *each* $G_i^t \in \mathcal{G}^t$ **do**
    $Aggregator \leftarrow$ Select-Aggregator(V_i^t)
    $AggregatedBlock \leftarrow$ Compute-Aggregation$(B_i^t, Aggregator)$
    Broadcast$(AggregatedBlock)$
    for *each node* $v \in V_i^t$ **do**
      Delete(B_i^t) from local storage
      $B_i^t \leftarrow$ Re-generate$(B_i^t, AggregatedBlock)$
    end
  end
end

Algorithm 2: Mobility-Based Blockchain Management

Input : Voronoi diagram $\mathcal{C}$, sensor S

Output: Updated sensor S

if $S.C_{cur} \neq current\_cell$ **then**
  Delete(B_{cur}) from local storage
  Join(G_{cur}^t)
  Download(B_{cur}^t) from peers in V_{cur}^t
  $S.C_{cur} \leftarrow current\_cell$
end

We argue that, with such spatiotemporal and mobility-based blockchain management, Sensor-Chain provides the best solution. To prove its validity, we now analyze the space requirement to store a blockchain in this scheme. Referring to the discussion on spatial blockchain, with the space partitioning, the size of a local blockchain in Sensor-Chain can be at most,

$$size(sensor - chain) = \sum_{t=1}^{t=\infty} \frac{m(m-1)}{2}$$

However, as the temporal constraint T_{chain} is applied to all the local blockchains, according to the discussion on temporal blockchain, the size of a local blockchain can be further reduced as follows,

$$size(sensor - chain) = \sum_{t=1}^{t=T_{chain}} \frac{m(m-1)}{2}$$

This analysis gives us the required storage space in Sensor-Chain. Next, we analyze the scheme case by case and draw comparison with our proposed improved temporal and spatial blockchain frameworks.

In the first case, all the nodes are assumed as static. Also, the partitioning of the region is such that all the nodes reside in a single cell. In such a case, $m = n$.

$$size(Sensor - Chain) = size(improved - temporal) = \sum_{t=1}^{t=T_{chain}} \frac{n(n-1)}{2} < size(spatial)$$

Where $size(spatial) = \sum_{t=1}^{t=\infty} \frac{n(n-1)}{2}$

In the second case, all the nodes are moving in such a way that each local blockchain becomes empty (more correctly, it doesn't exist anymore) every time before the temporal constraint is satisfied. This case is depicted in figure 5(T_{i+2}) where cell C_3 is empty so that B_3 does not exist anymore. In such a case,

$$size(Sensor - Chain) = size(spatial) = \sum_{t=1}^{t<T_{chain}} \frac{m(m-1)}{2} < size(improved - temporal)$$

Where $m < n$ and $size(spatial) = \sum_{t=1}^{t=T_{chain}} \frac{n(n-1)}{2}$

In all other cases,

$(size(Sensor - Chain) < size(spatial))$ & $(size(Sensor - Chain) < size(spatial))$

4.0 RESULTS AND DISCUSSION

In this section, we first present the proof-of-concept evaluation of Sensor-Chain. The evaluation results justify the theoretical analysis of Sensor-Chain. Then, we present the implementation detail of the framework, including its class, architectural, sequence, and use case diagrams.

4.1 Proof of Concept Evaluation

To carry out the experiment we use synthetic data. The parameters and their different values used in the experiment are presented in table 1. We implemented all the four (conventional, improved-temporal, spatial, and Sensor-Chain) approaches. We ran the simulation for 6 hours and generated statistics for all the approaches. Specifically, we compared the approaches in terms of number transactions needed to be stored on a single IoT sensor device, as it defines the size of a blockchain. The evaluation is done from three different points of view: 1) duration of the simulation, 2) number of cells, and 3) number of sensors. The detail of the evaluation results is discussed below.

Table 1. Parameters used in the Experiment

Parameters	Values
Area of the region	$5000m \times 5000m$
Number of Voronoi Cells	$50, 100, 150, 200, 1000$
Number of sensor nodes	$1000, 3000, 5000, 7000$
Speed of the nodes	$[0,50]\ km/h$
Temporal constraint, T_{chain}	$1\ hour$
Block creation time constraint, T_{block}	$10\ minute$

Figure 6(a) shows the result of the simulation for Sensor-Chain. In every hour, the curve moves upward. As $T_{chain} = 1$ hour, the size of the blockchain becomes 1 (with the aggregated block) at the end of each hour. It is also clear that in Sensor-Chain, using the temporal constraint, it is possible to keep the size of the blockchain within a limit. Figure 6(b) shows the comparison between Sensor-Chain and conventional approaches. From nearly the beginning of the simulation, the required storage space in Sensor-Chain is far less than in conventional approach. Next, we evaluate how Sensor-Chain, with the fusion of spatiotemporal and mobility-based blockchain management, outperforms the improved temporal and spatial schemes. For both of the improved temporal and Sensor-Chain, we used the same temporal constraint. Although the improved temporal blockchain shows a trend similar to Sensor-Chain, its required storage space is much higher than Sensor-Chain. Figure 6(d) shows more interesting results on the comparison with spatial blockchain. In the 1^{st} hour, both spatial and Sensor-Chain approaches go toe-to-toe. However, just after the 1^{st} hour (as $T_{chain} = 1$ hour), the local blockchains in Sensor-Chain restore to genesis block, while spatial blockchain continues to grow over the time.

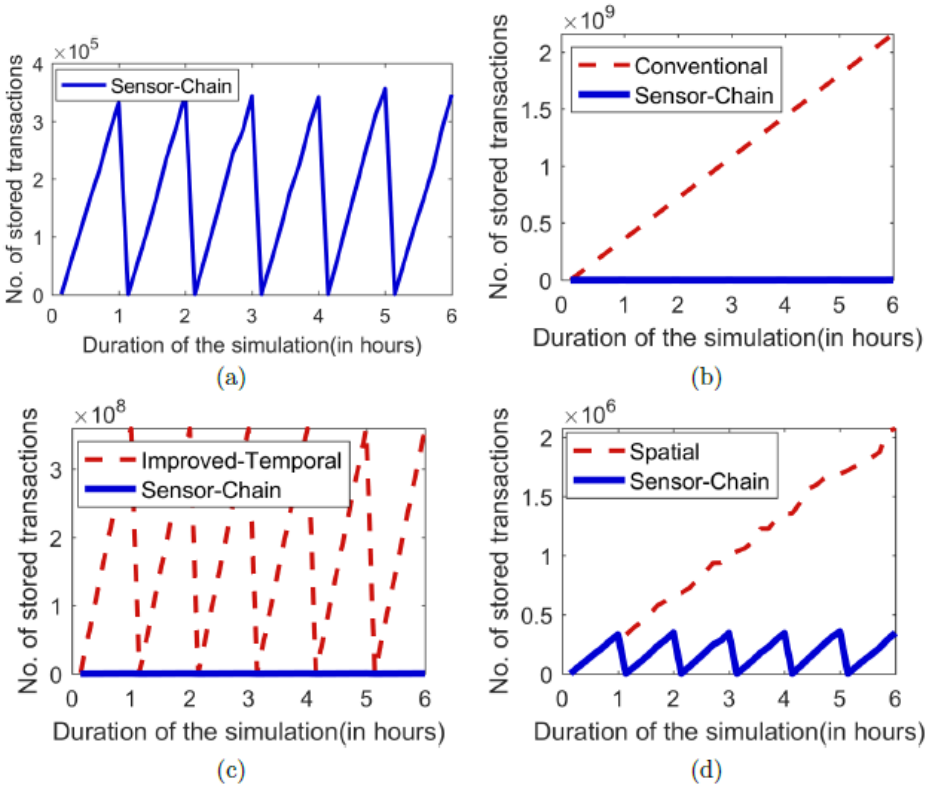

Figure 6. Evaluation results: (a) Sensor-Chain, (b) conventional, (c) improved-temporal, and (d) spatial blockchains (experiment Settings: number of cells = 50, number of sensors = 1000).

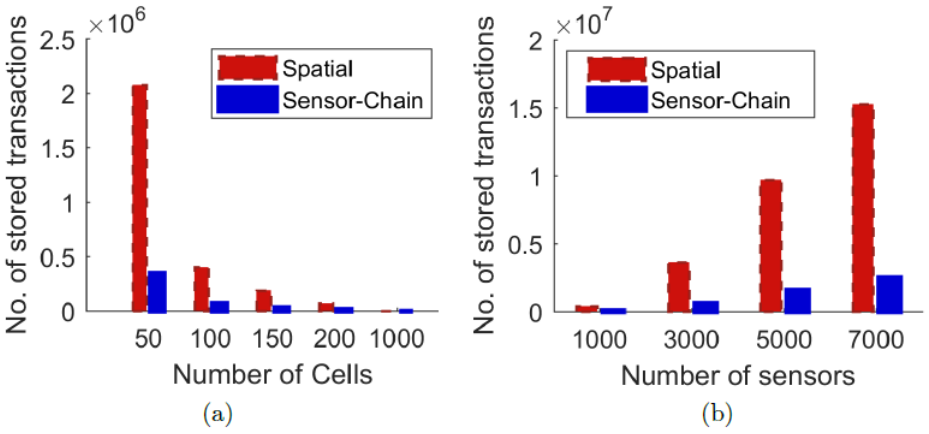

Figure 7. Comparison between Sensor-Chain and spatial approaches in terms of number of (a) cells and (b) sensors.

Then, we analyze the impact of number of cells and sensors on the size of the blockchain. As only spatial and Sensor-Chain use cell-based partitioning, here we analyze their comparison. Figure 7(a) presents the comparison result in terms of number of cells. It is understandable that with the increase in the number of cells, the size of a local blockchain decreases. Furthermore, it seems that

when this number is relatively high (e.g. 1000 in the figure), both approaches require similar storage capacity. However, it is the number of sensors that makes the difference in such a particular case. With the increase in the number of sensors, the required storage space increases rapidly in spatial approach than in Sensor-Chain. Figure 7(b) shows the results for 1000 cells with different number of sensors.

4.2 Implementation Detail of Sensor-Chain

In this section we present the detail of the implementation steps of Sensor-Chain. The development was carried out with Go programming language, an open source programming language. For P2P communication, we used \verb|go-libp2p-pubsub| library [36], an open source golang implementation of pubsub system with flooding and gossiping variants. Figure 8 presents the key components of the Sensor-Chain platform. Figures 9, 10, 11, and 12 illustrate the class, architecture, sequence, and use case diagrams of the platform.

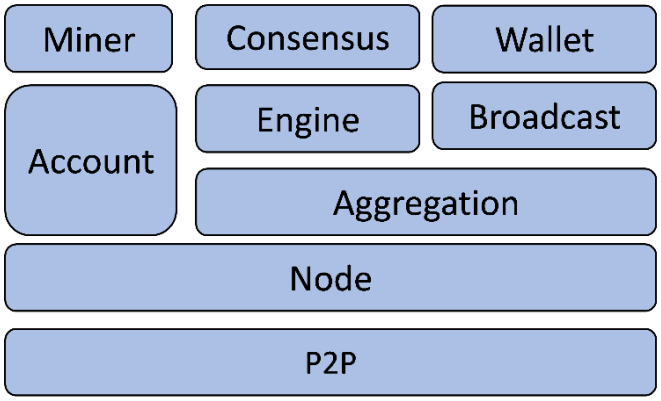

Figure 8. Key components of Sensor-Chain Framework.

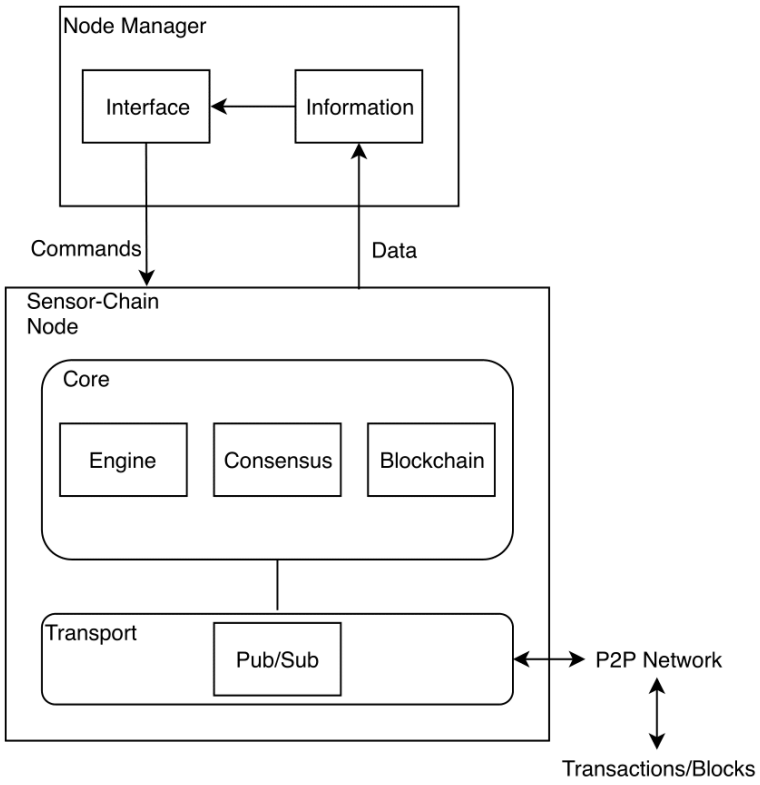

Figure 9. Architecture Diagram of Sensor-Chain.

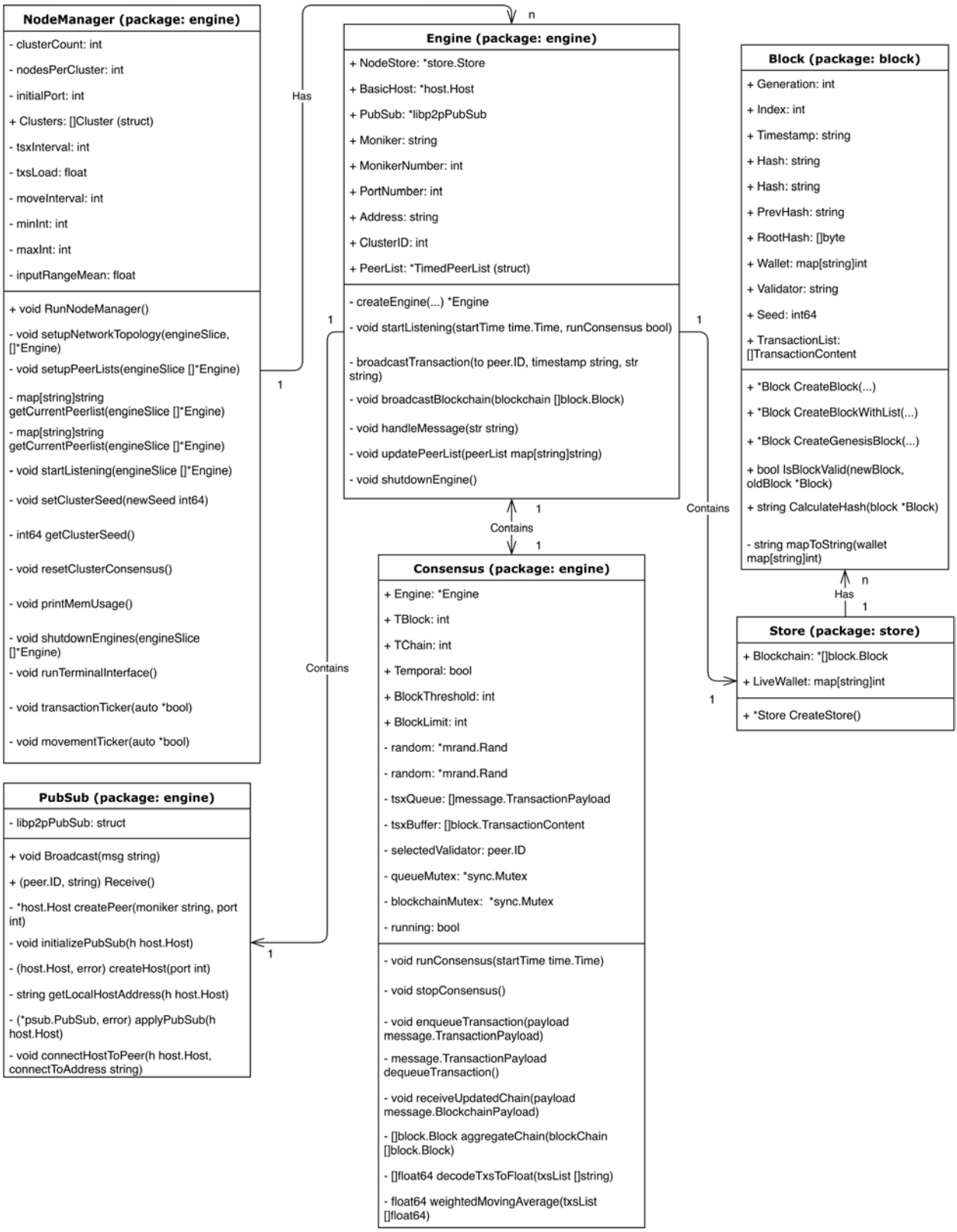

Figure 10. Class Diagram of Sensor-Chain.

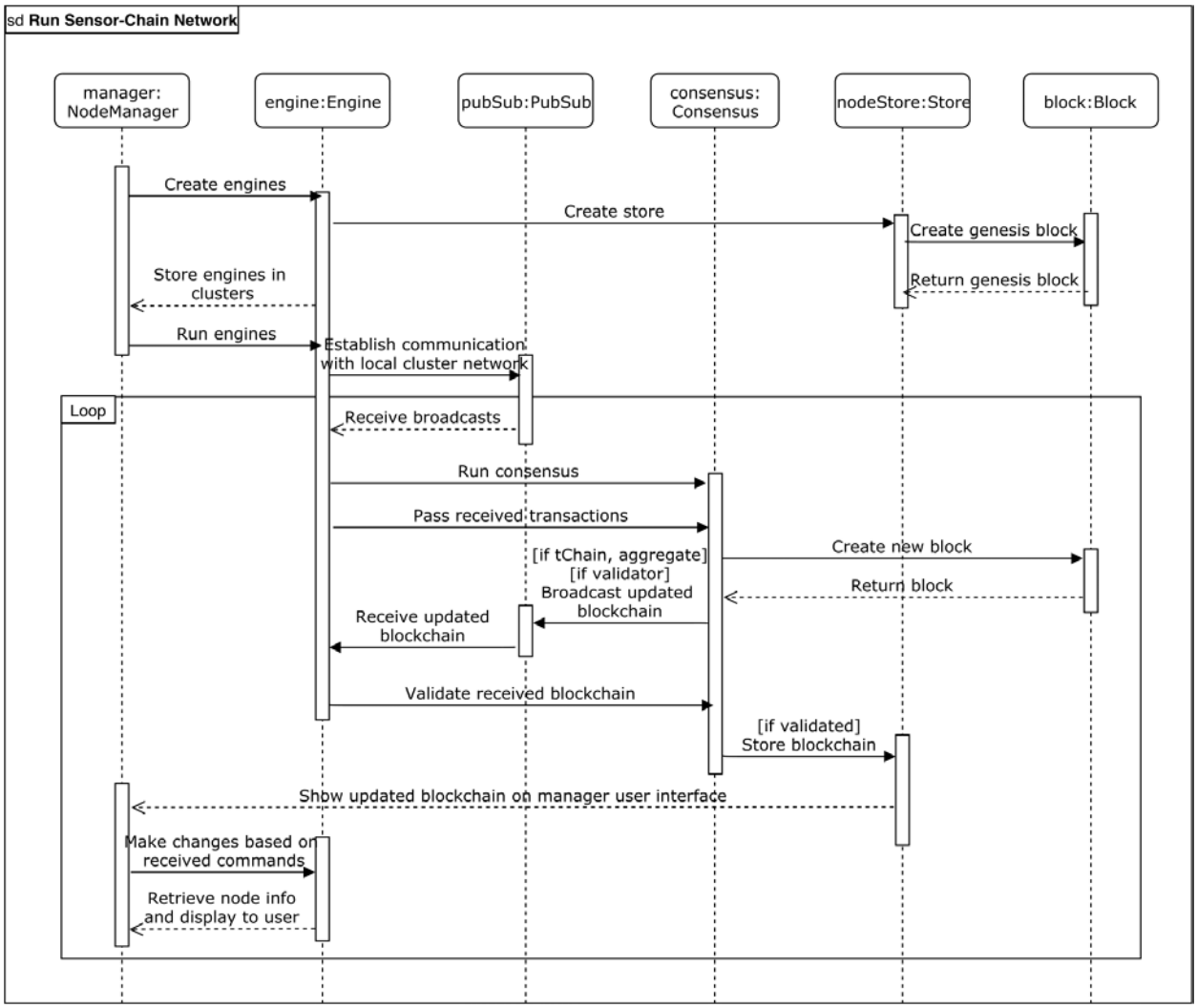

Figure 11. Sequence Diagram of Sensor-Chain.

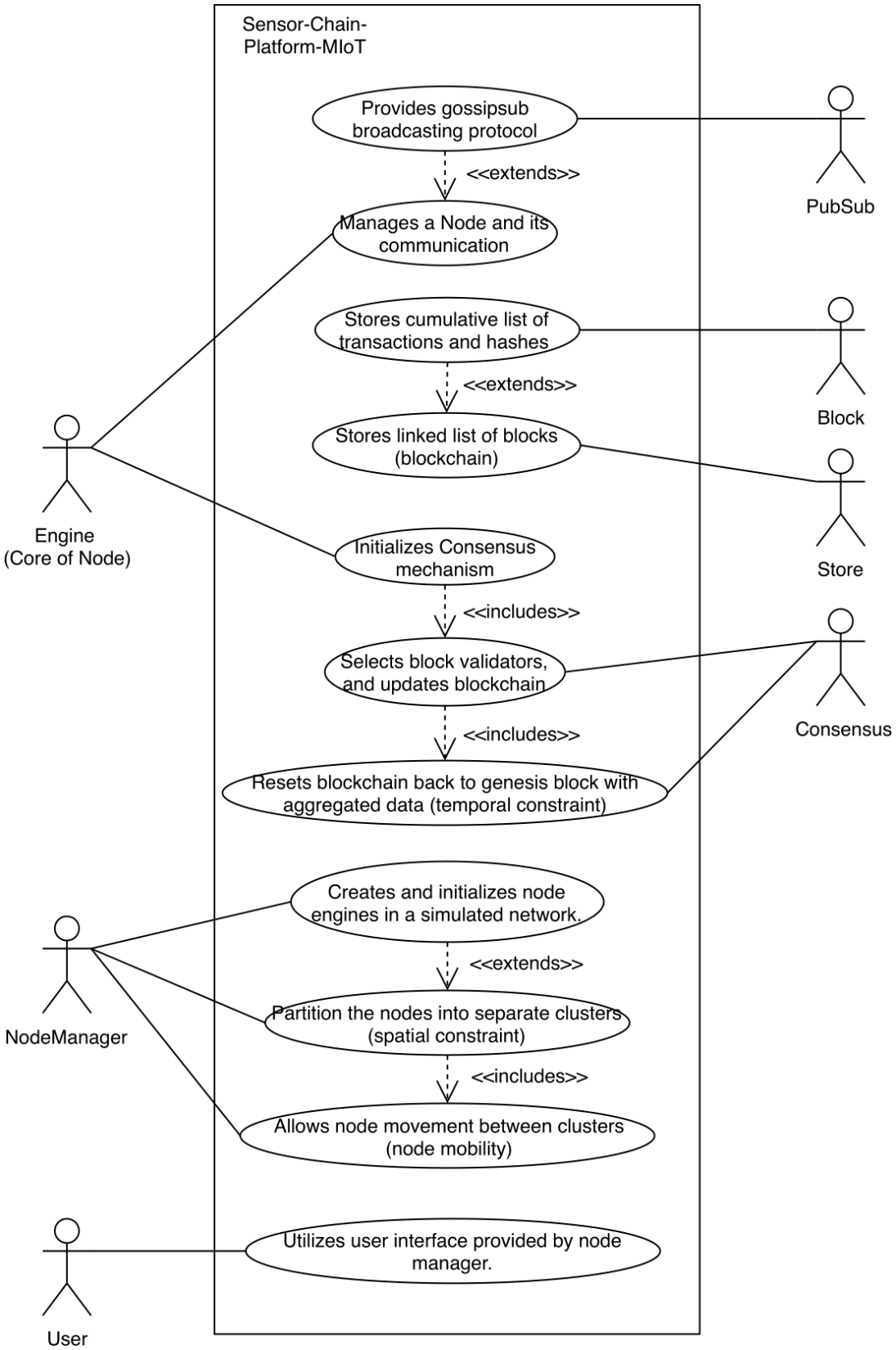

Figure 12. Use-case diagram.

5.0 CONCLUSION

In this report, we proposed “Sensor-Chain”, a lightweight scalable blockchain framework for resource-constrained IoT sensor devices. In this framework, a conventional blockchain is made lightweight in three steps. First, a global blockchain is divided into smaller disjoint local blockchains in spatial domain such that the required storage space to hold a local blockchain for an IoT device is always smaller than that in conventional blockchain. Second, a temporal constraint is imposed on the life span of the local blockchains to limit their size in temporal domain. Finally, a sensor node is required to keep at most one local blockchain in its memory at any time instance. We analyzed and tested Sensor-Chain in terms of both long-run performance and scalability; and compared with other approaches. Experimental results show that it consumes far little storage space than other approaches. Further, we implemented the demonstration of the framework for a P2P network of permissioned blockchain.

6.0 REFERENCES

1. Ashton, K. (2009). That 'internet of things' thing. *RFID journal*, *22*(7), 97-114.
2. Kott, A., Swami, A., & West, B. J. (2016). The internet of battle things. *Computer*, *49*(12), 70-75.
3. Castiglione, A., Choo, K. K. R., Nappi, M., & Ricciardi, S. (2017). Context aware ubiquitous biometrics in edge of military things. *IEEE Cloud Computing*, *4*(6), 16-20.
4. Fernández-Caramés, T. M., & Fraga-Lamas, P. (2018). A Review on the Use of Blockchain for the Internet of Things. IEEE Access, 6, 32979-33001.
5. S. Nakamoto, Bitcoin: A peer-to-peer electronic cash system, bitcoin.org, 2008
6. P. Franco,Understanding Bitcoin: Cryptography, engineering and economics. John Wiley & Sons, 2014.
7. K. Christidis and M. Devetsikiotis, "Blockchains and smart contracts for the internet of things," Ieee Access, vol. 4, pp. 2292–2303, 2016.
8. B. Yu, J. Wright, S. Nepal, L. Zhu, J. Liu, and R. Ranjan, "Iotchain: Establishing trust in the internet of things ecosystem using blockchain," IEEE Cloud Computing, vol. 5, no. 4, pp. 12–23, Jul./Aug. 2018. [Online]. Available: doi.ieeecomputersociety.org/10.1109/MCC.2018.043221010
9. Z. Xiong, Y. Zhang, D. Niyato, P. Wang, and Z. Han, "When mobile blockchain meets edge computing: challenges and applications," arXiv preprint arXiv:1711.05938, 2017.
10. J. Pan, J. Wang, A. Hester, I. Alqerm, Y. Liu, and Y. Zhao, "Edgechain: An edge-iot framework and prototype based on blockchain and smart contracts," arXiv preprint arXiv:1806.06185, 2018.
11. Z. Xiong, S. Feng, W. Wang, D. Niyato, P. Wang, and Z. Han, "Cloud/fog computing resource management and pricing for blockchain networks," IEEE Internet of Things Journal, pp. 1–1, 2018.
12. D. W¨orner and T. von Bomhard, "When your sensor earns money: exchanging data for cash with bitcoin," in Proceedings of the 2014 ACM International Joint Conference on Pervasive and Ubiquitous Computing: Adjunct Publication. ACM, 2014, pp. 295–298.
13. O. Novo, "Blockchain meets iot: An architecture for scalable access management in iot," IEEE Internet of Things Journal, vol. 5, no. 2, pp. 1184–1195, April 2018.
14. A. Z. Ourad, B. Belgacem, and K. Salah, "Using blockchain for iot access control and authentication management," in Internet of Things – ICIOT 2018. Cham: Springer International Publishing, 2018, pp. 150–164.
15. G. Zyskind, O. Nathan et al., "Decentralizing privacy: Using blockchain to protect personal data," in Security and Privacy Workshops (SPW), 2015 IEEE. IEEE, 2015, pp. 180–184.
16. L. Axon, "Privacy-awareness in blockchain-based pki," 2015.
17. A. Moinet, B. Darties, and J.-L. Baril, "Blockchain based trust & authentication for decentralized sensor networks," arXiv preprint arXiv:1706.01730, 2017.
18. A. Durand, P. Gremaud, and J. Pasquier, "Decentralized web of trust and authentication for the internet of things," in Proceedings of the Seventh International Conference on the Internet of Things, ser. IoT '17. New York, NY, USA: ACM, 2017, pp. 27:1–27:2. [Online]. Available: http://doi.acm.org/10.1145/3131542.3140263

19. G. Ayoade, V. Karande, L. Khan, and K. Hamlen, “Decentralized iot data management using blockchain and trusted execution environment,” in 2018 IEEE International Conference on Information Reuse and Integration (IRI), July 2018, pp. 15–22.
20. P. Angin, M. B. Mert, O. Mete, A. Ramazanli, K. Sarica, and B. Gungoren, “A blockchain-based decentralized security architecture for iot,” in Internet of Things – ICIOT 2018. Cham: Springer International Publishing, 2018, pp. 3–18.
21. R. Casado-Vara, J. Prieto, and J. M. Corchado, “How blockchain could improve fraud detection in power distribution grid,” in International Joint Conference SOCO’18-CISIS’18-ICEUTE’18. Cham: Springer International Publishing, 2019, pp. 67–76.
22. A. Dorri, S. S. Kanhere, R. Jurdak, and P. Gauravaram, “Blockchain for iot security and privacy: The case study of a smart home,” in Pervasive Computing and Communications Workshops (PerCom Workshops), 2017 IEEE International Conference on. IEEE, 2017, pp. 618–623.
23. P. K. Sharma, S. Y. Moon, and J. H. Park, “Block-vn: A distributed blockchain based vehicular network architecture in smart city,” Journal of Information Processing Systems, vol. 13, no. 1, p. 84, 2017.
24. I. Foundation. (2018) Iota. [Online]. Available: https://www.iota.org/
25. R. A. Michelin, A. Dorri, R. C. Lunardi, M. Steger, S. S. Kanhere, R. Jurdak, and A. F. Zorzo, “Speedychain: A framework for decoupling data from blockchain for smart cities,” arXiv preprint arXiv:1807.01980, 2018.
26. Z. Yang, K. Yang, L. Lei, K. Zheng, and V. C. Leung, “Blockchain-based decentralized trust management in vehicular networks,” IEEE Internet of Things Journal, 2018.
27. Z. Xiong, S. Feng, D. Niyato, P. Wang, and Z. Han, “Edge computing resource management and pricing for mobile blockchain,” CoRR, vol. abs/1710.01567, 2017.
28. R. C. Lunardi, R. A. Michelin, C. V. Neu, and A. F. Zorzo, “Distributed access control on iot ledger-based architecture,” in NOMS 2018-2018 IEEE/IFIP Network Operations and Management Symposium. IEEE, 2018, pp. 1–7.
29. J. Kang, Z. Xiong, D. Niyato, D. Ye, D. I. Kim, and J. Zhao, “Towards secure blockchain-enabled internet of vehicles: Optimizing consensus management using reputation and contract theory,” arXiv preprint arXiv:1809.08387, 2018.
30. R. Dennis, G. Owenson, and B. Aziz, “A temporal blockchain: a formal analysis,” in Collaboration Technologies and Systems (CTS), 2016 International Conference on. IEEE, 2016, pp. 430–437.
31. X. Chen, K. Makki, K. Yen, and N. Pissinou, “Sensor network security: A survey.” IEEE Communications Surveys and Tutorials, vol. 11, no. 2, pp. 52–73, 2009.
32. T. Martin, M. Hsiao, D. Ha, and J. Krishnaswami, “Denial-of-service attacks on battery-powered mobile computers,” in Pervasive Computing and Communications, 2004. PerCom 2004. Proceedings of the Second IEEE Annual Conference on. IEEE, 2004, pp. 309–318.
33. J. C. Foster, V. Osipov, and N. Bhalla, Buffer overflow attacks. Syngress Publishing, 2005.
34. G. Liang, J. Zhao, F. Luo, S. R. Weller, and Z. Y. Dong, “A review of false data injection attacks against modern power systems,” IEEE Transactions on Smart Grid, vol. 8, no. 4, pp. 1630–1638, 2017.

35. S. Pumpichet, X. Jin, and N. Pissinou, “Sketch-based data recovery in sensor data streams,” in Networks (ICON), 2013 19th IEEE International Conference on. IEEE, 2013, pp. 1–6.
36. M. d. Berg, O. Cheong, M. v. Kreveld, and M. Overmars, Computational geometry: algorithms and applications. Springer-Verlag TELOS, 2008.
37. S. Tasnim, J. Caldas, N. Pissinou, S. Iyengar, and Z. Ding, “Semantic-aware clustering-based approach of trajectory data stream mining,” in 2018 International Conference on Computing, Networking and Communications (ICNC). IEEE, 2018, pp. 88–92.
38. S. Tasnim, N. Pissinou, and S. Iyengar, “A novel cleaning approach of environmental sensing data streams,” in Consumer Communications & Networking Conference (CCNC), 2017 14th IEEE Annual. IEEE, 2017, pp. 632–633.
39. C. S. Aleman, N. Pissinou, S. Alemany, and G. Kamhoua, “A dynamic trust weight allocation technique for data reconstruction in mobile wireless sensor networks,” in 2018 17th IEEE International Conference On Trust, Security And Privacy In Computing And Communications/12th IEEE International Conference On Big Data Science And Engineering (TrustCom/BigDataSE). IEEE, 2018, pp. 61–67.
40. B. Q. Ali, N. Pissinou, and K. Makki, “Belief based data cleaning for wireless sensor networks,” Wireless Communications and Mobile Computing, vol. 12, no. 5, pp. 406–419, 2012.
41. C. C. Aggarwal and S. Sathe, “Theoretical foundations and algorithms for outlier ensembles,” ACM SIGKDD Explorations Newsletter, vol. 17, no. 1, pp. 24–47, 2015.
42. S. Ramaswamy, R. Rastogi, and K. Shim, “Efficient algorithms for mining outliers from large data sets,” in ACM Sigmod Record, vol. 29, no. 2. ACM, 2000, pp. 427–438.
43. Hyperledger. (2018) Hyperledger sawtooth. [Online]. Available: https://www.hyperledger.org/projects/sawtooth
44. Sawtooth. (2018) Private networks with the sawtooth permissioning features. [Online]. Available: https://sawtooth.hyperledger.org/docs/core/releases/latest/introduction.html#about-distributed-ledgers
45. E. Cho, S. A. Myers, and J. Leskovec, “Friendship and mobility: user movement in location-based social networks,” in Proceedings of the 17th ACM SIGKDD international conference on Knowledge discovery and data mining. ACM, 2011, pp. 1082–1090.
46. D. Yang, D. Zhang, Z. Yu, and Z. Yu, “Fine-grained preference-aware location search leveraging crowdsourced digital footprints from lbsns,” in Proceedings of the 2013 ACM international joint conference on Pervasive and ubiquitous computing. ACM, 2013, pp. 479–488.

APPENDIX A – Publications and Presentations

1. Shahid, A. R., Pissinou, N., Staier, C., & Kwan, R. (2019, July). Sensor-Chain: A Lightweight Scalable Blockchain Framework for Internet of Things. In 2019 International Conference on Internet of Things (iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData) (pp. 1154-1161). IEEE.

2. Shahid, A.R., Pissinou, N., Njilla, L., Aguilar. E., & Perez. E. (2019, November). Demo: Towards the Development of a Differentially Private Lightweight and Scalable Blockchain for IoT. In 16th IEEE International Conference on Mobile Ad-Hoc and Smart Systems (MASS), IEEE.

3. Shahid, A.R., Pissinou, N., Njilla, L., Alemany, S., Imteaj, A., Makki, K., & Aguilar, E. (2019, November). Quantifying Location Privacy in Permissioned Blockchain-Based Internet of Things (IoT). In 2019 16th EAI International Conference on Mobile and Ubiquitous Systems: Computing, Networking and Services (MobiQuitous), EAI.

APPENDIX B – Source codes

node-manager.go

```
package engine


import (
                        “bufio”
                        “encoding/json”
                        “fmt”
                        “log”
                        mrand “math/rand”
                        “os”
                        “runtime”
                        “strconv”
                        “strings”
                        “sync”
                        “time”


                        “github.com/EAGnR/sensor-chain/src/block”
)


var (
                        clusterCount    = 4     // The amount of clusters, with more clusters then more nodes can be created without the risk of connection pruning.
                        nodesPerCluster = 6     // The starting amount of nodes per cluster, note that if this number is way too high then excessive connections may get pruned by libp2p.
                        initialPort     = 10000 // The port of the first node created, it is incremented for each node, this is necessary as the node manager runs them on localhost.


                        // Clusters can be used to access all the running nodes within their clusters.
                        Clusters = make([]Cluster, clusterCount)


                        tsxInterval = 5000 // Transactions attempted every tsxInterval by node, in ms.
                        tsxLoad     = 0.2  // Probability that each transaction attempt will go through, adds randomness, [0.0,1.0].
```

```
		moveInterval = tBlock * 2 // Each moveInterval a random amount of nodes will move to another cluster, being too low may cause stalling of cluster consensus.

		minInt = 0    // The minimum value for the transaction generation.
		maxInt = 1000 // The maximum value ...

		inputRangeMean = (float64(maxInt-minInt) / 2.0) + float64(minInt) // Middle of range

		consensusDebug   = false  // Debug flag for debugging consensus, right now it would show validator selection for the current cluster.
		currClusterDebug *Cluster // USed for validator selection dubugging.
)

// Cluster stores the nodes belonging to a certain cluster in the network.
type Cluster struct {
		ClusterID int       // Unique identifier of the cluster.
		Engines   []*Engine // Nodes belonging to the cluster.

		joinQueue []*Engine   // Nodes waiting to join the cluster.
		seed      int64       // This seed value can be used for consensus purposes, where all nodes can share a seed for a random number generator.
		resetting bool        // Flag that signifies if the consensus for this cluster is being reset.
		seedLock  *sync.Mutex // Locks the seed whenever a validator changes it for the cluster, or any node reads it.
}

// RunNodeManager initializes the node creation and management process.
func RunNodeManager() {
		counter := 0

		for i := range Clusters {
			Clusters[i] = Cluster{ClusterID: i, Engines: []*Engine{}, seed: int64(i), resetting: false, seedLock: &sync.Mutex{}}
			Clusters[i].Engines = make([]*Engine, nodesPerCluster)
		}
```

```
                for _, cluster := range Clusters {
                        engines := cluster.Engines

                        defer shutdownEngines(engines)

                        for i := range engines {
                                engines[i] = createEngine(fmt.Sprintf("node %d", counter), counter, cluster.ClusterID,
initialPort+counter)
                                counter++
                        }

                        setupNetworkTopology(engines)
                        setupPeerLists(engines)
                        startListening(engines)
                }

                runTerminalInterface()
}

// setupNetworkTopology sets up a random and sparse network topology.
func setupNetworkTopology(engineSlice []*Engine) {
                mrand.Seed(time.Now().UTC().UnixNano())

                if len(engineSlice) > 1 {
                        edges := "Graph Topology: {"
                        for i := range engineSlice {
                                n := i

                                for n == i || (len(engineSlice) > 2 &&
len((*engineSlice[i].BasicHost).Network().ConnsToPeer((*engineSlice[n].BasicHost).ID())) != 0) {
                                        n = mrand.Intn(len(engineSlice))
                                }

                                connectHostToPeer(*engineSlice[i].BasicHost,
getLocalHostAddress(*engineSlice[n].BasicHost))
```

```
                        edges += fmt.Sprintf("(%d, %d), ", i, n)
                    }

                    edges = strings.TrimSuffix(edges, ", ")
                    edges += "}"

                    fmt.Println(edges)
                }

                // Wait so that subscriptions on topic will be done and all peers will "know" of all other peers
                time.Sleep(time.Second * 2)
}

func setupPeerLists(engineSlice []*Engine) {
                peerList := getCurrentPeerlist(engineSlice)

                for i := range engineSlice {
                    engineSlice[i].updatePeerList(peerList)
                }
}

func getCurrentPeerlist(engineSlice []*Engine) map[string]string {
                peerList := make(map[string]string)

                for _, e := range engineSlice {
                    peerList[(*e.BasicHost).ID().Pretty()] = getLocalHostAddress(*e.BasicHost)
                }

                return peerList
}

func startListening(engineSlice []*Engine) {
                startTime := time.Now()
```

```
                for i := range engineSlice {
                    go engineSlice[i].startListening(startTime, true)
                }
}
func (c *Cluster) setClusterSeed(newSeed int64) {
                c.seedLock.Lock()
                c.seed = newSeed
                c.seedLock.Unlock()
}

func (c *Cluster) getClusterSeed() int64 {
                c.seedLock.Lock()
                currSeed := c.seed
                c.seedLock.Unlock()

                return currSeed
}

func (c *Cluster) resetClusterConsensus() {
                if !c.resetting {
                    c.resetting = true

                    for i := range c.Engines {
                            c.Engines[i].Consensus.stopConsensus()
                    }

                    // Waiting for current consensus round to finalize, before proceeding with reset.
                    time.Sleep(time.Second * time.Duration(tBlock))

                    // synchronizing cluster seed.
                    for i := range c.Engines {
                            c.Engines[i].Consensus.selectedValidator = ""
                            c.Engines[i].Consensus.random.Seed(c.getClusterSeed())
                    }
```

```
                startTime := time.Now()

                // restarting consensus
                for i := range c.Engines {
                        go c.Engines[i].Consensus.runConsensus(startTime)
                }

                fmt.Printf("\nConsensus reset and synchronized for cluster %d.\n", c.ClusterID)

                c.resetting = false
        }
}

// PrintMemUsage outputs the current, total and OS memory being used. As well as the number
// of garage collection cycles completed.
func printMemUsage() {
        var m runtime.MemStats
        runtime.ReadMemStats(&m)
        // For info on each, see: https://golang.org/pkg/runtime/#MemStats
        // Green console color:  \x1b[32m
        // Reset console color:  \x1b[0m
        fmt.Print("\x1b[32m")
        fmt.Println("\nOverall Memory Usage Stats:")
        fmt.Printf("Current allocated memory = %v KiB\n", bToKb(m.Alloc))
        fmt.Printf("Total memory allocated over time = %v KiB\n", bToKb(m.TotalAlloc))
        fmt.Printf("System memory obtained from OS = %v KiB\n", bToKb(m.Sys))
        fmt.Printf("Garbage Collector cycles ran = %v\n", m.NumGC)
        fmt.Print("\x1b[0m")
}

func bToMb(b uint64) uint64 {
        return b / 1024 / 1024
}
```

```
func bToKb(b uint64) uint64 {
                        return b / 1024
}

func shutdownEngines(engineSlice []*Engine) {
                        fmt.Println("Shutting down node engines.")
                        for _, e := range engineSlice {
                            e.shutdownEngine()
                        }
}

func runTerminalInterface() {
                        stdReader := bufio.NewReader(os.Stdin)
                        switchClusterCommand := "SwitchCluster"
                        switchNodeCommand := "SwitchNode"
                        sendTransactionCommand := "Send"
                        autoTransactionsCommand := "AutoSend"
                        manualTransactionsCommand := "ManualSend"
                        showActivityCommand := "ShowActivity"
                        hideActivityCommand := "HideActivity"
                        printBlockChainCommand := "PrintBlockchain"
                        autoMoveOnCommand := "AutoMoveOn"
                        autoMoveOffCommand := "AutoMoveOff"
                        resetClusterConsensusCommand := "ResetConsensus"
                        shutdownNetworkCommand := "Shutdown"
                        currNode := Clusters[0].Engines[0] // Starting node
                        currNode.Verbose = true
                        verbosity := currNode.Verbose
                        currCluster := &Clusters[0]
                        autoSend := false
                        autoMove := false
                        currClusterDebug = currCluster
                        for {
```

```
currNode.Verbose = false
fmt.Println()
fmt.Printf("Current cluster, ID: %d, node count: %d\n", currCluster.ClusterID,
len(currCluster.Engines))
nodes := "Nodes in cluster: "
for i := range currCluster.Engines {
        nodes += fmt.Sprintf("([%d]: %s), ", i, currCluster.Engines[i].Moniker)
}
nodes = strings.TrimSuffix(nodes, ", ")
fmt.Println(nodes)

fmt.Printf("Current node, moniker: %s, address: %s\n", currNode.Moniker, currNode.Address)

fmt.Printf("\nCommands: %s, %s, %s, %s, %s, %s, %s, %s, %s, %s, %s, %s\n",
switchClusterCommand, switchNodeCommand, sendTransactionCommand,
        autoTransactionsCommand, manualTransactionsCommand, showActivityCommand,
hideActivityCommand, printBlockChainCommand, autoMoveOnCommand,
        autoMoveOffCommand, resetClusterConsensusCommand, shutdownNetworkCommand)
fmt.Print("Enter command: ")
input, err := stdReader.ReadString('\n')
if err != nil {
        log.Fatal(err)
}
input = strings.Replace(input, "\n", "", -1)
currNode.Verbose = verbosity
switch strings.ToLower(input) {
case strings.ToLower(switchClusterCommand):
        clusterNumber := 0
        currNode.Verbose = false
        fmt.Printf("Enter cluster ID [0-%d]: ", len(Clusters)-1)
        _, err := fmt.Scanf("%d", &clusterNumber)
        if err != nil {
                fmt.Println("Invalid cluster ID!")
                break
        }
```

```
            if clusterNumber >= 0 && clusterNumber < len(Clusters) {
                        currCluster = &Clusters[clusterNumber]
                        currClusterDebug = currCluster
                        currNode = currCluster.Engines[0]
            } else {
                        fmt.Println("Invalid cluster ID range!")
                        break
            }
            currNode.Verbose = verbosity
case strings.ToLower(switchNodeCommand):
            nodeNumber := 0
            currNode.Verbose = false
            fmt.Printf("Enter node index [0-%d]: ", len(currCluster.Engines)-1)
            _, err := fmt.Scanf("%d", &nodeNumber)
            if err != nil {
                        fmt.Println("Invalid node index!")
                        break
            }
            if nodeNumber >= 0 && nodeNumber < len(currCluster.Engines) {
                        currNode = currCluster.Engines[nodeNumber]
            } else {
                        fmt.Println("Invalid node index range!")
                        break
            }
            currNode.Verbose = verbosity

case strings.ToLower(sendTransactionCommand):
            message := ""
            nodeNumber := 0
            currNode.Verbose = false
            fmt.Print("Enter transaction message: ")
            message, err := stdReader.ReadString('\n')
            if err != nil {
```

```
                fmt.Println("Invalid message!")
                break
            }
            message = strings.Replace(message, "\n", "", -1)

            fmt.Printf("Enter receiver node number [0-%d]: ", len(currCluster.Engines)-1)
            _, err = fmt.Scanf("%d", &nodeNumber)
            if err != nil {
                fmt.Println("Invalid node number!")
                break
            }

            if nodeNumber >= 0 && nodeNumber < len(currCluster.Engines) {
                currNode.broadcastTransaction((*currCluster.Engines[nodeNumber].BasicHost).ID(),
time.Now().Format(time.RFC1123), message)
            } else {
                fmt.Println("Invalid node range!")
                break
            }

            currNode.Verbose = verbosity

        case strings.ToLower(autoTransactionsCommand):
            if !autoSend {
                autoSend = true
                go transactionTicker(&autoSend)
            }

        case strings.ToLower(manualTransactionsCommand):
            autoSend = false

        case strings.ToLower(showActivityCommand):
            verbosity = true
            currNode.Verbose = verbosity
```

```
case strings.ToLower(hideActivityCommand):
        verbosity = false
        currNode.Verbose = verbosity

case strings.ToLower(printBlockChainCommand):
        bytes, err := json.MarshalIndent(*currNode.NodeStore.Blockchain, "", "  ")
        if err != nil {
                log.Fatal(err)
        }
        // Green console color:         \x1b[32m
        // Reset console color:         \x1b[0m
        fmt.Printf("\x1b[32m%s\x1b[0m\n", string(bytes))
        printMemUsage()

case strings.ToLower(autoMoveOnCommand):
        if !autoMove {
                autoMove = true
                go movementTicker(&autoMove)
        }

case strings.ToLower(autoMoveOffCommand):
        autoMove = false

case strings.ToLower(resetClusterConsensusCommand):
        go currCluster.resetClusterConsensus()

case strings.ToLower(shutdownNetworkCommand):
        return

default:
        fmt.Println("Invalid command!")
}
```

```
                stdReader.ReadLine()
            }
}

// transactionTicker can be called as a Goroutine, asynchronously submits node transactions based on ticker time interval, and generated from a nomral dist.
func transactionTicker(auto *bool) {
            mrand.Seed(time.Now().UTC().UnixNano())

            for *auto {
                for _, cluster := range Clusters {
                    engines := cluster.Engines

                    for _, engine := range engines {
                        value := mrand.NormFloat64()*(inputRangeMean/3.0) + inputRangeMean // Alters standard normal dist. to correspond to value range.
                        message := fmt.Sprintf("%s", strconv.FormatFloat(value, 'f', 3, 64))
                        destNodeIndex := mrand.Intn(len(engines))

                        if mrand.Float64() <= tsxLoad {
            engine.broadcastTransaction((*engines[destNodeIndex].BasicHost).ID(), time.Now().Format(time.RFC1123), message)
                        }

                        if !*auto {
                            break
                        }
                    }
                }

                time.Sleep(time.Millisecond * time.Duration(tsxInterval))
            }
            fmt.Println("Stopped automatic transaction generation.")
}
```

```
// TODO: Seems to work okay, but needs a lot more testing to catch all possible edge cases, especially in preventing fork conditions.

//*****

// movementTicker can be called as a Goroutine, asynchronously moves random nodes to different clusters.

func movementTicker(auto *bool) {
                        mrand.Seed(time.Now().UTC().UnixNano())


                        for *auto {
                            if len(Clusters) < 2 {
                                    *auto = false
                                    fmt.Println("Automatic node movement cannot be activated with only 1 cluster.")
                                    break
                            }


                            srcClusterID := mrand.Intn(len(Clusters))
                            destClusterID := srcClusterID


                            for destClusterID == srcClusterID {
                                    destClusterID = mrand.Intn(len(Clusters))
                            }


                            srcCluster := &Clusters[srcClusterID]
                            destCluster := &Clusters[destClusterID]


                            movingNodeIndex := mrand.Intn(len(srcCluster.Engines))
                            movingNode := srcCluster.Engines[movingNodeIndex]


                            if len(srcCluster.Engines) > 1 {
                                    // Removing node from source cluster engine list.
                                    if movingNodeIndex < len(srcCluster.Engines)-1 {
                                            srcCluster.Engines = append(srcCluster.Engines[:movingNodeIndex], srcCluster.Engines[movingNodeIndex+1:]...)
                                    } else {
                                            srcCluster.Engines = srcCluster.Engines[:movingNodeIndex]
```

```
}

// Updating peer list for nodes in source cluster.
setupPeerLists(srcCluster.Engines)

// Disconnecting node from current cluster, and emptying its storage.
movingNode.Consensus.stopConsensus()

for _, conn := range (*movingNode.BasicHost).Network().Conns() {
        conn.Close() // Closes all connections from source cluster to this node.
}

// In case some nodes become disconnected as this one leaves, reconnect them to the source cluster again.
// ISSUE: Partial solution, helps mitigate the problem, but doesn't entirely solve it. There is the possibility of a new disconnected graph forming.
// A possible solution would be disconnect all nodes, and reconnect them again using setupNetworkTopology(), but this is a brute force solution.
// TODO: Come up with a complete and efficient solution.
if len(srcCluster.Engines) > 1 {
        for i, engine := range srcCluster.Engines {

                // If a node in srcCluster is found to have 0 connections then reconnect it.
                if len((*engine.BasicHost).Network().Conns()) == 0 {
                        n := i

                        for n == i || (len(srcCluster.Engines) > 2 &&
                len((*srcCluster.Engines[i].BasicHost).Network().ConnsToPeer((*srcCluster.Engines[n].BasicHost).ID())) != 0) {

                                n = mrand.Intn(len(srcCluster.Engines))
                        }

                        connectHostToPeer((*engine.BasicHost), getLocalHostAddress((*srcCluster.Engines[n].BasicHost)))
```

```
                                        }
                                }
                        }

                        movingNode.ClusterID = destClusterID
                        movingNode.PeerList = &TimedPeerList{time.Now(), make(map[string]string)}
                        movingNode.NodeStore.Blockchain = &[]block.Block{}
                        movingNode.NodeStore.LiveWallet = make(map[string]int)
                        movingNode.Consensus = createConsensus(movingNode, tBlock, tChain, temporal,
blockThreshold, blockLimit)

                        // Updating join queue of destination cluster.
                        destCluster.joinQueue = append(destCluster.joinQueue, movingNode)
                }

                // Adding nodes waiting to join destination clusters to network.
                for len(destCluster.joinQueue) > 0 {
                        joiningNode := destCluster.joinQueue[0]

                        for i := range destCluster.Engines {
                                destCluster.Engines[i].Consensus.stopConsensus()
                        }

                        connectHostToPeer((*joiningNode.BasicHost),
getLocalHostAddress((*destCluster.Engines[mrand.Intn(len(destCluster.Engines))].BasicHost)))

                        destCluster.Engines = append(destCluster.Engines, joiningNode)

                        setupPeerLists(destCluster.Engines)

                        // Dequeueing join queue.
                        if len(destCluster.joinQueue) > 1 {
                                destCluster.joinQueue = destCluster.joinQueue[1:]
                        } else {
                                destCluster.joinQueue = []*Engine{}
```

```
                        }


                        go destCluster.resetClusterConsensus()
                    }


                    // For info on each, see: https://golang.org/pkg/runtime/#MemStats
                    // Green console color:     \x1b[32m
                    // Reset console color:     \x1b[0m
                    fmt.Print("\x1b[32m")
                    fmt.Printf("\n%s moving from cluster %d to cluster %d\n", movingNode.Moniker,
srcCluster.ClusterID, destCluster.ClusterID)
                    fmt.Print("\x1b[0m")
                    time.Sleep(time.Second * time.Duration(moveInterval))
                }
                fmt.Println("Stopped automatic node movement.")
}
```

engine.go

```
package engine

import (
                "encoding/json"
                "fmt"
                "log"
                "time"

                "github.com/EAGnR/sensor-chain/src/block"
                msg "github.com/EAGnR/sensor-chain/src/message"
                "github.com/EAGnR/sensor-chain/src/store"

                "github.com/libp2p/go-libp2p-core/host"
                "github.com/libp2p/go-libp2p-core/peer"
)
```

```
// 30 seconds tBlock is suggested for up to 10 nodes per cluster, for more nodes per cluster consider raising the tBlock for longer broadcasting times.

var (

		tBlock 	= 30 		// In seconds, should allow enough time for broadcasting over the network, otherwise more forking is likely to occur.

		tChain 	= 240 		// In seconds, can be used to determine the maximum amount of blocks based on tBlock.

		temporal 	= true 		// Whether the temporal constraint is activated or not.

		blockThreshold = 8 		// Minimum amount of blocks needed before aggregating.

		blockLimit 	= blockThreshold + 2 // Minimum amount of blocks needed before aggregating.

)


// Engine manages the host of a node and its communication over the network, essentially it is the core of a node.

type Engine struct {

		NodeStore 	*store.Store 	// This node's storage.

		BasicHost 	*host.Host 	// The engine's host for communication over the P2P network.

		PubSub 	*libp2pPubSub 	// The broadcasting mechanism used.

		Moniker 	string 	// Unique nickname.

		MonikerNumber int 	// Numerical portion of nickname.

		PortNumber 	int 	// Listen Port Number

		Address 	string 	// The address used to connected with this node.

		ClusterID 	int 	// Identifying ID of the cluster this node belongs to.

		PeerList 	*TimedPeerList // The list of peers belonging to this node's current cluster.

		Consensus 	*Consensus 	// Consensus mechanism used by this engine.

		Verbose 	bool 	// Whether this engine is currently selected by the Node Manager to output to stdout.


		shutdown bool // The engine shuts down when this flag is set.

}


// TimedPeerList is the struct type that stores a node's peer list alongside its update time.

type TimedPeerList struct {
```

```
                Timestamp time.Time
                Peers     map[string]string
}

// createEngine creates a new engine thereby running a new node.
func createEngine(moniker string, monikerNumber, clusterID int, listenPort int) *Engine {
                pubsub := new(libp2pPubSub)
                // creating libp2p host
                host := pubsub.createPeer(moniker, listenPort)
                // creating pubsub
                pubsub.initializePubSub(*host)

                engine := Engine{
                    NodeStore:     store.CreateStore(),
                    BasicHost:     host,
                    PubSub:        pubsub,
                    Moniker:       moniker,
                    MonikerNumber: monikerNumber,
                    PortNumber:    listenPort,
                    Address:       getLocalHostAddress(*host),
                    ClusterID:     clusterID,
                    PeerList:      &TimedPeerList{time.Now(), make(map[string]string)},
                    Consensus:     nil,
                    Verbose:       false,

                    shutdown: false,
                }

                engine.Consensus = createConsensus(&engine, tBlock, tChain, temporal, blockThreshold,
blockLimit)

                return &engine
```

```
}

func (e *Engine) startListening(startTime time.Time, runConsensus bool) {
                        if runConsensus {
                            e.Consensus.runConsensus(startTime)
                        }

                        for !e.shutdown {
                            sender, message := e.PubSub.Receive()

                            if e.Verbose {
                                    fmt.Println("\nIncoming broadcast...")

                                    data := &msg.Message{}
                                    if err := json.Unmarshal([]byte(message), &data); err != nil {
                                            log.Fatal(err)
                                    }

                                    fmt.Printf("Node %s sent Message of type: '%s'\n", sender.Pretty(), data.Type)

                                    if data.Type == msg.TransactionType {
                                            fmt.Println(message)
                                    }
                            }

                            e.handleMessage(message)
                        }

                        e.Consensus.stopConsensus()

                        err := (*e.BasicHost).Close()
                        if err != nil {
```

```
                log.Println(err)
            }
}

func (e *Engine) broadcastTransaction(to peer.ID, timestamp string, str string) {
            from := (*e.BasicHost).ID()

            rawData, err := json.Marshal(msg.TransactionPayload{From: from, To: to, Timestamp:
timestamp, Transaction: str})
            if err != nil {
                log.Fatal(err)
            }

            message := msg.Message{Type: msg.TransactionType, RawPayload: rawData}

            bytes, err := json.Marshal(message)
            if err != nil {
                log.Println(err)
            }

            e.PubSub.Broadcast(string(bytes))
}

func (e *Engine) broadcastBlockchain(blockchain []block.Block) {
            rawData, err := json.Marshal(msg.BlockchainPayload{Blockchain: blockchain})
            if err != nil {
                log.Fatal(err)
            }

            message := msg.Message{Type: msg.BlockchainType, RawPayload: rawData}

            bytes, err := json.Marshal(message)
```

```
                if err != nil {
                    log.Println(err)
                }

                e.PubSub.Broadcast(string(bytes))
}

func (e *Engine) handleMessage(str string) {
                if str == "" {
                    return
                }
                if str != "\n" {
                    message := &msg.Message{}
                    if err := json.Unmarshal([]byte(str), &message); err != nil {
                            log.Fatal(err)
                    }

                    switch message.Type {
                    case msg.TransactionType:
                            payload := msg.TransactionPayload{}
                            if err := json.Unmarshal(message.RawPayload, &payload); err != nil {
                                log.Fatal(err)
                            }

                            e.Consensus.enqueueTransaction(payload)

                    case msg.BlockchainType:
                            payload := msg.BlockchainPayload{}
                            if err := json.Unmarshal(message.RawPayload, &payload); err != nil {
                                log.Fatal(err)
                            }
```

```
                    e.Consensus.receiveUpdatedChain(payload)

                default:
                    if e.Verbose {
                        fmt.Println("Unknown message type received, it was discarded.")
                    }
                }
            }
}

func (e *Engine) updatePeerList(peerList map[string]string) {
            e.PeerList.Timestamp = time.Now()
            e.PeerList.Peers = peerList

            for key := range e.PeerList.Peers {
                if _, found := e.NodeStore.LiveWallet[key]; !found {
                    e.NodeStore.LiveWallet[key] = 0
                }
            }
}

func (e *Engine) shutdownEngine() {
            e.shutdown = true
}
```

Consensus.go

```
package engine

import (
            "encoding/json"
            "fmt"
            "log"
```

```
                mrand "math/rand"
                "runtime"
                "sort"
                "strconv"
                "sync"
                "time"

                "github.com/EAGnR/sensor-chain/src/block"
                "github.com/EAGnR/sensor-chain/src/message"
                "github.com/libp2p/go-libp2p-core/peer"
)

// Consensus is the struct which handles the consensus mechanism of the platform.
type Consensus struct {
                Engine        *Engine
                TBlock        int  // Time for block creation, in seconds.
                TChain        int  // Time for blockchain aggregation, in seconds.
                Temporal      bool // Whether the temporal constraint is activated or not.
                BlockThreshold int  // Minimum amount of blocks needed before aggregating.
                BlockLimit    int  // Maximum amount of blocks allowed.

                random          *mrand.Rand
                tsxQueue        []message.TransactionPayload // This servers as the transaction pool.
                tsxBuffer       []block.TransactionContent   // Transactions to be commited on next block.
                selectedValidator peer.ID
                queueMutex      *sync.Mutex
                blockchainMutex  *sync.Mutex
                running         bool // Whether the consensus mechanism is running or not.
}

// createConsensus creates a new consensus struct which manages the consensus mechanism.
```

```
func createConsensus(engine *Engine, tBlock int, tChain int, temporal bool, blockThreshold int, blockLimit int)
*Consensus {
                        consensus := &Consensus{
                            Engine:         engine,
                            TBlock:         tBlock,
                            TChain:         tChain,
                            Temporal:       temporal,
                            BlockThreshold: blockThreshold,
                            BlockLimit:     blockLimit,

                            random:           mrand.New(mrand.NewSource(0)),
                            tsxQueue:          []message.TransactionPayload{},
                            tsxBuffer:         []block.TransactionContent{},
                            selectedValidator: "",
                            queueMutex:        &sync.Mutex{},
                            blockchainMutex:   &sync.Mutex{},
                            running:          false,
                        }

                        return consensus
}

// runConsensus runs the randomized consensus mechanism for a node.
func (c *Consensus) runConsensus(startTime time.Time) {
                        c.running = true

                        c.blockchainMutex.Lock()
                        blockchain := c.Engine.NodeStore.Blockchain
                        c.blockchainMutex.Unlock()

                        // Broadcast blockchain on consensus restart if this is the previously elected validator.
                        go func() {
```

```
                c.blockchainMutex.Lock()
                if len(*blockchain) > 0 && (*blockchain)[len(*blockchain)-1].Header.Validator != "" {
                        recentValidator, err := peer.IDB58Decode((*blockchain)[len(*blockchain)-
1].Header.Validator)
                        if err != nil {
                                log.Fatal(err)
                        }

                        if recentValidator == (*c.Engine.BasicHost).ID() {
                                c.Engine.broadcastBlockchain(*blockchain)
                        }
                }
                c.blockchainMutex.Unlock()
        }()

        nextTBlockTime := startTime.Add(time.Second * time.Duration(c.TBlock/2)) // Initial
tBlock interval has a delay from start time.
        nextTChainTime := startTime.Add(time.Second * time.Duration(c.TChain))

        c.random.Seed(Clusters[c.Engine.ClusterID].getClusterSeed()) // Deterministic consensus
seed.

        go func() {
                for c.running {
                        if (time.Now().After(nextTBlockTime) || time.Now().Equal(nextTBlockTime))
&& len(c.tsxQueue) > 0 {
                                if len(*blockchain) < 1 {
                                        nextTBlockTime = nextTBlockTime.Add(time.Second *
time.Duration(c.TBlock))
                                        time.Sleep(time.Second)
                                        continue
                                }

                                // List of validator candidates is created and populated.
```

```
candidates := []string{}

for key := range c.Engine.PeerList.Peers {
        candidates = append(candidates, key)
}
sort.Strings(candidates)

// Validator is chosen radomly but in a deterministic manner for all
nodes.
validator := candidates[c.random.Intn(len(candidates))]

c.blockchainMutex.Lock()
lastBlock := (*blockchain)[len(*blockchain)-1]
c.blockchainMutex.Unlock()

lastBlockTime, err := time.Parse(time.RFC1123,
lastBlock.Header.Timestamp)
if err != nil {
        log.Fatal(err)
}

// Synchronizing the next consensus round seed
if nextTBlockTime.Before(lastBlockTime.Add(time.Second *
time.Duration(c.TBlock))) {

        c.random.Seed(Clusters[c.Engine.ClusterID].getClusterSeed()) // Deterministic consensus
seed.
        //nextTBlockTime = lastBlockTime
}

// Obtaining most recent previous validator.
c.blockchainMutex.Lock()
var recentValidator peer.ID
if (*blockchain)[len(*blockchain)-1].Header.Validator != "" {
```

```
                recentValidator, err =
peer.IDB58Decode((*blockchain)[len(*blockchain)-1].Header.Validator)
                if err != nil {
                        log.Fatal(err)
                }
        }
        c.blockchainMutex.Unlock()

        // If true then empty the transaction buffer, as the previous consensus
round was succesful,
        // otherwise keep the previous transactions to attempt committing them
again.
        if recentValidator == c.selectedValidator || c.selectedValidator == "" {
                c.tsxBuffer = []block.TransactionContent{}
        } else {
                if c.Engine.Verbose {
                        fmt.Println("Expected validator timed out, they could
have left the cluster, or there is congestion, or there is a possible fork.")
                }
        }

        pid, err := peer.IDB58Decode(validator)
        if err != nil {
                log.Fatal(err)
        }
        c.selectedValidator = pid
        if c.Engine.ClusterID == currClusterDebug.ClusterID &&
consensusDebug {
                fmt.Printf("Elected Validator of %s: %s\n",
c.Engine.Moniker, c.selectedValidator.Pretty())
        }

        // Dequeuing transaction queue into buffer, and updating node live
wallet.
```

```
c.queueMutex.Lock()
for len(c.tsxQueue) > 0 {
	t := c.dequeueTransaction()

	if _, found := c.Engine.NodeStore.LiveWallet[t.From.Pretty()]; found {
		c.Engine.NodeStore.LiveWallet[t.From.Pretty()]++
	} else {
		c.Engine.NodeStore.LiveWallet[t.From.Pretty()] = 1
	}

	c.tsxBuffer = append(c.tsxBuffer, block.TransactionContent{
		From: t.From.Pretty(), To: t.To.Pretty(), Timestamp: t.Timestamp, Transaction: t.Transaction})
}
c.queueMutex.Unlock()

// Broadcasting updated chain if chosen as validator.
go func() {
	c.blockchainMutex.Lock()
	if c.selectedValidator == (*c.Engine.BasicHost).ID() {
		seed := time.Now().UTC().UnixNano()
		candidateBlock := block.CreateBlockWithList(&lastBlock, c.tsxBuffer, c.Engine.NodeStore.LiveWallet, (*c.Engine.BasicHost).ID(), seed)
		currentChain := c.Engine.NodeStore.Blockchain

		// Temporal constraint aggregation.
		if c.Temporal && len(lastBlock.TransactionList) > 1 &&
			(((time.Now().After(nextTChainTime) || time.Now().Equal(nextTChainTime)) && len(*currentChain) >= c.BlockThreshold) ||
				len(*currentChain) >= c.BlockLimit) {
```

```
                                        if c.selectedValidator ==
(*c.Engine.BasicHost).ID() {
                                            aggregatedChain :=
c.aggregateChain(append(*currentChain, *candidateBlock))

                                            if aggregatedChain != nil {
                    c.Engine.broadcastBlockchain(aggregatedChain)
                                            }
                                        }

                                        nextTChainTime =
nextTChainTime.Add(time.Second * time.Duration(c.TChain))

                                        // Else append new block without
aggregating.
                                    } else {
                                        if block.IsBlockValid(candidateBlock,
&lastBlock) {
                    c.Engine.broadcastBlockchain(append(*currentChain, *candidateBlock))
                                        } else {
                                            if c.Engine.Verbose {
                                                fmt.Println("Block
Creation: Block validation failed!")
                                            }
                                        }
                                    }

                                    Clusters[c.Engine.ClusterID].setClusterSeed(seed)
                                }
                                c.blockchainMutex.Unlock()
                            }()
```

```
                                        nextTBlockTime = nextTBlockTime.Add(time.Second *
time.Duration(c.TBlock))
                                }
                                time.Sleep(time.Second)
                        }
                }()
}

// stopConsensus stops the consensus mechanism for a node.
func (c *Consensus) stopConsensus() {
                c.running = false
}

// enqueueTransaction adds a transaction to the transaction queue, which will later be commited by the consensus
mechanism.
func (c *Consensus) enqueueTransaction(payload message.TransactionPayload) {
                c.queueMutex.Lock()
                found := false
                for _, val := range c.tsxQueue {
                        if val.From == payload.From && val.To == payload.To && val.Timestamp ==
payload.Timestamp && val.Transaction == payload.Transaction {
                                found = true
                        }
                }

                if !found {
                        c.tsxQueue = append(c.tsxQueue, payload)
                }
                c.queueMutex.Unlock()
}

// Make sure that queue is not empty before using.
func (c *Consensus) dequeueTransaction() message.TransactionPayload {
```

```
		var element message.TransactionPayload

		if len(c.tsxQueue) > 0 {
			element = c.tsxQueue[0]

			if len(c.tsxQueue) > 1 {
				c.tsxQueue = c.tsxQueue[1:]
			} else {
				c.tsxQueue = []message.TransactionPayload{}
			}
		} else {
			element = message.TransactionPayload{}
		}

		return element
}

// receiveUpdatedChain accepts broadcasted blockchains, validates them, and checks if they were sent by the validator.
func (c *Consensus) receiveUpdatedChain(payload message.BlockchainPayload) {
		c.blockchainMutex.Lock()
		currentChain := c.Engine.NodeStore.Blockchain
		updatedChain := payload.Blockchain

		// if len(*currentChain) == 0 then this node has just moved to a new cluster and emptied its blockchain storage, so it should receive the blockchain from
		// the new cluster.
		if len(*currentChain) == 0 || (((len(updatedChain) > len(*currentChain) && updatedChain[0].Header.Generation == (*currentChain)[0].Header.Generation) ||
			updatedChain[0].Header.Generation > (*currentChain)[0].Header.Generation) &&
			updatedChain[len(updatedChain)-1].Header.Validator == c.selectedValidator.Pretty()) {
```

```
                if len(updatedChain) == 1 || block.IsBlockValid(&updatedChain[len(updatedChain)-1],
&updatedChain[len(updatedChain)-2]) {
                        *c.Engine.NodeStore.Blockchain = updatedChain
                        currentChain = c.Engine.NodeStore.Blockchain
                        lastBlock := (*currentChain)[len(*currentChain)-1]

                        // Synchronizing node live wallet.
                        c.Engine.NodeStore.LiveWallet = make(map[string]int)
                        for key, value := range lastBlock.Header.Wallet {
                                c.Engine.NodeStore.LiveWallet[key] = value
                        }

                        // Updated locally stored blockchain and outputing it to stdout.
                        bytes, err := json.MarshalIndent(*c.Engine.NodeStore.Blockchain, "", "  ")
                        if err != nil {
                                log.Fatal(err)
                        }
                        // Green console color:      \x1b[32m
                        // Reset console color:      \x1b[0m
                        if c.Engine.Verbose {
                                runtime.GC() // Run garbage collector every time blockchain is
updated, reduces memory usage immediately after aggregation.
                                fmt.Printf("\x1b[32m%s\x1b[0m\n", string(bytes))
                                printMemUsage()
                        }
                } else {
                        if c.Engine.Verbose {
                                fmt.Println("ReceiveUpdatedChain: Block validation failed!")
                        }
                }
        }
        c.blockchainMutex.Unlock()
```

```
}

// Returns nil if there wasn't more than one transaction value to aggregate.
func (c *Consensus) aggregateChain(blockChain []block.Block) []block.Block {
                        tempList := blockChain[len(blockChain)-1].TransactionList
                        newBlockChain := []block.Block{}

                        if len(tempList) > 1 {
                            txsList := []string{}

                            for _, c := range tempList {
                                    txsList = append(txsList, c.Transaction)
                            }

                            aggregate := weightedMovingAverage(decodeTxsToFloat(txsList))

                            newGenesisBlock := block.CreateGenesisBlock(blockChain[0].Header.Generation+1,
&blockChain[len(blockChain)-1],
                                    strconv.FormatFloat(aggregate, 'f', 6, 64), blockChain[len(blockChain)-
1].Header.Wallet, (*c.Engine.BasicHost).ID(), time.Now().UTC().UnixNano())
                            newBlockChain = append(newBlockChain, *newGenesisBlock)

                            if c.Engine.Verbose {
                                    fmt.Println("\nThis node aggregated the blockchain.")
                            }
                        } else {
                            newBlockChain = nil
                        }

                        return newBlockChain
}
```

```
func decodeTxsToFloat(txsList []string) []float64 {
		txsFloatList := []float64{}

		for i := range txsList {
			tempFloat, err := strconv.ParseFloat(txsList[i], 64)
			if err != nil {
				continue
			}

			txsFloatList = append(txsFloatList, tempFloat)
		}

		return txsFloatList
}

func weightedMovingAverage(txsList []float64) float64 {
		alpha := 0.1
		wAvg := 0.0

		if len(txsList) == 1 {
			return txsList[0]
		}

		for i := range txsList {
			wAvg = (1.0-alpha)*wAvg + alpha*float64(txsList[i])
		}

		return wAvg
}
```

pubsub.go

```
package engine
```

```
// Our use of PubSub is based on this example: https://github.com/libp2p/go-libp2p-examples/pull/74/files
// Credit to: https://github.com/MBakhshi96

import (
                        "context"
                        "crypto/ecdsa"
                        "crypto/rand"
                        "fmt"
                        "log"
                        "strings"

                        "github.com/btcsuite/btcd/btcec"
                        libp2p "github.com/libp2p/go-libp2p"
                        "github.com/libp2p/go-libp2p-core/crypto"
                        "github.com/libp2p/go-libp2p-core/host"
                        "github.com/libp2p/go-libp2p-core/peer"
                        psub "github.com/libp2p/go-libp2p-pubsub"
                        "github.com/multiformats/go-multiaddr"
)

type libp2pPubSub struct {
                        pubsub      *psub.PubSub      // PubSub of each individual node
                        subscription *psub.Subscription // Subscription of individual node
                        topic       string           // PubSub topic
}

// Broadcast Uses PubSub publish to broadcast messages to other peers
func (c *libp2pPubSub) Broadcast(msg string) {
                        // Broadcasting to a topic in PubSub
                        err := c.pubsub.Publish(c.topic, []byte(msg))
                        if err != nil {
```

```
                log.Printf("Error : %v\n", err)
                return
            }
}

// Receive gets message from PubSub in a blocking way
func (c *libp2pPubSub) Receive() (peer.ID, string) {
            // Blocking function for consuming newly received messages
            // We can access message here
            msg, _ := c.subscription.Next(context.Background())
            return msg.GetFrom(), string(msg.Data)
}

// createPeer creates a peer on localhost and configures it to use libp2p.
func (c *libp2pPubSub) createPeer(moniker string, port int) *host.Host {
            // Creating a node
            h, err := createHost(port)
            if err != nil {
                panic(err)
            }

            fmt.Printf("%s is %s\n", moniker, getLocalHostAddress(h))

            // Returning pointer to the created libp2p host
            return &h
}

// initializePubSub creates a PubSub for the peer and also subscribes to a topic
func (c *libp2pPubSub) initializePubSub(h host.Host) {
            var err error
            // Creating pubsub
            // every peer has its own PubSub
```

```
                c.pubsub, err = applyPubSub(h)
                if err != nil {
                    log.Printf("Error : %v\n", err)
                    return
                }

                // Registering to the topic
                c.topic = "TOPIC"
                // Creating a subscription and subscribing to the topic
                c.subscription, err = c.pubsub.Subscribe(c.topic)
                if err != nil {
                    log.Printf("Error : %v\n", err)
                    return
                }

}

// createHost creates a host with some default options and a signing identity
func createHost(port int) (host.Host, error) {
                // Producing private key
                prvKey, err := ecdsa.GenerateKey(btcec.S256(), rand.Reader)
                if err != nil {
                    return nil, err
                }
                sk := (*crypto.Secp256k1PrivateKey)(prvKey)

                // Starting a peer with default configs
                opts := []libp2p.Option{
                    libp2p.ListenAddrStrings(fmt.Sprintf("/ip4/0.0.0.0/tcp/%d", port)),
                    libp2p.Identity(sk),
                    libp2p.DefaultTransports,
                    libp2p.DefaultMuxers,
```

```
                libp2p.DefaultSecurity,
        }

        h, err := libp2p.New(context.Background(), opts...)
        if err != nil {
            return nil, err
        }

        return h, nil
}
// getLocalHostAddress is used for getting address of hosts
func getLocalHostAddress(h host.Host) string {
        for _, addr := range h.Addrs() {
            if strings.Contains(addr.String(), "127.0.0.1") {
                    return addr.String() + "/p2p/" + h.ID().Pretty()
            }
        }
        return ""
}

// applyPubSub creates a new GossipSub with message signing
func applyPubSub(h host.Host) (*psub.PubSub, error) {
        optsPS := []psub.Option{
            psub.WithMessageSigning(true),
        }

        return psub.NewGossipSub(context.Background(), h, optsPS...)
}

// connectHostToPeer is used for connecting a host to another peer
func connectHostToPeer(h host.Host, connectToAddress string) {
        // Creating multi address
```

```
                multiAddr, err := multiaddr.NewMultiaddr(connectToAddress)
                if err != nil {
                    log.Printf("Error : %v\n", err)
                    return
                }

                pInfo, err := peer.AddrInfoFromP2pAddr(multiAddr)
                if err != nil {
                    log.Printf("Error : %v\n", err)
                    return
                }

                err = h.Connect(context.Background(), *pInfo)
                if err != nil {
                    log.Printf("Error : %v\n", err)
                }
}
```

block.go

```
package block

import (
                "crypto/sha256"
                "encoding/hex"
                "fmt"
                "log"
                "sort"
                "strconv"
                "time"

                "github.com/cbergoon/merkletree"
                "github.com/libp2p/go-libp2p-core/peer"
```

```
)

// BlockHeader holds the Block struct contents which are hashed for blockchain integrity.
type BlockHeader struct {
        Generation int
        Index      int
        Timestamp  string
        Hash       string
        PrevHash   string
        RootHash   []byte // Merkle tree root hash which is composed of the concatenated hashes of all transactions in block.
        Wallet     map[string]int
        Validator  string
        Seed       int64 // This seed value can be used for consensus purposes, where all nodes can share a seed for a random number generator.
}

// Block is the struct type held by the blockchain.
type Block struct {
        Header          BlockHeader
        TransactionList []TransactionContent
}

// CreateBlock creates a new block using the previous block hash, and appends one payload.
func CreateBlock(oldBlock *Block, from string, to string, timestamp string, transaction string, wallet map[string]int, validator peer.ID, seed int64) *Block {

        var newBlock Block
        var contentList []merkletree.Content

        for _, c := range oldBlock.TransactionList {
            contentList = append(contentList, c)
        }
```

```
contentList = append(contentList, TransactionContent{From: from, To: to, Timestamp:
timestamp, Transaction: transaction})

tree, err := merkletree.NewTree(contentList)
if err != nil {
    log.Fatal(err)
}

validRootHash, err := tree.VerifyTree()
if err != nil {
    log.Fatal(err)
}

if !validRootHash {
    err := tree.RebuildTree()
    if err != nil {
        log.Fatal(err)
    }

    validRootHash, err = tree.VerifyTree()
    if err != nil {
        log.Fatal(err)
    }

    if !validRootHash {
        log.Fatalln("CreateBock: Failed to build correct merkle tree multiple times.")
    }
}

var tsxContentList []TransactionContent
for _, c := range contentList {
    tsxContentList = append(tsxContentList, c.(TransactionContent))
```

```
        }
        newBlock.TransactionList = tsxContentList

        newBlock.Header = BlockHeader{
            Generation: oldBlock.Header.Generation,
            Index:      oldBlock.Header.Index + 1,
            Timestamp:  time.Now().Format(time.RFC1123),
            Hash:       "",
            PrevHash:   oldBlock.Header.Hash,
            RootHash:   tree.MerkleRoot(),
            Wallet:     wallet,
            Validator:  validator.Pretty(),
            Seed:       seed,
        }

        newBlock.Header.Hash = CalculateHash(&newBlock)

        return &newBlock
}

// CreateBlockWithList creates a new block using the previous block hash, and appends a list of transactions.
func CreateBlockWithList(oldBlock *Block, transactions []TransactionContent, wallet map[string]int, validator peer.ID, seed int64) *Block {

        var newBlock Block
        var contentList []merkletree.Content

        for _, c := range oldBlock.TransactionList {
            contentList = append(contentList, c)
        }

        for _, t := range transactions {
```

```
            contentList = append(contentList, TransactionContent{From: t.From, To: t.To,
Timestamp: t.Timestamp, Transaction: t.Transaction})
        }

        tree, err := merkletree.NewTree(contentList)
        if err != nil {
            log.Fatal(err)
        }

        validRootHash, err := tree.VerifyTree()
        if err != nil {
            log.Fatal(err)
        }

        if !validRootHash {
            err := tree.RebuildTree()
            if err != nil {
                log.Fatal(err)
            }

            validRootHash, err = tree.VerifyTree()
            if err != nil {
                log.Fatal(err)
            }

            if !validRootHash {
                log.Fatalln("CreateBockWithList: Failed to build correct merkle tree multiple
times.")
            }
        }

        var tsxContentList []TransactionContent
```

```
		for _, c := range contentList {
			tsxContentList = append(tsxContentList, c.(TransactionContent))
		}
		newBlock.TransactionList = tsxContentList

		newBlock.Header = BlockHeader{
			Generation: oldBlock.Header.Generation,
			Index:      oldBlock.Header.Index + 1,
			Timestamp:  time.Now().Format(time.RFC1123),
			Hash:       "",
			PrevHash:   oldBlock.Header.Hash,
			RootHash:   tree.MerkleRoot(),
			Wallet:     wallet,
			Validator:  validator.Pretty(),
			Seed:       seed,
		}

		newBlock.Header.Hash = CalculateHash(&newBlock)

		return &newBlock
}

// CreateGenesisBlock creates a new genesis block, if generation is 0 then all other arguments can be nil or 0, they won't be used.
func CreateGenesisBlock(generation int, oldBlock *Block, startingVal string, wallet map[string]int, validator peer.ID, seed int64) *Block {

		var newBlock Block
		var contentList []merkletree.Content

		contentList = append(contentList, TransactionContent{From: "", To: "", Timestamp: time.Now().String(), Transaction: startingVal})
```

```
tree, err := merkletree.NewTree(contentList)
if err != nil {
    log.Fatal(err)
}

validRootHash, err := tree.VerifyTree()
if err != nil {
    log.Fatal(err)
}

if !validRootHash {
    err := tree.RebuildTree()
    if err != nil {
        log.Fatal(err)
    }

    validRootHash, err = tree.VerifyTree()
    if err != nil {
        log.Fatal(err)
    }

    if !validRootHash {
        log.Fatalln("CreateGenesisBlock: Failed to build correct merkle tree multiple
times.")
    }
}

var tsxContentList []TransactionContent
for _, c := range contentList {
    tsxContentList = append(tsxContentList, c.(TransactionContent))
}
newBlock.TransactionList = tsxContentList
```

```
var prevHash string
var tempValidator string

if generation == 0 {
    oldBlock = nil
    prevHash = ""
    startingVal = ""
    wallet = make(map[string]int)
    tempValidator = ""
    seed = 0
} else {
    prevHash = oldBlock.Header.Hash
    tempValidator = validator.Pretty()
}

newBlock.Header = BlockHeader{
    Generation: generation,
    Index:      0,
    Timestamp:  time.Now().Format(time.RFC1123),
    Hash:       "",
    PrevHash:   prevHash,
    RootHash:   tree.MerkleRoot(),
    Wallet:     wallet,
    Validator:  tempValidator,
    Seed:       seed,
}

newBlock.Header.Hash = CalculateHash(&newBlock)

return &newBlock
}
```

```
// IsBlockValid makes sure block is valid by checking index, and comparing the hash of the previous block.
func IsBlockValid(newBlock, oldBlock *Block) bool {
        if oldBlock.Header.Index+1 != newBlock.Header.Index {
            return false
        }
        if oldBlock.Header.Hash != newBlock.Header.PrevHash {
            return false
        }
        if CalculateHash(newBlock) != newBlock.Header.Hash {
            return false
        }
        var contentList []merkletree.Content
        for _, c := range newBlock.TransactionList {
            contentList = append(contentList, c)
        }
        tree, err := merkletree.NewTree(contentList)
        if err != nil {
            log.Fatal(err)
        }
        validRootHash, err := tree.VerifyTree()
        if err != nil {
            log.Fatal(err)
        }
        if !validRootHash {
            err := tree.RebuildTree()
            if err != nil {
                log.Fatal(err)
            }
            validRootHash, err = tree.VerifyTree()
            if err != nil {
                log.Fatal(err)
```

```
                }
                if !validRootHash {
                        log.Fatalln("IsBlockValid: Failed to build correct merkle tree multiple times.")
                }
        }
        if fmt.Sprintf("%x", newBlock.Header.RootHash) != fmt.Sprintf("%x", tree.MerkleRoot()) {
                return false
        }
        return true
}

// CalculateHash performs SHA256 hashing on the contents of the block struct.
func CalculateHash(block *Block) string {
        record := strconv.Itoa(block.Header.Generation) + strconv.Itoa(block.Header.Index) +
block.Header.Timestamp +
                fmt.Sprintf("%x", block.Header.RootHash) + block.Header.PrevHash +
mapToString(block.Header.Wallet)
        h := sha256.New()
        h.Write([]byte(record))
        hashed := h.Sum(nil)
        return hex.EncodeToString(hashed)
}

// mapToString converts map to string of "Key: Value" ordered pairs. Mainly used for hashing purposes.
func mapToString(wallet map[string]int) string {
        var list []string
        str := ""
        for key, value := range wallet {
                list = append(list, fmt.Sprintf("%s:%d", key, value))
        }
        sort.Strings(list)
        for _, pair := range list {
```

```
                str += pair + " "
        }
        return str
}
```

transaction-content.go

```
package block

import (
        "crypto/sha256"
        "fmt"
        "github.com/cbergoon/merkletree"
)

// TransactionContent implements the Content interface provided by merkletree and represents the content stored in the tree.
type TransactionContent struct {
        From        string
        To          string
        Timestamp   string
        Transaction string
}
// CalculateHash hashes the values of a TransactionContent
func (t TransactionContent) CalculateHash() ([]byte, error) {
        h := sha256.New()
        if _, err := h.Write([]byte(fmt.Sprintf("%s%s%s%s", t.From, t.To, t.Timestamp, t.Transaction))); err != nil {
                return nil, err
        }
        return h.Sum(nil), nil
}

// Equals tests for equality of two Contents
```

```
func (t TransactionContent) Equals(other merkletree.Content) (bool, error) {
                return t.From == other.(TransactionContent).From && t.To ==
other.(TransactionContent).To &&
                        t.Timestamp == other.(TransactionContent).Timestamp && t.Transaction ==
other.(TransactionContent).Transaction, nil
}

//String returns a string representation of the content.
func (t TransactionContent) String() string {
                return fmt.Sprintf("From: %s, To: %s, Timestamp: %s, Transaction: %s", t.From, t.To,
t.Timestamp, t.Transaction)
}

message.go
package message

import (
        "encoding/json"

        "github.com/EAGnR/sensor-chain/src/block"
        "github.com/libp2p/go-libp2p-core/peer"
)

// PayloadType is the identifier for the type of data in the Payload.
type PayloadType string

const (
        // TransactionType is the payload type for holding transaction strings.
        TransactionType PayloadType = "TransactionPayload"

        // BlockchainType is the payload type for holding a peers list, alongside its update time.
        BlockchainType PayloadType = "BlockchainPayload"
```

```
	// // PeerListType is the payload type for holding a peer list, alongside its update time.
	// PeerListType PayloadType = “PeerListPayload”
)

// Message is the struct that is sent as a marshaled JSON over the network,
// with the metadata field Type to let the receiving node know how unmarshal it.
type Message struct {
	Type       PayloadType
	RawPayload json.RawMessage
}

// TransactionPayload is the payload type for holding transaction strings.
type TransactionPayload struct {
	From       peer.ID
	To         peer.ID
	Timestamp   string
	Transaction string
}

// BlockchainPayload is the payload type for holding a slice of block (the blockchain).
type BlockchainPayload struct {
	Blockchain []block.Block
}
```

Store.go

```
package store

import (
	“github.com/EAGnR/sensor-chain/src/block”
)
```

```
// Store contains the storage of a node.
type Store struct {
	Blockchain *[]block.Block
	LiveWallet map[string]int
}

// CreateStore creates a new empty node.
func CreateStore() *Store {
	var newStore Store

	newStore.Blockchain = &[]block.Block{}
	newStore.LiveWallet = make(map[string]int)

	*newStore.Blockchain = append(*newStore.Blockchain, *block.CreateGenesisBlock(0, nil, “”, nil, “”, 0))

	return &newStore
}
```

sensor-chain.go

```
package main

import (
	“github.com/EAGnR/sensor-chain/src/engine”
)

func main() {
	engine.RunNodeManager()
}
```

LIST OF SYMBOLS, ABBREVIATIONS, AND ACRONYMS

Symbol	Description
TX	Abbreviation of transaction
BC	Abbreviation of blockchain
$\mathbb{C}$	Voronoi diagram or set of Voronoi cells
C_i	i-th Voronoi cell
T_{chain}	Temporal constraint for blockchain
T_{block}	Block creation time constraint
n	Total number of sensing nodes
m	Number of sensors in a single cell
G_i^t	Local network in i-th cell at time t
V_i^t	Set of vertices of location network G_i^t
E_i^t	Set of edges between the nodes in V_i^t
S	A sensor node
B_i^t	Local blockchain generated by G_i^t