HELLENIC REPUBLIC
National and Kapodistrian University of Athens
EST. 1837

Department of Physics
Section of Electronic Physics
and Systems Ensuring

Ensuring consensus on trust issues in capability-limited node networks with Blockchain technology

Maria Koutsoukou, 2020513

MSc Electronics and Radioelectrology
Control and Computing

Athens, 2023

Contents

List of Figures

Abstract

In this thesis, we review how the blockchain protocol could be applied so as to ensure the consensus of the network nodes on matters of trust, while taking into account the limited resources of the nodes. First, we will present the system we will be working on, a Mobile Ad Hoc Network (MANET) which uses a clustering scheme, based on the cost of analysis/processing concept, along with a trust mechanism that together result in cluster persistence and quick discovery of malicious nodes. Then, we will introduce the blockchain protocol and its main concepts. A review of related work will be conducted to assess the various approaches utilized by others in incorporating the blockchain protocol into their respective systems. Subsequently, the Hyperledger Fabric framework, which shall aid in the implementation of the proposed system, shall be presented in detail. Finally, the proposed system, which merges the given MANET simulation with the blockchain technology enabled by Hyperledger Fabric, will be presented, and its results discussed. Our analysis will also include an evaluation of its effectiveness in addressing consensus on trust issues, and possible avenues for future enhancements.

Keywords – Blockchain, MANET, Hyperledger Fabric, Clustering, Trust Consensus

Περίληψη

Σε αυτή τη διπλωματική εργασία, εξετάζουμε πώς θα μπορούσε να εφαρμοστεί το πρωτόκολλο blockchain, ώστε να διασφαλιστεί η συναίνεση των κόμβων του δικτύου σε θέματα εμπιστοσύνης, λαμβάνοντας παράλληλα υπόψη τους περιορισμένους πόρους των κόμβων. Αρχικά, θα παρουσιάσουμε το σύστημα στο οποίο θα εργαστούμε, ένα Mobile Ad Hoc Network (MANET) το οποίο χρησιμοποιεί ένα σχήμα ομαδοποίησης, βασισμένο στην έννοια του κόστους ανάλυσης/επεξεργασίας, μαζί με έναν μηχανισμό εμπιστοσύνης που από κοινού έχουν ως αποτέλεσμα τη διαρκή και γρήγορη ανακάλυψη κακόβουλων κόμβων. Στη συνέχεια, θα παρουσιάσουμε το πρωτόκολλο blockchain και τις κύριες έννοιές του. Μια ανασκόπηση της σχετικής εργασίας θα διεξαχθεί για να αξιολογηθούν οι διάφορες προσεγγίσεις που χρησιμοποιούνται από άλλους για την ενσωμάτωση του πρωτοκόλλου blockchain στα αντίστοιχα συστήματά τους. Στη συνέχεια, θα παρουσιαστεί αναλυτικά το Hyperledger Fabric framework, το οποίο θα βοηθήσει στην εφαρμογή του προτεινόμενου συστήματος. Τέλος, θα παρουσιαστεί το προτεινόμενο σύστημα, το οποίο συγχωνεύει τη δεδομένη προσομοίωση MANET με την τεχνολογία blockchain μέσα από το Hyperledger Fabric και θα συζητηθούν τα αποτελέσματά του. Η ανάλυσή μας θα περιλαμβάνει, επίσης, αξιολόγηση της αποτελεσματικότητάς του στην διασφάλιση της συναίνεσης των κόμβων σε θέματα εμπιστοσύνης και πιθανές μελλοντικές βελτιώσεις.

Λέξεις κλειδιά – Blockchain, MANET, Hyperledger Fabric, Clustering, Συναίνεση Εμπιστοσύνης

Introduction

In this thesis, we review how the blockchain protocol could be applied so as to ensure the consensus of the network nodes on matters of trust, while taking into account the limited resources of the nodes. Blockchain technology was initially used in the field of cryptocurrency (and digital currencies in general), but it can have wider applications in other areas such as e-commerce, the transparency of historical, financial and other records, in areas such as banking, construction sectors, in news transparency, etc.

Blockchain is ideal for delivering information because it provides immediate, shared and completely transparent information stored on an immutable ledger that can be accessed either by anyone or only by permissioned network members. A blockchain network can track orders, payments, accounts, production and much more. And because members share a single view of the truth, you can see all details of a transaction end to end, giving you greater confidence, as well as new efficiencies and opportunities.

The same technology could be applied in the case of e.g. mobile unstructured networks, the Internet of Things (IoT) and generally any network of nodes with possibly limited capabilities in which security and trust issues arise in transactions between nodes. However, the limited capabilities (computing, memory, energy) of the nodes impose limitations on the techniques for detecting malicious nodes and disseminating this information in the network.

Through the next chapters, we will look into how we could integrate the blockchain protocol into a Mobile Ad Hoc Network (MANET) and examine how its security and efficiency could be improved.

In Chapter 1, we will present the system we will be working on as the current framework, a MANET which uses a clustering scheme, based on the cost of analysis/processing concept, along with a trust mechanism that together result in cluster persistence and quick discovery of malicious nodes.

In Chapter 2, we will introduce the blockchain protocol in a theoretical perspective. The main focus of the present chapter is to review the fundamentals of the blockchain technology and its concepts.

In Chapter 3, related work will be studied in order to assess the various approaches utilized by others in incorporating the blockchain protocol into their respective systems.

In Chapter 4, the Hyperledger Fabric framework, which we will be using in our proposed system, shall be presented in detail.

Finally, in Chapter 5, the proposed system, which combines the given MANET simulation with the blockchain technology facilitated by Hyperledger Fabric, will be presented and its results discussed. Also, an evaluation of the system's effectiveness in addressing consensus on trust issues will be conducted, and prospects for future improvements will be explored.

Chapter 1

Current Framework

1.1 Overview

The given framework [1], [2], [3], this project is based on, is a Mobile Ad-Hoc Network (MANET), which is efficiently clustered based on the cost of analysis concept and then using nodes' experience, trust information dissemination is achieved through the network.

The nodes of such network have limited capabilities, processing power, energy and memory. Meanwhile, they are prone to a growing number of malicious attacks, meaning systems like this increase the need for trust. Thus, the network under study focuses on trust establishment and trust management of nodes. Following its deployment, the network should be able to operate without hindrances, fending off security threats while establishing the integrity of the exchanged data.

In this framework, the nodes are first clustered together, each cluster made up of one leader, the cluster head (CH), who is chosen after an election precedes, and its cluster members (CMs). Trust information exchange occurs when the cluster is initially formed and later when a cluster member queries the cluster head for trust information on another cluster member before proceeding with a transaction with it.

In order to keep these interactions honest, a mechanism, that promptly and accurately alerts the network for malicious node activity, has been developed. It may, also, be used to restore the trust of a node in case of rejecting former malicious behaviour or in case it was wrongly classified as malicious. Such mechanism is quite convenient when it comes to avoiding network attacks which can be devastating for the network performance and the integrity of the exchanged data.

1.1.1 Mobile Ad-Hoc Networks

MANETs consist of mobile nodes connected wirelessly without having a fixed infrastructure. They may move in specific paths, but usually the nodes are free to move randomly as the network topology changes frequently. Each node is required to behave as a router, provide intrusion detection services and the ability to pass on information to other nodes, using available resources.

The first thing that needs to be fulfilled in such network is an efficient power management scheme to ensure the availability of the nodes. Nodes must always be available to perform the services they are required to. Inability to do so may prove to be catastrophic to the whole network. In case of such an event, a temporary solution would be the diversion of traffic to other nodes. Yet, a network crash may still be unavoidable due to heavy traffic in the alternate paths [4].

Another thing that should be properly handled, is the trust management mechanism. This will ensure that if any malicious nodes are present in the network, they will be timely identified and isolated until their honest behaviour is restored. So as to make this possible, the network must not be plagued with high latency as this will hinder the immediate propagation of trust information and the respective response of the nodes to malicious behaviour [5].

Lastly, if the malicious node does indeed restore its previously healthy behaviour, the trust management mechanism should be able to re-enter the node to the network. The same should be done in case the node in question was falsely classified as malicious.

All these issues are addressed in this framework and will be explained in detail in the following sections.

1.1.2 Network Security

Network security in MANET is the most important issue for the basic functionality of the network [6]. It is what protects the network from activities that threaten its smooth operation and its intended purpose.

Security issues present a challenge in MANET networks due to power constrains, transmission range and the mobility of the nodes. The CIA model is used to address the security needs of the MANET; Confidentiality, Integrity of the data and Availability of the network services.

Confidentiality refers to the protection of sensitive information from unauthorized access. In MANETs, this can be achieved through the use of encryption algorithms, such as Advanced Encryption Standard (AES), to secure communication between nodes.

Integrity refers to the protection of information from unauthorized modification or alteration. In MANETs, this can be achieved through the use of digital signatures and message authentication codes (MACs) to ensure that data has not been tampered with during transmission. This usually calls for a third party trusted Certificate Authority (CA), which adheres to the X.509 standard. By using a trusted CA, nodes in a MANET can be assured that their communication partners are indeed who they claim to be, thus enhancing the security of the network. For that to be successfully applied in a mobile network the nodes must initially be within the CA transmission range, which will severely restrain the node deployment.

Availability refers to ensuring that authorized users have access to the information they need, when they need it. In MANETs, this can be achieved through the use of routing protocols that help to maintain connectivity between nodes, even in the presence of node failures or network partitions.

There are also several security mechanisms that can be used to enhance the CIA security of MANETs, such as firewalls, intrusion detection systems (IDSs), and secure routing protocols. Additionally, it's important to follow best practices for security, such as regularly updating software and firmware, using strong passwords, and disabling unnecessary services.

1.1.3 Network Clustering

Nodes are exposed to various kinds of risks, some of which can be confronted by properly clustering the network [1]. Clustering can also be an asset when required to uniformly exploit network resources in order to optimally diffuse information throughout the network. Furthermore, in the event of depletion of energy in a vital number of nodes, the network could avoid an imminent termination if enough nodes may still have adequate power and resources to keep functioning. In pursuit of these benefits, the correct clustering algorithm must be selected to suit the network topology in question.

An appropriate resource management scheme will increase the longevity, efficiency and credibility of the network. To create a scheme like that, the Cost of Analysis/Processing (CoA/P) as an extension of Cost of Analysis [7] will be utilized. CoA/P has been proven to be reliable and secure to deal with issues that in any other case would require the system to reveal sensitive information about the status of specific node resources parameters, such as the node's power status. If that were to happen, for example one node's computational capabilities be revealed, the node could potentially become a target of an attack and potentially be used to corrupt the whole network.

As stated before, the cluster will consist of the CH and the CMs. The main role will be acquired by the CH, which will be of immense help to the cluster even though it is an ordinary node. CH stores routing paths information and trust information in respect to their CMs. They act as a council to CMs during their transactions with other CMs. They, also, monitor their CMs for malicious behaviour and in such case they immediately inform the other CMs. They promptly notify their CMs for an opposite transition, as well. On top of that, they run Intrusion Detection Software (IDS), which, in our case, is substituted by the task of maintaining trust values for the cluster, processing them and responding to pertinent queries. All these services are energy heavy for one node, thus motivating a node to undertake that role is of great importance. This is possible through a reward scheme, as proposed in [7], which offers the CH future privileges such as increased bandwidth and service priority.

1.2 Cluster Formation and Maintenance

The concept of CoA/P and payment [7] was introduced to incentivise nodes to accurately reveal their ability to serve as CHs without compromising the security of the node or the network. This clustering method, forming 1-hop clusters, motivates the nodes to be truthful when the time comes to be elected as CHs without revealing sensitive information, such as the energy level of the node. The proposal aims to address the issue of nodes potentially lying about their energy levels to conserve it or to become a CH and negatively impact the cluster. Thus, the node who actually has the most resources will be elected by neighbouring nodes as CH until another election is held. Due to the additional responsibilities the CH has, the CH will probably have consumed most of its resources by the time the next election is held, so a different CH will most certainly be elected. Thus, the tasks burdening the former CH will now be transferred to the new CH. In the next sections, the cluster formation and maintenance will be presented in more detail, as proposed in [1], [2].

1.2.1 Trust Vectors

Every node in the network has its own Local Trust Vector (LTV) which reflects its evaluation of the trustworthiness of all other nodes. The LTV components range from 0 to 1, where 0 represents the least trusted node and 1 represents a node that is fully trusted. After a cluster has been formed, the CH's LTV is elevated to the Global Trust Vector (GTV). The CH then requests a trust report, called T_{rep}, from every node in the cluster.

These reports consist of the node's LTV components and are represented as $T_{rep}(i, j, t)$, where i is the reporting node, j is the node whose trust is reported by i node and t is the time this trust was calculated. This system allows each node to maintain its own trust evaluations of other nodes, while also promoting the CH's LTV to serve as a collective trust evaluation for the entire cluster. In short, the trust vector of an ordinary CM is referred to as LTV, while that of the CH is referred to as GTV.

1.2.2 Cost of Analysis/Processing

Following [7], first the percentage of sampling needs to be defined, which is essentially the relative reputation of node in question:

$$PS_i = \frac{R_i}{\sum_{k=1}^{N} R_k}, \tag{1.1}$$

where R_i is the reputation of the node, i.e. the trust score the CH has stored in its LTV component for this node, and $\sum_{k=1}^{N} R_k$ is the total reputation of the nodes in the network that appear in the CH's LTV.

Now, the CoA/P may be defined as:

$$c_i = \begin{cases} \infty, & \text{if } E_i < E_{ch} \\ \frac{\frac{R_i}{\sum_{k=1}^{N} R_k}}{E_i}, & \text{otherwise} \end{cases}, \tag{1.2}$$

where E_i is the energy of the node and E_{ch} is the minimum energy required for the operation of the cluster.

1.2.3 Cluster head election

Elections for a new CH are held whenever is deemed necessary. That is, at the beginning of the MANET's operation, when nodes move in or out of the cluster, and can have an impact on the entire network, or when a CH quits due to energy levels dropping below a certain, CH-specific, threshold, as seen in eq. 1.2, a node like that will have $c_i = \infty$. On the other end, small values of c_i indicate that a node has enough resources to act as a CH.

The election process begins with each node broadcasting its own cost of analysis/processing c_i to the nodes that are within transmission range. The nodes receive the c_i of all of their neighbours and sort them in ascending order. CoA/P c_i does not reveal sensitive information about the node, yet it serves as measure of the node's capability to act as a CH.

Once the cost of analysis for all nodes is known, each node votes for its neighbour or itself, with the lowest cost of analysis c_i as it appears in the respective node list. The CH is elected by its 1-hop neighbours. All the nodes that are 1-hop away from the CH become CMs. The result of this procedure will be 1-hop clusters, since only the CMs that are within range of one another vote, preventing cluster overpopulation as shown in Fig. 1.1. Note that isolated nodes, and thus with no neighbours, are considered to be CHs, forming clusters with only themselves.

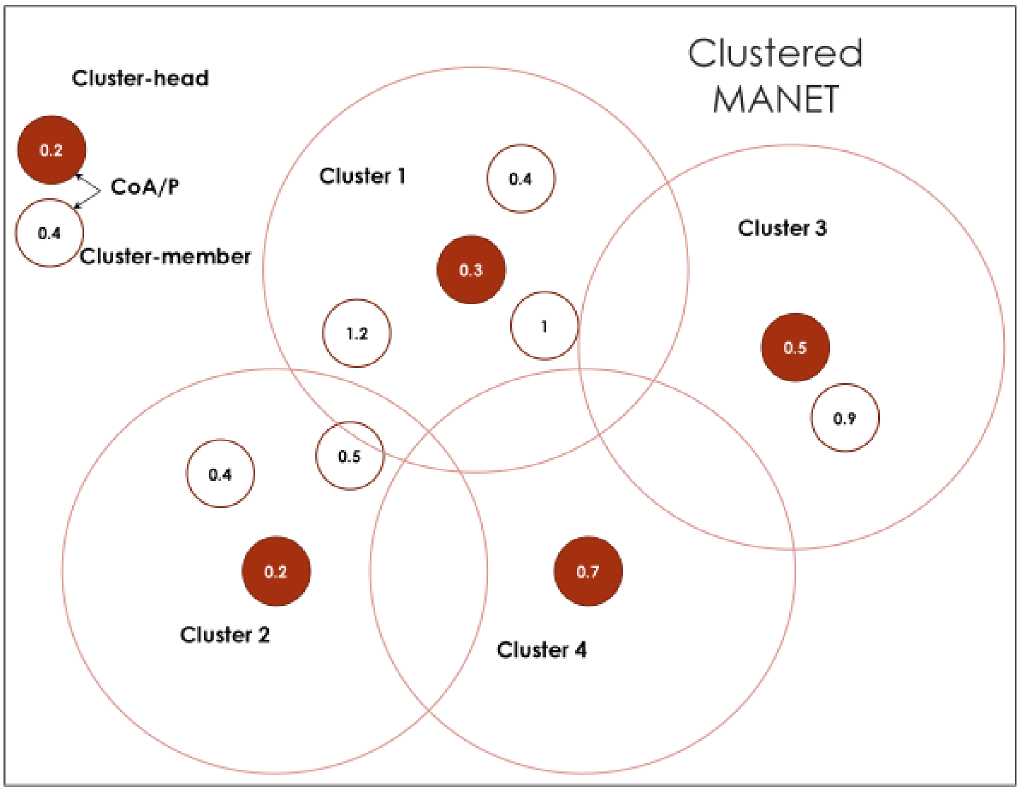


Figure 1.1: *MANET clustering example*

1.2.4 Transactions mechanism

Once the cluster is formed, the CH merges the LTVs of its CM's with its own and updates the respective GTV appropriately. Now, members of the same cluster can begin interacting with each other with the transaction mechanism. The client node, the one that initiates the transaction, and the server node follow the procedure shown in Fig. 1.2. If the server node is available to proceed with the transaction with the client node, then the client node must consult the CH for the server's estimated trust level.

Trust T_{ij} is a float number in the [0, 1] range which reflects the opinion that node i has, regarding node j, and is formed from past observations of the behavior of node j and a weighted consultation of the reputation vector kept by the CH. The closest to 1, the more reputable node j is. Therefore, a trust T_{ij} value above a certain threshold, set by the client node, will initiate the transaction, while the opposite will cause the client node to abort the transaction. If the transaction is ended successfully, the client node evaluates the transaction in terms of trust, updates its LTV and notifies the CH about the new trust value of the server-node so that the GTV is updated accordingly.

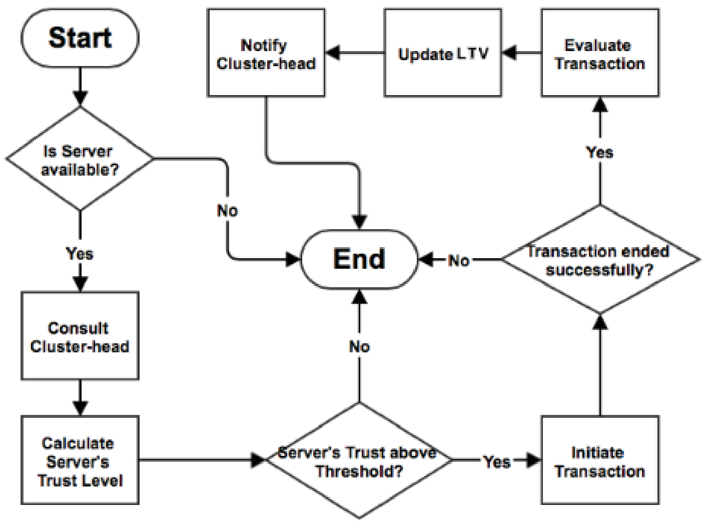


Figure 1.2: *Transaction flowchart*

The GTV components may be changed in two occasions. Firstly, as already stated, when a transaction is completed successfully and the client node transmits the server's trust evaluation, according to its behaviour during the transaction, to the CH. The CH, then, combines this evaluation to its own and broadcasts the new GTV. Secondly, when the CH concludes that a node is malicious due to other nodes' evaluations. The CH proceeds with the broadcast of the proper message.

Therefore, the GTV is disseminated to every CM, so that every node is up to date when it comes to trust information. The GTV is a reflection of a node's reputation in the cluster by the whole cluster and it will be preserved even if the cluster is reformed due to a new round of elections.

Bear in mind that a node that is beyond CH's broadcasting range can not initiate a transaction, since it cannot receive CH's consultation, or report back the outcome of the transaction.

1.2.5 Trust evaluation

As discussed, before a client-node proceeds with a transaction with a server-node, the client-node must consult its CH first. The client-node's i LTV is updated after the CH transmits a message $CH_{rep}(i, j, t)$ containing the estimation of the reputation, at a specific time t, regarding the server-node j, as shown below:

$$T_{ij} = LTV(j,t) = eLTV \times CH_{rep}(i,j,t) + (1 - eLTV) \times LTV(j,t-1), \tag{1.3}$$

where $eLTV$ is the weighing factor which is node-specific.

Thus, the trust T_{ij} the node i has for node j is estimated. If $T_{ij} \geq T_{thr}$, where T_{thr} is the trust threshold of node i, the transaction is initiated. At the end of the transaction, node i reports to the CH about the outcome of this transaction regarding the behaviour of the server-node j, at a certain time t, in a message $T_{rep}(i, j, t)$. In the opposite case, $T_{ij} < T_{thr}$, where T_{thr}, the transaction is cancelled.

The CH collects all the information from the CMs and integrates it to its GTV as follows:

$$GTV(j,t) = aLPF \times \frac{\sum_{i=1}^{n} T_{rep}(i,j,t)}{N} + (1 - aLPF) \times GTV(j,t-1), \tag{1.4}$$

where $aLPF$ is CH's weighing factor in the range [0, 1] or else an α-parameter low-pass filter and N is the number of reporting nodes. It expresses the behavior of the CH. A "selfish" CH would have an $aLPF$ closer to 0 and vice versa. The $aLPF$ can also be chosen as a function of time to reflect the information aging, as older information may be of less importance.

In the event that a client node observes malicious behavior from a server node that reduces the trust level of the node to the point of characterizing that node malicious, then an epidemic algorithm is invoked and this information is spread throughout the network. Direct effect of that is that trust information is readily available and the network reaction time is minimized.

1.2.6 Preservation of trust information

When an election is initiated, the clusters as were before and after the new election will most likely be different. It is of vital importance to preserve trust information during re-clustering by transferring it intact to the newly formed clusters. Thus, trust information is disseminated through the network and can be used to predict the future behavior of CMs. Meanwhile, malicious nodes need to be kept at bay so as not to disseminate false information regarding the trust of other nodes, also known as badmouthing attack.

Right after cluster formation, the CH's LTV remains the same. Then, all the nodes that belong to the newly formed cluster report to the CH their own LTVs so they can be incorporated in the CH's GTV alongside its own LTV. All CMs will carry the same-accurate-information that was acquired before the cluster re-formation. This way, malicious nodes, if present in a cluster, will be prevented from announcing false LTVs as they have no way of knowing if some of the former CMs, that may have flagged them as malicious, are also participating in the new cluster. It is not in the malicious node's best interest to be revealed from the beginning so the only option they have, at least during re-clustering, is to announce the correct LTV.

1.3 Simulation Description and Assumptions

For simulation purposes, Java was used to set up a MANET with nodes moving in a rectangle field (1000 units $\times$ 1000 units) following a Random Waypoint pattern. Each node is able to determine its exact location on the field. The nodes are uniquely identified by an ID number and are able to transmit information up to a certain range (100 units). In this simulation, no cryptographic scheme was applied. The motion data file was generated using mobism [8]. The healthy-to-malicious and malicious-to-healthy transition is shown in Fig. 1.3 along with the transition probabilities.

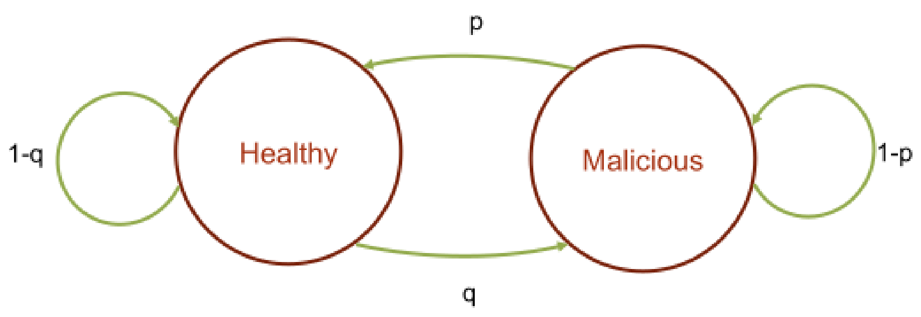


Figure 1.3: *Healthy to malicious transition*

Some assumptions made for the purpose of the simulation are shown below [1]:

- Node number: Any number between 20 - 100 will be sufficient as it will allow as to have adequate clusters to draw conclusions, while not making the simulation too heavy. For the purposes of the project, we chose 100.

- Simulation duration: 5000 clock ticks.

- Node range: 100 units will suffice, as a short range leads to a large number of unstable clusters, while a long range leads to stable but overpopulated clusters.

- Transaction duration and mean time between transactions: Exponentially distributed with λ = 0.1, using $[\frac{log(1-randomFloat)}{(-1)\times lambda} + 1]$.

- Trust threshold: We set the trust threshold to 0.5 for every node.

- Initial trust: We assumed that all nodes are equally trustworthy, at the beginning of the simulation, by setting a trust value 0.6 for all. Anything lower will exclude many nodes from further transactions, while higher is too optimistic and possibly exposes the entire network to risks.

- Trust evaluation following a transaction: It is essentially a decimal number in the range [0, 1], although this is typically assessed and quantified by the user or an intelligent agent with user defined parameters. In our case, following the performance of a certain transaction, this is quantified at random following the normal distribution with expectation (mean) $EV = 0.5$ and standard deviation $\sigma = 0.1$, using $randomGaussian \times \sigma + EV$.

- $aLPF$ parameter of CH's low pass filter: We set $aLPF = 0.4$ which means that, for the GTV formation and update, the CH relies 60% on its own opinion and 40% on the opinion of the cluster members.

- $eLTV$ parameter of cluster members: Similar to the above, we chose $eLTV = 0.8$.

- Node energy: All nodes have an initial energy in the range of [3000, 4000]. With energy in this range we end up with only a couple of inactive nodes at the end of the simulation, which we consider acceptable.

- Transaction cost: The energy cost of a transaction is 1 energy unit per time unit.

- Healthy-to-malicious probability: The probability of a healthy node to become malicious was assumed to be 0.01.
- Malicious-to-healthy probability: The probability of a malicious node to become healthy was assumed to be 0.03.
- Malicious behavior penalty: If a node exhibits malicious behavior for more than 50% of the transaction duration, gets a penalty of 0.5 decrease to its trust which is enough to exclude it from future transactions.
- Trust restoration: Provision is made for the trust restoration of (possibly ex-) malicious nodes.The A3 metric deals with that, which will be discussed below.

After initialization and the first election which will lead to the formation of the clusters each node will have a table of its 1-hop neighbours. At some point a node will choose to have a transaction with a random neighbour. As already stated, the client-node will consult the CH for the server-node's trust evaluation and after combining it with its own opinion will end up with a trust value either above the trust threshold, which will allow the client node to proceed with the transaction, or below, which means the transaction will be cancelled.

The transaction will terminate successfully if the nodes remain within range throughout the transaction. When the transaction is completed, the client-node will evaluate the server-node regarding its trust level. At first, the client-node will update its LTV. Then, it will inform the CH with its evaluation, who in turn will update its GTV accordingly. In case the server-node shows signs of malicious behaviour for more than 50% of the transaction duration, it will be penalized by a 0.5 decrease to its trust by the client-node.

We assume 3 metrics, namely A1, A2 and A3 to measure how quickly the mechanism responds to changes in the node environment.

- A1: The A1 metric refers to the time interval between a healthy-to-malicious transition when spotted by a client-node, as shown in Fig. 1.4. At the beginning of the transaction server-node's trust was above the trust threshold, thus it was initiated. While the transaction was taking place, the server-node fell to a malicious state which lasted for more than 50% of the total transaction time resulting to the server-node receiving the malicious behaviour penalty.
- A2: The A2 metric refers to the time interval from a healthy-to-malicious transition until the beginning of a transaction, which was

ultimately canceled. The client will not initiate a transaction with a server-node that has a low trust score, even if it is not malicious.

- A3: The A3 metric refers to the time interval from a healthy-to-malicious transaction, until the trust of the node is restored. The CH intervenes by increasing the trust value of the low-trust node in its GTV.

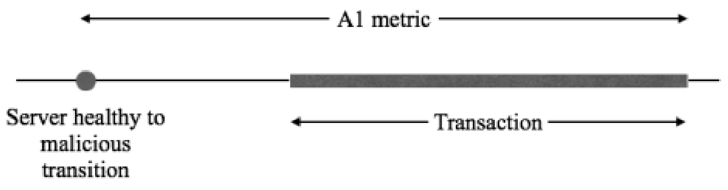


Figure 1.4: *A1 metric*

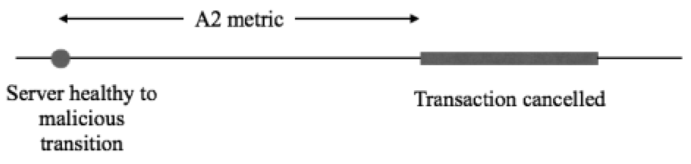


Figure 1.5: *A2 metric*

In our case, in order to spot a malicious node at least one transaction with it must happen before the node is evaluated and flagged as malicious. Furthermore, it is not certain that a node is malicious based only on the trust evaluation of the node. The A1 metric shows how many nodes seemed healthy prior to a transaction. The A2 metric refers to nodes that have already been evaluated, meaning that at least one transaction with another client-node has already taken place, and have been assigned a trust level value below the trust threshold. This will show how many of the low-trust nodes are actually healthy and transactions with them are unfairly cancelled either because they were strictly evaluated earlier, or they were malicious but they have been restored to healthy in the course of the simulation. The A3 metric deals with the restoration of healthy status. CHs will randomly select nodes that are classified as malicious and restore their health status in order to give them a chance to improve their trust level in future transactions. The risk involved in such a decision must be taken by the CH.

Note that the $aLPF$ parameter, which shows how selfish the CH behavior is, in this simulation does not account for the age of the information received, as the more recent means more reliable, or the trust of the reporting node, as the evaluation of a trusted node should be more valuable than that of a low-trust node.

Chapter 2

Blockchain Fundamentals

2.1 Overview

Blockchain is a distributed and immutable database or ledger that is shared among the nodes of a computer network consisting of growing lists of definite and verifiable records of every single event ever occurred securely linked together using cryptography, as mentioned in [9] and [10]. As a database, a blockchain stores information electronically in digital format. The cryptocurrency systems, such as Bitcoin, use the blockchain to maintain a secure and decentralized record of transactions. The novelty of a blockchain is that it guarantees the credibility and security of a data file and assures trust without the need for a trusted authority or central server.

The main difference between a typical database and a blockchain is how the data is structured. A typical database usually structures its data into tables, whereas a blockchain structures its data into groups, known as blocks, that are linked together forming a chain of data known as the blockchain. This data structure inherently makes an irreversible timeline of data when implemented in a decentralized nature. When a block is filled, it cannot be tampered with and becomes a part of this timeline.

Blocks have certain storage capacities and, when filled, are closed and linked to the previously filled block. Each block contains a cryptographic hash of the previous block, an exact timestamp when it is added to the chain, and transaction data (generally represented as a Merkle tree, where data nodes are represented by leaves). The timestamp proves that the transaction data existed when the block was created. Since each block contains information about the previous block, they effectively form a chain (linked list data structure), with each additional block linking to the ones before it. Consequently, blockchain transactions are irreversible in that, once they are recorded, the data in any given block cannot be altered retroactively without altering all subsequent blocks.

Blockchain communications are typically peer-to-peer (P2P). Nodes collectively adhere to a consensus algorithm protocol to add and validate new transaction blocks. The consensus algorithm has been designed to achieve reliability in a network that includes unreliable nodes. The consensus algorithm within the blockchain network ensures that all network agents have the same copy of the ledger. Confirmation of transactions and aggregation in blockchain is accomplished through consensus protocols.

One more thing that needs to be mentioned is that blockchain networks can be categorised into public, private or permissioned [11]. Public blockchains can be accessed by anyone, private blockchains are accessed only by selected users and permissioned blockchains are a hybrid of public and private blockchains where anyone can access them as long as they have permission from the administrators to do so.

2.2 Features

Now that we have some basic understanding of blockchain technology let us review its primary features, as in [12], [13].

- Decentralization: With an increasing number of IoT devices in a network, the use of centralized framework may not be feasible, as such an approach exposes the network to the single point of failure problem. Blockchain technology utilizes the idea of distributed and decentralized computing and storage, meaning that a group of nodes maintains the network, making it decentralized. Each entity will be in charge of storing its own copy of the blockchain. Hence, there is no need for a central management entity and the problems that follow. The decentralization of a system makes it highly fault-tolerant, as it is organized by algorithms, and less prone to breakdowns from malicious attacks, since attacking the system is more expensive for hackers. Moreover, users have control over their properties, as they don't have to rely on any third party to maintain their assets, and they will not be scammed as the systems run purely on algorithms.

- Immutability: Immutability means something that can't be changed or altered. One of the key characteristics of blockchain technology that ensures that the network remains permanent and unalterable. As previously noted, every node in the system has a copy of the digital ledger, contributing to the stability and consistency of the network. In order to add a transaction every node needs to check its validity. If the majority thinks it's valid, then it's added to the ledger, otherwise it will not be added. This promotes transparency and makes it corruption-proof. So, once a piece of information is recorded and confirmed in the blockchain, then it cannot be modified or deleted from the network. Also, information cannot be added arbitrarily.

- Security: Security is comprised mainly of confidentiality, integrity, and availability. Blockchain uses cryptography to add another layer of protection for users. Cryptography is a rather complex mathematical algorithm that acts as a firewall for attacks. Hash functions are used to chain blocks which ensures integrity and immutability. Once blocks are connected, data contained within cannot be subsequently altered, as it will mean changing the hash IDs, which is impossible. The availability requirement is fulfilled inherently in blockchain given its distributed nature and that data is always available. The confidentiality requirement is achieved in permissioned blockchain solutions, where users are permitted to access just the data they are authorized to access through the use of permissions. Let us not forget the fact that transactions are encrypted before being linked to the existing ledger. Moreover, for each entity willing to join the blockchain network, there is a need for membership authentication. When entities' identities are verified in accordance with the established security policies, it results in the generation of specific keys to ensure secure authentication; a private key to access the data and its pair, a public key to make transactions.

- Traceability and Transparency: Blockchain provides effective sharing of historical information, thus guarantying traceability and transaction transparency. The decentralized nature of technology creates a transparent profile of every node/participant, as it stores details of every single transaction or event that occurs. Every change on the blockchain is viewable and makes it more concrete. In the area of IoT applications tracing historical data is crucial. For instance, by reviewing data, identifying factors that might impact product quality is possible. By filtering through the data, weak spots in production may be discovered.

- Faster transactions: Blockchain offers a faster settlement compared to other systems. It is very easy to set up a blockchain, and the transactions are confirmed very fast. It takes only a few seconds to a few minutes to process the transactions or events. Another feature is the smart contract system. This can allow making faster settlements for any kind of contract.

2.3 Concepts

In this section, we will analyse the main concepts of the blockchain technology. But before that let's review how a blockchain is formed [14].

In a blockchain, once a full node is connected to its peers, it first tries to construct a complete blockchain. The root of the blockchain is a genesis block, the first block in the blockchain. It is the common origin of all blocks and contains the information that is generally known to all nodes. The block consists of cryptographic hashes of records, with each block holding the information about the previous block's hash, forming a chain of data, and creating a blockchain, as shown in Fig. 2.1.

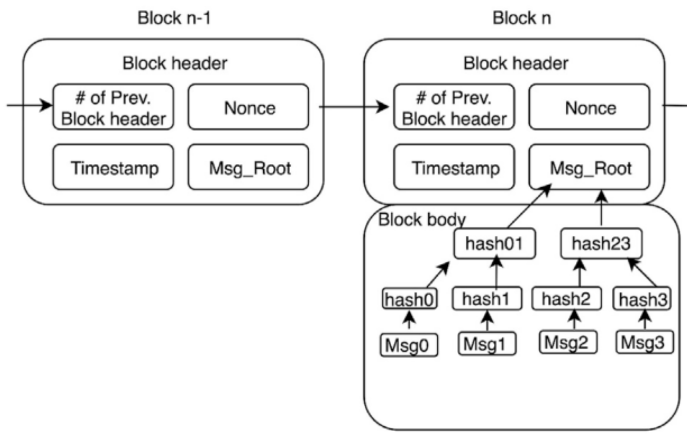
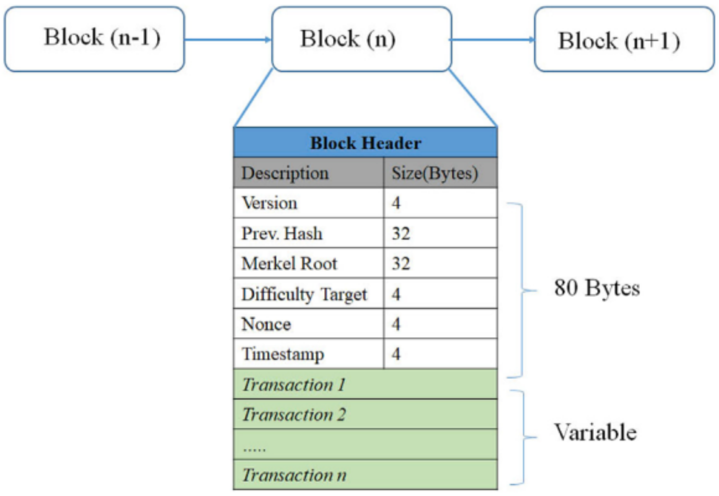


Figure 2.1: *The structure of blocks in a blockchain*

The blockchain begins with a genesis block on top of which stacked the successor blocks. The structure of each block contains a block header and a block body. The block header consists of a previous block's hash, nonce, timestamp, as well as the Merkle root, as shown in Fig. 2.2. The block body contains lists of transactions and some additional data, depending on the requirement of the blockchain. Each current block is interconnected with the previous block, using the hash of the previous block like a chain. For immutability, the transactions should be hashed using a Merkle hash, which needs to be included in the block header. In other words, the blocks are cryptographically verified and chained up to form an immutable chain of blocks called a blockchain or a ledger. More on that in the next sections.



Description	Size(Bytes)
Version	4
Prev. Hash	32
Merkel Root	32
Difficulty Target	4
Nonce	4
Timestamp	4
Transaction 1	
Transaction 2	
.....	
Transaction n	

Figure 2.2: *Block header format*

2.3.1 Cryptography, hashing and digital signature

The blockchain has gained huge popularity due to its security features of using cryptography. As mentioned before, all data in the blockchain is secured by cryptographic hashing and digital signature.

Hashing

As stated in [15], hashing refers to the concept of taking an arbitrary amount of input data, applying some algorithm to it, and generating a fixed-size output data called the hash. The input can be any number of bits that could represent a single character or something infinitely big. The hashing algorithm can be chosen depending on the needs of the project. There are many publicly available, for example, the secure hash algorithm with a digest size of 256 bits, or SHA256 algorithm, is one of the most widely used.

A hash is used to verify that a file has not been tampered with or modified in any way not intended by the creator. Let us assume that someone publishes a set of files along with their MD5 hashes. Then, whoever downloads those files can verify their ownership by calculating the MD5 hash of the downloaded files. If the hash does not match what was published by the creator, then it is certain that the file has been modified in some way.

Hashes are used in blockchain to represent the current state of the blockchain. The input is the entire state of the blockchain, which includes all the transactions that have taken place so far and the resulting output hash represents the current state of the blockchain. The hash is used to reach an agreement between all parties that the blockchain state is one and the same.

The hash of each block in the blockchain is generated by the hash of its own transactions as well as the hash of the previous block. Since the blockchain is formed with each new block hash pointing to the block hash that came before, it is guaranteed that no transaction in the history can be tampered with. That is because if any single part of the transaction changes, so does the hash of the block to which it belongs, and any following blocks' hashes as a result. It would be fairly easy to catch any tampering as a result, because it would suffice to compare the hashes. Furthermore, hashes cannot be reverse-engineered, meaning users cannot make use of the output string for the purpose of finding the input data.

Therefore, everyone on the blockchain only needs to agree on 256 bits to represent the potentially infinite state of the blockchain. For instance, the Ethereum blockchain is currently tens of gigabytes, but the current state of the blockchain, as of this recording, is this hexadecimal hash representing 256 bits.

Digital Signatures

Digital signatures are a way to prove that somebody is who they claim to be, except that cryptography or math is used, which is more secure and cannot easily be forged compared to handwritten signatures [15].

In asymmetric encryption systems, users generate a pair of related keys (one public key and one private key), using some known algorithm, to encrypt and decrypt a message and protect it from unauthorized access or use. The public key and private key are are mathematically connected with each other. This relationship between the keys differs from one algorithm to another. The algorithm is basically a combination of the encryption function and decryption function.

On one hand, the public key is intended for public distribution, to serve as an address to receive messages from other users, like an IP address. On the other hand, the private key is used to digitally sign messages sent to other users and it must be accessible only to the owner of the key pair. The signature is included in the message so that the recipient can verify using the sender's public key. This way, the recipient can ascertain that only the sender could have sent this message. This is how authenticity is ensured, as well as the validity of the message.

Generating a key pair is like creating an account on the blockchain, but without having to actually register anywhere. Also, every transaction that is executed on the blockchain is digitally signed by the sender using their private key. This signature ensures the authenticity of the transactions.

To bring it all together, blockchain could not exist without hashing and digital signatures. Hashing provides a way for everyone on the blockchain to agree on the current blockchain state, while digital signatures provide a way to ensure that all transactions are only made by the rightful owners. We rely on these two properties to ensure that the blockchain has not been corrupted or compromised.

2.3.2 Immutable ledger

The core of the blockchain ledger is the security of data and the proof that data remain unaltered. Therefore, an immutable ledger cannot be tampered with and hence the data cannot be changed with ease, thereby establishing that the security is quite tight. Immutability means that it is very difficult to make changes without collusion. So, blockchain offers an immutable ledger that records every state change in history. In its technical nature offers an immutable database. Hence, it's impossible to manipulate data already in the blockchain afterwards.

As we said before, each of the blocks contains a hash value or digital signature for itself and for the previous one as well, making them retroactively coupled together. This way no one is able to interfere with the system or change the already saved data into the block. Cryptographic hashes and digital signature are among the most important elements that make the blockchain ledger immutable.

In this regard, it is also quite essential to know that blockchain is distributed and decentralized in nature [16]. In order to deal with the different copies of the data, a consensus is made. It is this consensus that makes sure the originality of data is rightly maintained. More will be explained in the next sections.

2.3.3 P2P network

Before we move any further, it is important to understand the role of P2P network in the blockchain [17]. P2P network is a decentralized network communications model that consists of a group of devices, representing the nodes of the system. Each node acts as an individual peer and may communicate with other nodes without the need of a third party. In other words, the exchange of data occurs without a central server, each computer or node can act as both a file server and a client, which means all nodes are equal and perform the same tasks, whereas in the traditional client-server architecture, there is a dedicated server and specific clients.

P2P architecture is suitable for various use cases and can be categorized into structured, unstructured, and hybrid P2P networks. The unstructured P2P networks are formed by nodes randomly connecting with each other, but they are inefficient compared to structured ones, where the nodes are organized, and every node can efficiently search the network for the desired data. Hybrid models, a combination of P2P and client-server models, compared to the previous two systems tend to present improved overall performance.

Blockchain is a decentralized ledger tracking of one or more digital assets on a P2P network. In such a network, all the computers are connected in some way, and each maintains a complete copy of the ledger and compares it to other devices to ensure the data is accurate. Thus, the system is backed up by every single node participating in the network. No centralized authority manages the blockchain networks and only the participant nodes can validate blocks among each other. This new form of distributed data storage and management acts as a digital ledger that publicly records all transactions and events.

Now, there are many benefits in utilizing the P2P architecture in the blockchain. Due to P2P networking capability, the system is always available. Even if one peer gets down, the other peers are still present. Thus nobody can take down the blockchain. Also, P2P networks offer greater security compared to traditional client-server systems. The distributed P2P network, when paired with a majority consensus requirement, gives blockchain a relatively high degree of resistance to malicious activity. More than that, these systems are virtually immune to the Denial-of-Service (DoS) attacks. Finally, the absence of third parties makes these networks more privacy-friendly than centralized networks, as there's no need for a central authority to store or access user data.

P2P network in blockchain, however, raises a few concerns. Distributed ledgers must be updated on every single node, and adding transactions requires a considerable amount of computational power. Although this provides an increased level of security, it significantly reduces efficiency, and this acts as one of the main hindrances in terms of scalability and mass adoption.

2.3.4 Public, Private, Permissioned networks

As already mentioned, blockchain networks can be categorised into public, private or permissioned.

Public blockchains are extraordinarily valuable since more and more users may easily join, keeping the network agile and safe from data breaches, hacking attempts, or other cybersecurity issues. The more participants, the safer a blockchain is. In other words, the network is highly secure, as there are simply too many nodes to allow a cyberattacker to take control of the decentralized network. Also, all transactions are recorded and can't be altered. However, many public blockchains require heavy energy consumption to maintain them. Low throughput is another disadvantage caused by trying to reach consensus with a disparate group of users. Also, since they are public there is the issue of the lack of complete privacy and anonymity.

Private blockchains focus on privacy, operating as a closed database secured with cryptographic concepts and the organization's needs. Only those with permission can run a full node, make transactions, or validate/authenticate the blockchain changes. Also, they prioritize efficiency and immutability as there are a handful of users who need to achieve consensus to validate transactions. Despite the advantages of a faster, more efficient and trusted system, private blockchains also come with disadvantages as well. As they are usually built to accomplish specific tasks and functions, more often than not they are not widely applicable. In this respect, private blockchains are susceptible to data breaches and other security threats, as there is generally a limited number of validators used to reach a consensus, if there is a consensus mechanism.

Permissioned blockchain offers better performance, varying levels of decentralization and customizability. Since permissioned blockchains are closed to the public, they are usually much "lighter" than public blockchains, meaning less data in the chain clogging the network which leads to faster transactions and improved overall performance. The network operators of permissioned blockchains can choose the desired level of decentralization, from partly decentralized to fully centralized, and also the different roles they may give to each participant. The hybrid nature of this chain carries with it the disadvantages of public and private blockchains, depending on how they are configured. One major drawback is that the security of these blockchains relies entirely on the chosen consensus algorithm and participants, which in case of malicious nodes, can compromise the entire network.

2.3.5 Consensus Algorithms

From what we reviewed, we know that blockchain is a distributed decentralized network that provides immutability, privacy, security, and transparency. There is no central authority present to validate and verify the transactions, yet every transaction in the blockchain is considered to be completely secured and verified. Furthermore, all entities have the same copy of the ledger, a single history of blocks. This is possible only because of the presence of the consensus protocol which is a fundamental component of every blockchain.

A consensus algorithm is a procedure that ensures an agreement is reached between participants to support decision making. In other words, a protocol which all the nodes in the blockchain network use in order to come to a common consensus on the current data state in the ledger and whether or not to trust unknown peers in the network. With this mechanism, for every new block added to the blockchain a decision needs to be made and must be agreed upon by all the nodes in the blockchain. That is, if the block is the one and only version of the truth. Essentially, consensus algorithms provide a set of rules that enable the addition of new blocks to the chain while protecting the network against attackers. Blocks cannot be added to the chain until they are validated. Thus, a consensus algorithm aims at finding a common agreement that is a win for the entire network.

How consensus is reached could easily impact the security and the performance of the blockchain network. In the current literature, several approaches were developed to reach consensus [18], [14], [19], among them:

- Proof of Work (PoW): In the proof of work algorithm, customary in public blockchains, the validators or miners or node participants, need to prove that the work they have done and submitted gives them the right to add new transactions to the blockchain network. To achieve this, miners must compete against each other to solve a computer problem –calculate a nonce value to meet certain value requirements of the block hash– with a difficulty d, on the new block before approving it to the ledger. After that, the solution is forwarded to other validators and verified by them before accepting the copies of the ledger. A PoW consensus algorithm is a cryptographic puzzle that is very hard to solve, but once all inputs are known, it is easy for others to verify. Using PoW verification eliminates duplicate transactions in the blockchain network,as they will be seen in the system and will not be accepted. Hence, no one can change the transaction once it has been verified and approved by every node participant.

- Proof of Stake (PoS): This is the most common alternative to PoW, popular in public blockchains. In this type of consensus algorithm, instead of investing in expensive hardware to solve a complex puzzle, validators invest in the currency of the system by locking up some of their currency as stakes. The miners are required to stake their assets, by placing a bet if they discover a block that they think can be added to the chain, and validate ownership without being required to prove the legitimacy of each transaction. Based on the actual blocks added in the blockchain, all the validators get a reward proportionate to their bets and their stake increase accordingly. Meanwhile, the more currency forgers exist, the greater chance they have to generate the next block. In the end, a validator is chosen to generate a new block based on its economic stake in the network. Thus, PoS encourages validators through an incentive mechanism to reach to an agreement. This approach is introduced to reduce energy and time. It is further categorized into Byzantine Fault Tolerant-based Proof-of-Stake (BFT-PoS) and chain-based PoS.

  - In BFT-PoS protocol, the validators keep a full copy of the blockchain and are identified by their public keys. However, the majority decides, i.e., 2/3 of all validators, whether or not to approve the proposed block. This may take several rounds before the block gets finalized. Simply put, a BFT system is used to fix the problem of unreliable nodes in the network, as it can continuously operate even when nodes act maliciously or fail.
  - In the chain-based PoS, a validator is chosen at random and given an opportunity to create a block, as long as they own currency on that particular chain. They are selected at random using a specific set of criteria; if they don't produce a block within some time limit or have some other rules, they are replaced with another randomly selected participant. Then, the new block is linked with the hash of the previous block of the longest chain. A block is finalized when there is no chance of it being revised. The validators are required to vote and sign their votes before propagating it in the network, using a single type of message, i.e., vote, which combines the roles of preparing and committing.

- Practical byzantine fault tolerance (PBFT): This consensus strategy, commonly used in private blockchains, ensures a consensus regardless of malicious node behaviors on the part of some participating nodes. The algorithm used is made to tolerate byzantine failures. All nodes interconnect with each other, and the legitimate nodes contribute to reaching a consensus regarding the state of the system using the majority rule, meaning that a validator requires the consensus of the remaining nodes to generate new blocks in the chain. The assumption here being that the sum of malicious nodes cannot exceed the 1/3 of the overall nodes in the network. That said, the more the nodes join the PBFT network, the more secure the network is.

- Proof of Elapsed Time (PoET): This is a private blockchain consensus mechanism that needs all participating nodes to identify themselves before they participate in the network. It requires the nodes to wait for a random time period before achieving the consensus on a new block. The idea behind PoET is based on a fair lottery system. For the nodes to win the lottery, they need to select a short random time and have to complete certain waiting time. So, whenever a block is available and the node is activated, that node gets the authority to share the information on the network and earn rewards.

- Proof of Authority (PoA): This consensus algorithm uses the value of identities, so the block validators don't stake currency but their reputations. Since it depends on the limited number of validators, it is used in private blockchains and it makes the system highly scalable while maintaining privacy.

- Raft: This system consists of usually five server nodes. Two nodes are allowed to fail at the same time. The server node has three states: leader, follower, and candidate. Usually, there is only one leader who is responsible for handling all of the client requests while other servers are followers. The third state elects a new leader. A candidate receiving votes from the majority of the cluster now becomes the new leader of the consensus mechanism.

- RR (Round Robin): Here, generation of a valid blockchain is achieved through permitted entities creating blocks in rotation. Every entity in a given time window can only create a finite number of blocks calculated using a network parameter called mining diversity that determines the number of blocks that should be wait for before attempting to mine again.

A misjudgement in the selection of a consensus algorithm may lead to compromised data being recorded on the blockchain. Below are some of the issues that can result when the consensus mechanism fails [18].

- Blockchain Fork: This happens whenever the chain splits, producing a second blockchain that shares all of its history with the original, but is headed off in a new direction. It can result in different nodes in the system converging on different blocks as being part of the blockchain. In some cases temporary forks may naturally exist, as the protocol is designed such that all nodes will eventually converge on a single chain. In other cases, forks can be catastrophic as they can lead to completely inconsistent view of data recorded on the blockchain thereby forcing applications to behave in an unpredictable manner.

- Consensus Failure: When the consensus algorithm chosen for the system cannot guarantee reaching consensus, a consensus failure occurs. For example, in case of a majority rule requirement, if the majority cannot be secured because of node or network failures, non-compliant nodes or as a result of valid honest nodes not being able to make a decision due to inconsistent messages received from other nodes, it will most certainly lead to a consensus failure.

- Dominance: If for any reason, one or a handful of malicious nodes can control millions of identities, consensus outcomes can be manipulated. Having such dominance allows the dominating group to confirm the transactions and blocks as per their rules.

- Cheating: In the event the consensus algorithm selected leaves room for validating malicious nodes which can independently maintain parallel forks in the blockchain of fraudulent transactions or alter reality, then that algorithm has failed.

- Poor Performance: Depending on the design of the consensus algorithm, it may require more time under certain conditions for consensus to converge. This issue may arise when other nodes have turned malicious or a network partition delays messages that are exchanged between nodes, etc. This may manifest as inconsistently high latencies in applications.

In conclusion, there is not one algorithm that surpasses the others. It all depends on the platform that it's going to be used in and the type of functionality the platform needs to provide along with its timidness to gain integrity. All these algorithms have the same purpose; to reach an agreement (consensus) in the decentralized blockchain network in order to verify and validate the blockchain authenticity.

Most protocols and their implementation are still subject to a thorough, peer-reviewed correctness and reliability analysis as there might be several vulnerabilities, security issues and protocol weakness. In respect of security, till now, the PoW is considered to be the most effective consensus mechanism although it has some flaws, such as scalability, transaction finalization, etc.

2.3.6 Smart Contracts

Smart contracts are self-executing programs stored on a blockchain that run automatically when specific conditions are met. They are used to automate the execution of agreements between parties, eliminating the need for intermediaries and reducing the time it takes to complete transactions. The contracts work by using "if/when...then..." statements written into code, and are verified and executed by a network of computers. The terms and conditions of the contract are established by the participants and can be customized to meet their needs. The contract can be programmed by a developer, or made easier to use with templates, web interfaces, and online tools provided by organizations using blockchain for business. The outcome of the contract is recorded and updated on the blockchain, and cannot be changed, ensuring the transparency and security of the transaction.

2.3.7 Blockchain Formation

By now we have mentioned the validity of the blockchain and the validation of blocks several times, yet haven't explained in great detail how it is achieved and the process that is followed. To reiterate how the blockchain is formed, first a node records new data values and broadcasts them to the network where the receiving nodes verify the data and store them in a block. Then all nodes in the network perform the mining process, where the consensus algorithm is used to reach consensus in order to add the block to the blockchain. Finally the blockchain is updated for all nodes in the network.

A node's records are transactions and related data. A transaction refers to a contract, agreement, transfer, or exchange of assets between two or more parties. In simple terms, a transaction is nothing but data transmission across the blockchain network. On a number of occasions, to perform transactions on the blockchain, a wallet is needed, a program linked with the blockchain to which only the node who owns the wallet has access, that keeps track of the node's data. Each wallet is protected by a private and a public key. With their wallet, a node (whoever has the private key) can authorize or sign transactions and thereby transfer value to a new owner. The transaction is then broadcast to the network to have its validity confirmed for the purpose of being included in the blockchain.

After a transaction takes place it needs to be selected to be part of a block that is going to be added to the blockchain. As long as it is not picked up, it hovers in a 'pool of unconfirmed transactions' [20]. This pool is a collection of unconfirmed transactions on the network that are waiting to be processed. These unconfirmed transactions are usually not collected in one giant pool, but more often in small subdivided local pools.

Then, the mining process needs to take place. That means miners on the network select transactions from these pools and form them into a 'block'. A block is basically a collection of transactions (at this moment in time, still unconfirmed transactions), in addition to some extra metadata. Every miner constructs their own block of transactions. Multiple miners can select the same transactions to be included in their block.

The mining process can be either done individually (solo mining) or collectively (pool mining) [21]. When mining is done by an individual, user registration as a miner is necessary. As soon as transactions are selected to be included in a block, the first one satisfying the chosen consensus tries to add it to the blockchain. All the other miners in the blockchain network will validate the block and then add it to the blockchain. Thus, verifying the transactions. On the other hand, in pool mining, a group of users works together to approve the transactions picked up from the pool of unconfirmed transactions. After the validation of the result, the reward is then split between all users.

Despite how the process is handled, solo or not, by selecting transactions and adding them to their block, miners create a block of transactions. To add this block of transactions to the blockchain, having all the nodes on the blockchain register the transactions in this block that is, the block must be validated. This means that every transaction in the block must also be verified by checking the digital signatures of the parties involved in the transaction, as well as the validity of the transaction itself. For instance, in an exchange fund transaction, the transaction will be rendered invalid if one party does not possess the amount of funds that need to be exchanged.

There are a number of different consensus mechanisms that can be used to validate a block, and each has its own strengths and weaknesses, as explained in previous sections. No matter which one is selected, in the end, after reaching consensus, a block of transactions will be added to the blockchain.

Every new block added on top of the block that has just been added to the blockchain counts as 'confirmation' for that block. It counts as a confirmation because every time another block is added on top of it, the blockchain reaches consensus again on the complete transaction history, including the transaction and the block in question. Essentially, it means that the transaction has been confirmed as many times as the blocks added after it at that point. The more confirmations the transaction has, or else the deeper the block is embedded in the chain, the harder it is for attackers to alter it.

Succeeding the addition of the new block to the chain, all miners need to start all over again by forming a new block of transactions. Miners cannot continue mining of the block they were working on earlier because of two reasons. Firstly, it may contain transactions that have been already confirmed by the last block that was added to the blockchain as multiple miners can include the same transactions in the block they are working on. Any of those transactions initiated again could render them invalid, because the conditions of this transactions have changed. And secondly, since every next block needs to include the hash of the previous block into its own data, if a miner keeps mining the same block other miners will notice that the hash output does not correspond with that of the latest added block on the blockchain, and will therefore reject the block.

Merkle Trees

Merkle trees, also referred to as binary hash trees, serve to encode blockchain data more efficiently and securely. The Merkle hash is derived from the Merkle algorithm [22], which is a cryptographic algorithm that hashes all transactions of the block to get the Merkle root.

The Merkle tree has a binary tree structure with the leaf nodes as the hashes of transaction in a block, also known as transaction IDs (TXIDs), the intermediate hashes, the non-leaf nodes that are hashed together in pairs and store the hash of the two leaf nodes they represent, and, finally, the root as the hash that combines the hashes of all transactions. This is depicted in Fig. 2.3. The Merkle root is what is eventually appended in the block header. The leaf nodes (transaction IDs/hashes) at the base of the Merkle tree can be verified using the Merkle root.

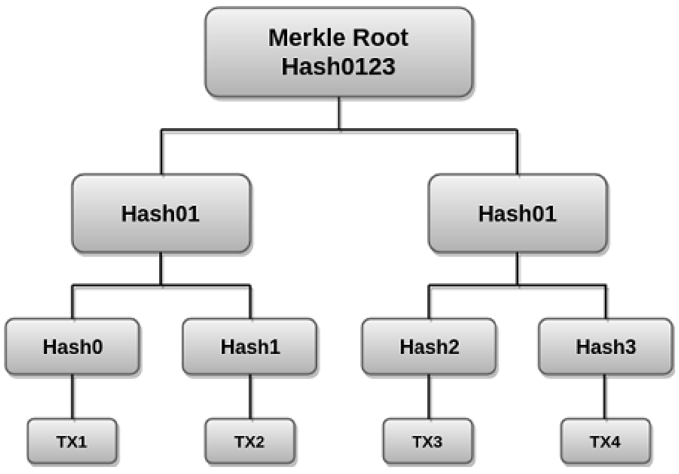


Figure 2.3: *Merkle Tree*

The benefit of Merkle tree is that it is able to break large pieces of data into considerably smaller chunks, ensuring that all the transactions can be verified promptly. The unique hash value it generates is used to verify the integrity of all transactions underneath it, and the size of the Merkle hash is very small as compared with the whole size of all transactions. As a result of its functionalities, Merkle trees are utilized more effectively and securely to encrypt blockchain data.

Chapter 3

Related Work in Blockchain Based Trust Management

In order to cope with the problems of centralization mentioned before, decentralized systems are often chosen for trust management. More and more so blockchain-based or assisted systems, which is what we hope to utilise in this work. Some of the research done on the matter will be presented below in order to give us an idea of that which we are trying to achieve.

3.1 Blockchain in Vehicular Network

In [23], the authors review a decentralized trust management system in vehicular networks based on the blockchain. Much like a MANET, the vehicular network consists of vehicles, mobile nodes, that generate and broadcast messages with ultimate goal to improve traffic safety and efficiency. Here, vehicles can validate the received messages from neighbouring vehicles using Bayesian Inference Model and based on the outcome generate a rating for the vehicle that send that message. Roadside Units (RSU's) use this ratings to calculate trust value offsets of the involved vehicles and insert these data into a block. With a joint PoW and PoS consensus mechanism, each RSU will attempt to add their blocks to the blockchain. Meaning that the higher the total value of trust value offsets is in the block - representing the stake here - the easier it will be for the RSU to find the hash function - representing the PoW. All RSUs cooperate to maintain an up-to-date and consistent, among all nodes, blockchain. Thus, all RSU participate in the network and all RSU's are provided with trust information of all the vehicles in the network.

In more detail, RSUs are responsible for collecting ratings of messages sent by vehicles and using them to calculate trust values for each vehicle. These trust values represent the historical credibility of the vehicle's messages and can be queried by other vehicles. The ratings are collected by RSUs, as vehicles are not able to store them locally long term due to fast-changing traffic environments and limited capacity of on-board devices.

On the other hand, vehicles use on-board devices to automatically detect traffic-related events and send warning messages to others using communication standards. However, not all messages are useful, so each vehicle maintains a reference set of neighbouring vehicles considered relevant to its traffic safety. These messages from the reference set provide insight into the current traffic conditions, but they may not always be trustworthy. Therefore, the vehicle needs to use specific models to aggregate the messages and determine which are credible, using a majority rule for example. Then, the vehicle generates ratings for the messages based on their credibility and uploads them to the RSUs.

Yet, vulnerabilities are found in both vehicles and RSUs. Two types of adversaries are considered: malicious vehicles and compromised RSUs. Malicious vehicles may deliberately broadcast fake messages (message spoofing attack) or generate unfair ratings on messages to degrade traffic safety or efficiency (bad mouthing and ballot stuffing attack). Compromised RSUs may have their data tampered with by attackers, but this is only a temporary and limited problem due to frequent security checks and the limited capacity of attackers.

As shown in Fig. 3.1, vehicles assess the credibility of received messages and generate ratings for them, which are then uploaded to an RSU. The RSU uses these ratings to calculate trust value offsets for each vehicle and packs them into a candidate block. The RSU then uses a joint PoW and PoS miner election scheme, where the sum of absolute values of offsets in the candidate block is used as the stake, to add the block to the blockchain. This ensures the timely update of trust values in the blockchain, and allows the RSU to obtain real-time trust values for each vehicle.

When the RSU receives a block from the miner, before adding it to the blockchain its validity of the nonce must be checked. When the RSU receives more than one block at the same time, the blockchain starts to fork. Each RSU chooses one fork and continues to add new blocks to it, with the fork acknowledged by the largest number of RSUs growing faster than others. Eventually, the longest fork becomes the consensus of the network, and the other forks are discarded. Each RSU then collects blocks from discarded forks and tries to add them to the blockchain in the future to ensure consistency among all RSUs.

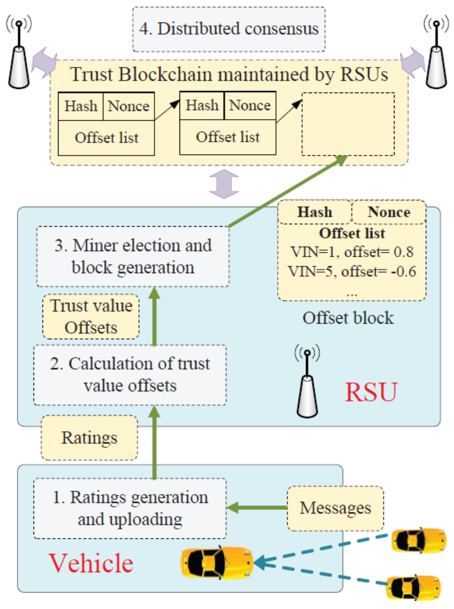


Figure 3.1: *System design of blockchain-based decentralized trust management*

As far as the security of this system is concerned, there are methods of defending against malicious vehicles and compromised RSUs in a vehicular network that utilizes blockchain technology. For the defense against malicious vehicles, the system uses a Bayesian Inference-based rating generation scheme to thwart message spoofing attacks and limit the effects of bad mouthing and ballot stuffing attacks. For the defense against compromised RSUs, it is assumed that only a small portion of RSUs may be compromised for a short period of time. Due to the use of blockchain techniques, where all RSUs store the same version of the blockchain, compromised RSUs can be detected by their differences from others. The system also includes a mechanism to prevent compromised RSUs from generating too many fake blocks by limiting the upper bound of the stakes in each block.

Similar technology is used in [24].

3.2 Multichain for IoT devices

In [13], a secure trust management system based on blockchain technology is proposed. The system aims to provide benefits such as tamper-proof data, improved trust information integrity verification, and enhanced privacy and availability during trust information sharing and storage. The system includes a blockchain-based trust management architecture to collect trust evidence, establish trust scores for devices, and securely store and share trust information with other devices in the network through blockchain transactions. Essentially, it aims to assign trust scores to devices, store and share those scores securely while ensuring transparency, integrity, authenticity, and authorization.

Further reading shows that the system is composed of a number of manufacturing zones, each consisting of physical resources, an authentication manager, and a trust manager, as shown in Fig. 3.2. Miners are also deployed to receive trust data, create blocks and broadcast them in the blockchain network. The architecture of the system is detailed in the paper, with interactions between the different modules presented. The design only considers one zone, so interactions between devices belonging to different zones are not considered.

The proposed system includes the device layer which consists of IoT devices that gather and process information, and also execute additional tasks related to trust management. These devices assess the behavior of other devices they interact with, using metrics such as packet delivery ratio and community of interest, to determine a trust score. This trust score is then sent to a trust manager entity and added to the blockchain network for secure storage and sharing with other devices within the system. This allows devices and industrial applications to make decisions and obtain data in a reliable way based on the trust scores of other devices.

The system management layer is responsible for ensuring the security and reliability of the proposed trust management system. It performs computations and verifications on trust-related information, stores trust and reputation scores securely in a decentralized structure, and manages access to trust data. It also guarantees the integrity and validation of actions through a network of consensus entities, authenticates devices and manages their access to trust data.

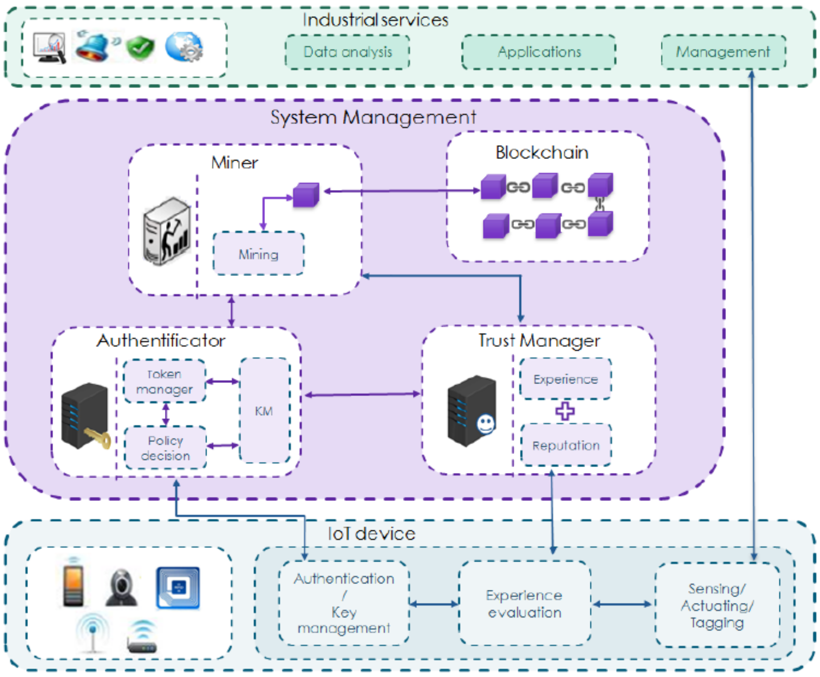


Figure 3.2: *System design of blockchain-based decentralized trust management for IoT*

One of the entities in the system management layer is the trust manager, who is responsible for creating a secure environment for devices to interact with one another and with industrial IoT services by assessing trust scores. Trust is defined as a relationship between a trustor and a trustee, which is limited to a specific time and based on direct observations and recommendations from neighbours. The entity uses a cyclical process to collect trust-related information, calculate trust scores for each entity, and produce an overall trust value for the system. The trust scores are based on properties such as cooperativeness, competence, and community of interest, and take into account past trust evaluations to account for changes in behavior over time.

Another entity in this layer is the authenticator, who is responsible for verifying the validity of devices' identities and the legitimacy of requests sent to trust data storage and management systems. The framework uses the openID Connect protocol, which allows for clients to verify the identity of a device and obtain basic profile information in a REST-like manner. The entity has two subcomponents: the Policy Decision Component (PDC) and the Key Management Component (KMC). The PDC is in charge of making authorization decisions based on policies and generating access tokens for successful authentication processes. The KMC generates authentication and transaction-related keys to authenticate devices and digitally sign actions on trust scores. These keys include a transaction private key, used to sign requests for trust data access, and a transaction public key, used to verify the requester's identity and encrypt packages sent to the requester device.

Last but not least, the miner entity that processes and verifies the authenticity, validity, and integrity of trust records and transactions. These records are broadcast to a network of miners who check their validity and package them into blocks which are added to a ledger. Multichain is used, a private blockchain protocol that manages access to the blocks using a list of registered participants and uses the Round Robin algorithm for approving transactions. The choice of Multichain is due to its characteristics, such as its private, permissioned nature, flexibility, ability to change permissions and delegations, and use of the Round Robin consensus mechanism which does not require complex computation resources. Additionally, it is based on streams that act as an independent append-only collection of items, which enforces data confidentiality, and does not require the use of currencies or computational power, like some other blockchain solutions.

The necessary message exchange and interactions that occur among different entities in the proposed system can be illustrated through a scenario that considers an IoT device attempting to store trust scores on a blockchain network. The process is split into three main stages. In the first stage, the device must have valid authorization credentials to access the system, and if it does not, it contacts an authenticator entity to obtain them. During this stage, the authenticator entity makes authorization decisions based on policies defined in the PDC. In the second stage, the device monitors the behavior of its neighbours and assesses their trustworthiness level, resulting in an experience score. This score is signed with a transaction private key and sent to the trust manager entity. In the third stage, the trust manager entity computes a reputation score and overall trust score, then sends the transaction to the miners, who check its validity and add it to the existing ledger.

3.3 TrsutChain for IoT supported Supply Chains

In [25], TrustChain is proposed, which is a three-layered trust management framework that uses a consortium blockchain to track interactions among supply chain participants and dynamically assign trust and reputation scores based on these interactions. The key features of TrustChain include: a reputation model that evaluates the quality of commodities and the trustworthiness of entities based on multiple observations of supply chain events, support for reputation scores that separate between a supply chain participant and products, use of smart contracts for transparent, efficient, secure, and automated calculation of reputation scores, and minimal overhead in terms of latency and throughput compared to a simple blockchain-based supply chain model. The framework is organized into three layers, as seen in Fig. 3.3: data, blockchain, and application, with smart contracts used to automate the process of generating reputation and trust values. TrustChain is implemented on a blockchain-as-a-service platform and uses Hyperledger Fabric for deployment, due to its support for business-related applications and ease of deployment. The system assumes that entities maintain a static public key for identification and use it to sign transactions.

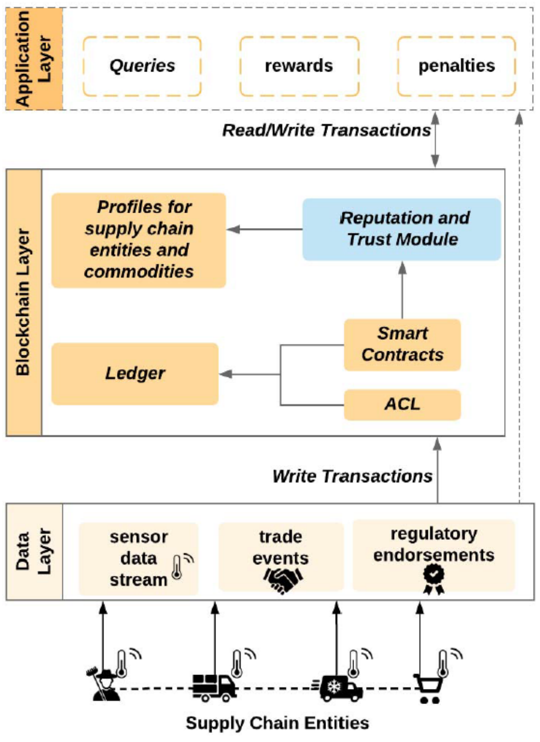


Figure 3.3: *Structure of the TrustChain Framework*

The data layer of TrustChain is responsible for collecting and processing data inputs from various sources such as sensor data streams, trade events, and regulatory endorsements. The raw data collected at the data layer can be stored in a database (off-chain) to improve scalability. Sensor data streams from IoT sensors installed at different supply chain entities such as primary producers, retailers, etc. are used to monitor the quality of food products. Temperature sensors are used as an example, and the commodity is given a rating based on the temperature readings. Warning messages can also be generated if the reported sensor temperature is out of bounds of the desired range. Trade events, such as a change of ownership of a commodity, are recorded on the blockchain, along with a rating attributed by the buyer for the seller. This rating is based on a quality assessment of the traded commodity. Regulatory endorsements, such as certificates and reports generated by food safety authorities, are also used to generate ratings. All these ratings are used by the trust and reputation module to calculate reputation and trust values for the supply chain entities and commodities.

The blockchain layer consists of the transactions, the smart contracts and the trust module. Transactions invoked at the data layer include the create transaction (TXcr), the trade transaction (TXtr), the sensory transaction (TXsens), the regulator transaction (TXReg), and the receipt of commodity transaction (TXrec) to record supply chain events, changes in the state of entities and commodities on the ledger. These transactions are governed by an Access Control List (ACL) which defines the permissions for submitting transactions, read/write access to the ledger, updating profiles and other actions for supply chain entities.

Additionally, these transactions invoke smart contracts which are used to automate the reputation calculation and update the overall rating of the commodity throughout the product chain. This information is made visible to consumers via the application layer for transparency and traceability in the supply chain. Quality Contracts are smart contracts that are installed for each supply chain commodity and specify the quality rating criteria, such as temperature thresholds, for the commodity. These contracts also generate warning notifications and reputation scores for the commodity based on temperature inputs. The Rating Contract is another type of smart contract that is invoked to compute the reputation of a seller based on inputs such as the reputation score of the commodity, the regulator's rating, and the buyer's rating. These contracts help to ensure that the calculation of ratings is transparent, secure, efficient, and automated, eliminating the need for intermediaries.

The reputation and trust module in TrustChain uses smart contracts to calculate reputation scores for supply chain entities and commodities based on supply chain events recorded on the blockchain. The reputation model uses an aggregation function and a time-varying, amnesic trust score calculation that adapts to recent events and gives them higher weight than older events. When a trader joins the network, they are assigned an initial trust score that must be maintained to participate. The trust score is updated by the reputation and trust module and involves calculating the overall reputation score and the trust score based on the reputation score and other application-specific features. The overall reputation score considers current and previous supply chain events and uses a forgetting factor to give more weight to recent events. The reputation of a trader for trading a single type of commodity is stored separately for multiple types of commodities and can be calculated periodically by network administrators.

The application layer of TrustChain addresses queries and transaction requests from administrators, regulators and consumers. It uses smart contracts to calculate ratings for entities and commodities in a transparent, secure and automated way. The application layer also includes rewards and penalties mechanisms to motivate supply chain entities to contribute trustworthy data to the network. Queries can include transactions to read or write data from the blockchain, and rewards can include publishing entities with high trust scores on the network. Penalties can include revoking entities from participating in the network for a certain period of time or publishing a list of revoked participants on the network.

In respect of security and privacy TrustChain may be able to fend off various types of attacks that can be made against a reputation system. These attacks include entities faking data, modifying sensor feeds, creating false commodities, ballot stuffing to raise the reputation of the malicious trader, bad mouthing to lower the reputation of honest traders, among others.

3.4 Blockchain using Mobile Edge Nodes

In [26], a blockchain-based trust management mechanism (BBTM) is proposed, where trustworthiness of sensor nodes is evaluated by mobile edge nodes. BBTM uses smart contracts for trust computation and verifies the computation process. The proposed method is analyzed for its trust accuracy, convergence, and resistance against attacks. The effectiveness of the proposed method is also demonstrated through experimental results, and it is also compared to other methods.

The framework includes 4 roles: sensor nodes, mobile edge nodes, miners, and BBTM clients. Sensor nodes perform collaboration services and applications, while mobile edge nodes have stronger computing and storage abilities and are responsible for evaluating the trustworthiness of sensor nodes within their radius by collecting feedbacks. Miners add past transactions into blocks and verify them for a reward, while BBTM clients are middleware that run locally on IoT nodes and have the ability to publish, receive, and finish trust computation tasks, and create smart contracts for trust computation. The trustworthiness of sensor nodes is calculated by mobile edge nodes using the subjective logic method, and feedback scores are written into the blockchain after the collaboration is completed. This allows for a decentralized, automated, and secure trust evaluation process in IoT environments.

This BBTM framework is divided into two layers, as shown in Fig. 3.4: the BBTM client layer and the blockchain layer. The BBTM client layer includes modules for publishing, receiving, and finishing trust computation tasks, a blockchain wallet for key management, smart contract creation, and a user interface for transactions. The blockchain layer is responsible for executing and verifying transactions and storing trust computation results. The protocols used in the blockchain layer include consensus protocols such as PoW and P2P network protocols for connecting nodes directly to each other in a flat network topology. The goal of the framework is to provide a more accurate trust evaluation method for sensor nodes in the presence of untrustworthy recommendations.

Smart contracts are used to depict the agreement between sensor nodes and mobile edge nodes for the process of trust computation task publishing, receiving, finishing, and reward assignment. The main type of smart contract is the Working Contract (WC) between the sensor node and mobile edge node. This contract contains functions for checking a mobile edge node's trust computation capacity, computing the trustworthiness of collaboration partners, evaluating the trust computation outcomes, and punishing sensor nodes for giving out malicious feedback scores. Additionally, there are two other types of smart contracts, the Mobile Edge Node Management Contract (MENMC) and Sensor Node Management Contract (SNMC), which store information about the node's resources and performance. These smart contracts can be associated with the same task ID and can be reused for the same collaboration type to increase efficiency.

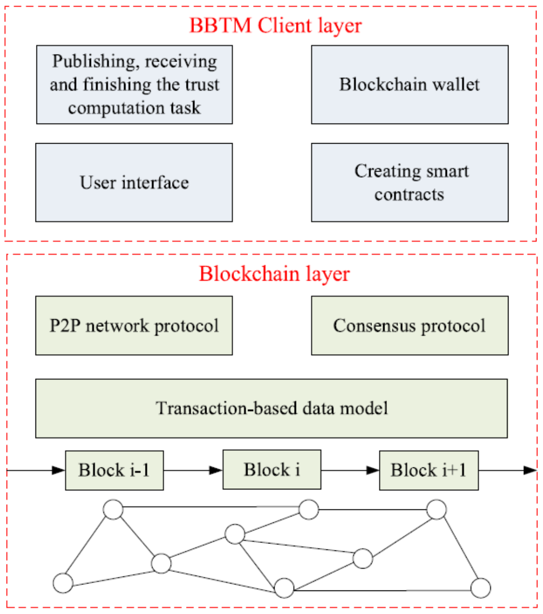


Figure 3.4: *BBTM with mobile edge nodes architecture*

BBTM aims to prevent malicious behavior from sensor nodes and mobile edge nodes. Malicious sensor nodes may aim to obtain trust computation outcomes without rewards, while malicious mobile edge nodes may attempt to obtain rewards without putting in enough effort. The security of BBTM relies on the honesty of most nodes, and it assumes that sensor nodes can bring back their rewards for mobile edge nodes and miners if they get effective trust computation outcomes, and mobile edge nodes and miners can only obtain rewards if they submit trust computation outcomes on time or contribute high-quality outcomes. However, the security of the underlying blockchain is also a concern, as malicious miners may collude to validate invalid blocks. Additionally, the framework is best suited for networks with a large number of IoT devices, and the weaknesses of the consensus algorithm present challenges for its practical implementation.

Chapter 4

Hyperledger Fabric

4.1 Overview

Hyperledger Fabric [27] is a blockchain platform that is designed to be used in enterprise contexts. It is built on a highly modular and configurable architecture, which allows for versatility and optimization for a wide range of industry use cases. The platform is permissioned, meaning that participants are known to each other and can operate under a governance model based on existing trust between them.

One of Fabric's key technical differentiators is its support for smart contracts written in general-purpose programming languages such as Java, Go, and Node.js, rather than in a constrained domain-specific language. This makes it more accessible to developers and eliminates the need for additional training. Fabric also supports pluggable consensus protocols, which allows for customization of the platform to fit specific use cases and trust models. This is particularly useful for situations where a fully Byzantine fault-tolerant consensus might be considered excessive, such as when deployed within a single enterprise. Another important aspect of Fabric is that it does not require a native cryptocurrency, which reduces risk and attack vectors. Instead, it leverages consensus protocols that do not involve cryptographic mining operations, making it more cost-effective to deploy and operate.

Overall, the combination of these technical features makes Fabric a platform that offers high performance and low latency for transaction processing and confirmation, and provides privacy and confidentiality for transactions and smart contracts.

4.2 Features

The main features of Hyperledger Fabric are described below:

- Modularity: Hyperledger Fabric has been designed to have a modular architecture to meet the needs of diverse enterprise use cases. The platform has a pluggable design, allowing for customization of key components such as consensus, identity management, key management, and cryptographic libraries. The main components of Fabric include a pluggable ordering service for establishing consensus, a membership service for associating entities with cryptographic identities, a gossip service for disseminating blocks, chaincode smart contracts that run within a container environment, a ledger that can support various DBMSs, and a pluggable endorsement and validation policy enforcement. With its modular design, Fabric can be configured in multiple ways to meet the needs of different use cases.

- Permissioned Blockchain: In a permissioned blockchain, access is restricted to a pre-authorized set of participants and the consensus mechanism is designed to be faster and more efficient compared to permissionless blockchains. The identities of participants are known and the network operates under a governance model that provides a degree of trust. This results in a lower risk of malicious activity and easier identification of bad actors compared to permissionless blockchains. Permissioned blockchains also allow for the use of more traditional consensus protocols (e.g. BFT), making them faster and less resource-intensive. However, the trade-off is a lack of decentralization and reduced censorship resistance compared to permissionless blockchains.

- Smart Contracts: Smart contracts, also known as "chaincode" in Fabric, is a trusted and distributed application that gains its security/trust from the blockchain and the consensus among the peers. Smart contracts in most blockchain platforms follow the order-execute architecture where the consensus protocol validates and orders transactions before they are executed sequentially by all peer nodes. However, since many smart contracts run concurrently, there is a risk of non-determinism, which can lead to issues with consensus. To address this, many platforms require smart contracts to be written in domain-specific languages to eliminate non-determinism, but this could hinder widespread adoption. Furthermore, the sequential execution of transactions by all nodes can limit performance and scale. Complex measures must be taken to protect the system from potentially malicious contracts and ensure the overall system's resiliency.

- Execute-order-validate architecture: Unlike the traditional order-execute model, Fabric executes transactions before reaching final agreement on their order. This allows for parallel execution and increased performance and scale of the system. Fabric also eliminates non-determinism by filtering out inconsistent results before ordering, making it possible to use standard programming languages. The endorsement policy, which specifies the peer nodes responsible for endorsing a transaction, helps maintain the security and trust of the system.

- Privacy and Confidentiality: The lack of confidentiality in public, permissionless blockchain networks can be problematic for certain business and enterprise use cases. Approaches such as encryption and zero knowledge proofs are being explored to address this issue, but they come with trade-offs such as risk of data compromise or performance impact. In a permissioned context, such as Hyperledger Fabric, confidentiality is achieved through channels and private data features, where a sub-network of authorized nodes have access to the smart contract and transaction data, preserving their privacy and confidentiality.

- Pluggable Consensus: In Fabric, the transaction ordering is handled by a separate component called the ordering service. This design allows the platform to have flexibility in choosing the type of consensus algorithm that best fits the needs of a particular deployment or solution. The ordering service can be either crash fault-tolerant (CFT) or byzantine fault-tolerant (BFT) based on the trust assumptions of the network. Fabric currently provides a CFT ordering service implementation based on the Raft protocol and the etcd library. The platform also supports having multiple ordering services for different applications or requirements within the same network.

- Performance and Scalability: Hyperledger Fabric is designed to address these performance challenges by providing a scalable, flexible, and secure platform for building blockchain applications. The execute-order-validate architecture and modular consensus design enable the platform to achieve high levels of performance by separating the transaction flow into multiple phases, reducing non-determinism and allowing for parallel execution. Additionally, the channel architecture and private data feature enable network participants to transact and store confidential information, further improving performance by reducing the amount of data transmitted and stored across the network. However, as with any system, the specific performance of a Hyperledger Fabric network will depend on the specific use case and deployment configurations.

All in all, Hyperledger Fabric is a highly scalable system for permissioned blockchains that supports a wide range of industry use cases due to its flexible trust assumptions. It is the most active of the Hyperledger projects with a growing community and continuous innovation.

4.3 Concepts

Hyperledger Fabric is a private and permissioned blockchain platform within the Hyperledger project. This project, initiated by the Linux Foundation, aims to promote cross-industry blockchain technologies through a collaborative community approach. Hyperledger Fabric enables participants to manage their transactions with a ledger, smart contracts and pluggable options, such as multiple data storage formats, customizable consensus mechanisms, and support for different Membership Service Providers (MSPs). This blockchain platform also provides the ability to create channels, allowing groups of participants to establish separate ledgers of transactions, ensuring privacy in a Business-to-Business (B2B) network. The ledger subsystem in Hyperledger Fabric comprises two components: the world state and the transaction log, both of which are accessible to every participant in the network. Smart contracts in this platform are written in chaincode and are invoked by external applications when they need to interact with the ledger. The degree of privacy offered by Hyperledger Fabric can vary, depending on the requirements of the network. The consensus mechanism used in the platform is flexible and can be customized to suit the relationships between participants in the network.

4.3.1 The Model

Hyperledger Fabric is a comprehensive and customizable enterprise blockchain solution that offers several key design features to fulfill its promise. These features include assets, chaincode, ledger features, privacy, security and membership services, and consensus.

Assets in Hyperledger Fabric can be anything with monetary value, from real estate to contracts and intellectual property. They are represented as key-value pairs and their state changes are recorded as transactions on a channel ledger. Chaincode is the business logic that defines the assets and the instructions for modifying them. Chaincode functions are executed against the ledger's current state database and are initiated through a transaction proposal.

The ledger is a tamper-resistant record of all state transitions in the fabric, which result from chaincode invocations submitted by participating parties. Each peer maintains a copy of the ledger for each channel of which they are a member. The ledger is comprised of a blockchain and a state database, and it provides query and update capabilities using key-based lookups, range queries, and composite key queries.

Privacy is achieved in Hyperledger Fabric through the use of channels and private data collections. A ledger exists within the scope of a channel and can be either shared or privatized to include only specific participants. Private data collections are used to segregate confidential data in a private database, accessible only to authorized organizations. Values within chaincode can also be encrypted using common cryptographic algorithms to further obfuscate the data.

Security and membership services in Hyperledger Fabric provide a trusted blockchain network where participants know that all transactions can be detected and traced by authorized regulators and auditors. This is achieved through permissioned membership.

Consensus in Hyperledger Fabric is achieved through a unique approach that provides the flexibility and scalability required for the enterprise. This enables the network to reach agreement on the next set of updates to the ledger, ensuring that all participants have a consistent view of the network state.

4.3.2 The Network Structure

Fabric networks are composed of participants, who can be organizations, individuals, or devices that interact with the network by creating, validating, or executing transactions. Each participant operates one or more nodes, which are instances of the Fabric software running on a server or device. There are two types of nodes in a Fabric network: Peer nodes and orderer nodes. Peer nodes execute chaincode, participate in validating transactions and maintain the ledger. Peers can be configured to play different roles in the network, such as endorsement, committing, and delivering transactions. They also play a role in access control. ACLs are used to define which users and applications are authorized to access and execute chaincode on a specific peer. On the other hand, orderer nodes are responsible for ensuring the ordering and delivery of transactions to the peer nodes. They also enforce basic access control for channels, restricting who can read and write data to them, and who can configure them.

Each participant in a Fabric network is identified by a unique digital identity, represented by a public key. This identity is used to sign transactions and secure the network. Fabric uses a public key infrastructure (PKI) to manage these digital identities and ensure the authenticity of transactions within the network.

Participants can communicate with each other through channels. A channel is a private ledger within the Fabric network, which can only be accessed by participants who have joined it. This allows for transactions within the channel to be kept private among the members of that channel, while preserving the confidentiality and integrity of their transactions. Channels allow a group of participants to reach a consensus on a shared ledger, enabling them to share a common view of the state of their transactions. Each channel has its own ledger and chaincode, which is the smart contract logic that executes transactions on the network and is installed on the peer nodes. The ledger provides a secure and transparent history of all transactions within the channel, and the chaincode contains the business rules that govern the transactions within the network.

In summary, the structure of Fabric networks is designed to provide secure and transparent means of executing transactions between participants. By combining participants with unique digital identities, nodes that store and validate transactions, and channels that provide private ledgers and chaincode, Fabric networks provide a highly flexible and scalable solution for executing transactions and maintaining a secure and transparent ledger of all transactions. A simple depiction of a hyperledger fabric network can be seen in 4.1.

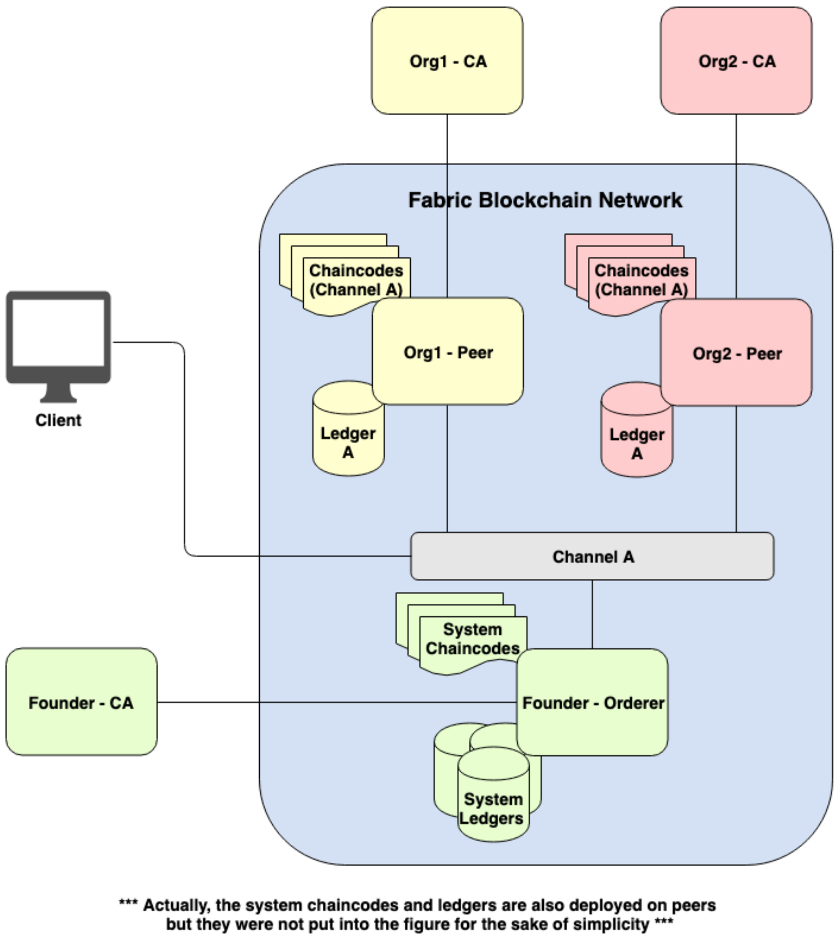


Figure 4.1: *Fabric Blockchain Network*

4.3.3 Identity and MSPs

In Fabric, a blockchain network is comprised of various actors such as peers, orderers, client applications, administrators, etc. Each of these actors has a unique digital identity encapsulated in an X.509 digital certificate. These identities determine the level of access and permission over resources and information in the network. In addition to the digital identity, additional attributes are also associated with the identity, forming a principal. A principal is simply a combination of an identity and its associated attributes and is used to determine the permissions of an actor.

For the identity to be considered verifiable, it must be issued by a trusted authority, known as an MSP. The MSP is responsible for managing the digital identities of participants in the network and for defining the rules that govern the valid identities in the organization. The MSP provides a way to define the relationships between different participants in the network, such as which participants are allowed to join a particular channel, who is authorized to transact on the network, and who can serve as an administrator for the network. It also defines the cryptographic material, such as public and private keys, that are used to secure transactions and establish the identity of participants. The MSP is used to ensure the confidentiality and privacy of transactions within the network by establishing secure communication channels between participants and enforcing access control policies. For example, an MSP can be used to define a policy that only authorized participants can transact on the network, and that transactions from unauthorized participants will be rejected. The MSP also provides a way to manage the revocation of identities in the network. If a participant's identity is compromised, the MSP can be used to revoke that identity and prevent further access to the network.

Fabric uses X.509 certificates as identities, and adopts a traditional PKI hierarchical model. The PKI provides secure communication in the network and is responsible for dispensing various verifiable identities. The role of the CA within a PKI is to issue digital certificates to parties, such as users or service providers, who then use these certificates to authenticate themselves in their communication. The CA's Certificate Revocation List (CRL) keeps a record of certificates that are no longer valid. Revocation of a certificate may occur due to various reasons, including exposure of the cryptographic private material associated with the certificate.

A PKI is crucial to ensure secure communication between network participants and to authenticate messages posted on the blockchain.

4.3.4 Policies

Hyperledger Fabric policies define the rules and regulations for access control, validation, and execution of transactions within a network. These policies are used to ensure that all participants in the network follow a specific set of guidelines, and are an essential component in maintaining the security and integrity of the network. The policies can be specified at various levels within the network, including the channel, peer, and application levels.

At the channel level, policies determine the level of permission that different participants have within the channel. For example, a policy can be set up to restrict the ability of certain participants to execute transactions on the network. At the peer level, policies can be used to define the validation rules for incoming transactions, ensuring that all transactions conform to specific standards before they are processed through ACLs. Finally, at the application level, policies can be used to control access to specific resources and functions within the application.

Policies in Hyperledger Fabric are defined using a declarative language called “policy syntax”. The syntax provides a concise and easy-to-use way to specify the conditions under which transactions can be executed, allowing for a wide range of use cases and flexible policy implementation. Additionally, policies can be easily modified or updated as the needs of the network evolve.

Overall, Hyperledger Fabric policies play a crucial role in maintaining the security and integrity of the network. By defining clear rules and regulations for access control, validation, and execution of transactions, they help to ensure that all participants in the network are able to trust each other and the network as a whole.

4.3.5 Ledger

The ledger in Hyperledger Fabric is a data structure that records all the transactions that occur in the network. It’s made up of multiple components, including the world state, blocks, transactions, and endorsements.

The world state represents the current state of all the assets in the network. It’s updated with every transaction and stores the latest version of the asset data.

Blocks are a collection of transactions that are grouped together and added to the ledger in a sequential manner. Each block contains a unique identifier and a list of transactions.

Transactions represent an exchange of information or assets within the network. They must be endorsed by multiple participants before being added to the ledger as a block.

Endorsements are a way to ensure the validity of transactions. They act as a form of consensus and help prevent fraud or tampering with the ledger. Endorsers are participants in the network who validate transactions before they are added to the ledger.

4.3.6 Smart Contracts/Chaincode

In Hyperledger Fabric, smart contracts, also called chaincode, are programs that run on a network of peer nodes. They serve as the business logic of a blockchain network, defining the rules and policies that govern interactions between participants. When a transaction is submitted to the network, it triggers the execution of one or more smart contracts, which perform operations like reading and writing data to the ledger, updating the state of assets, and triggering other smart contract executions.

Smart contracts in Hyperledger Fabric have access to a ledger, which is a shared database that is distributed across all the peer nodes in the network. This ledger provides a tamper-proof record of all transactions and states, allowing for transparency and trust in the network.

Hyperledger Fabric provides a secure and flexible framework for developing, deploying, and executing smart contracts. The smart contract code is stored on the peer nodes, and can be updated or replaced as necessary. The network's consensus mechanism ensures that all nodes agree on the current state of the ledger, and all transactions are validated before they are committed to the ledger.

4.3.7 Chaincode Lifecycle

The Hyperledger Fabric Chaincode Lifecycle refers to the process of creating, installing, instantiating, upgrading, and ultimately deprecating smart contracts in a Hyperledger Fabric network.

First, a chaincode must be written in a supported programming language. Next, the chaincode is installed onto the peers that will run it. After installation, the chaincode is instantiated on the channel and available for use. The chaincode can be later upgraded to add new functionality or fix bugs. The chaincode can also be deprecated, which means it is no longer used by the network and can be deleted from the peer nodes.

The lifecycle is managed through a series of REST APIs or command-line tools, which allow network administrators to control the various stages of the chaincode's existence. These tools provide an easy and secure way for administrators to manage the deployment and operation of chaincodes in a Hyperledger Fabric network.

4.3.8 Security Model

The security model aims to ensure the confidentiality, integrity, and availability of transactions and data on the network. This is achieved through a combination of authentication, authorization, encryption, and consensus mechanisms. Authentication verifies the identity of users, peers, and clients accessing the network. This is done through digital certificates, which are issued and managed by a certificate authority. Authorization defines the access and permissions granted to users, peers, and clients on the network. This is determined by the channel configuration and smart contract code. Encryption is used to secure sensitive information as it is transmitted and stored on the network. This includes data encryption, channel encryption, and end-to-end encryption. Consensus ensures that all participants in the network agree on the current state of the ledger, and that all transactions are valid. This is achieved through a consensus algorithm, such as PBFT or Raft. Overall, the security model in Hyperledger Fabric aims to maintain the privacy and security of transactions and data, while still enabling participants to collaborate effectively on the network.

4.3.9 Fabric Gateway

The Hyperledger Fabric Gateway provides an interface for applications to interact with a Fabric network. It abstracts the underlying implementation details of the network, such as the communication with peers, and offers a higher-level API for accessing the network's capabilities.

Applications can use the gateway to connect to a Fabric network, enroll a user, and execute transactions. The gateway can also be used to query the ledger for data. The gateway can be configured with one or more connections to Fabric networks, and each connection can be associated with a specific user identity. This allows for secure and controlled access to the network's resources, as each user is only able to perform actions for which they have been authorized.

To interact with the network, the gateway must first be connected to a network, and the user must be enrolled. The enrollment process involves the user providing their cryptographic materials, such as a private key, which are then used to authenticate the user to the network. Once connected and enrolled, the gateway can be used to execute transactions and query the ledger. The transactions are processed by the peers in the network, and the ledger is updated accordingly. The results of transactions and queries can be accessed through the gateway's API.

The Fabric Gateway offers a simplified and secure way for applications to interact with a Fabric network, abstracting the implementation details and providing a higher-level API.

4.4 Transaction Flow

A client application wants to access the ledger in a Fabric network. To do this, the client connects to the Fabric Gateway service running on a peer. Through the connection with the gateway, the client can run chaincodes to query or update the ledger. When a ledger update (write) is performed, it goes through three phases: Transaction Proposal and Endorsement, Transaction Submission and Ordering, and Transaction Validation and Commitment.

In the first phase, the client application submits a signed transaction proposal to the gateway service. The gateway selects a peer to execute the transaction, and the selected peer executes the chaincode, generates a proposal response, and signs it. The gateway repeats this process for each organization required by the endorsement policies. If the results collectively satisfy the endorsement policies, the proposal is forwarded to the ordering service running on orderer nodes, which are special nodes in the network working with the peers to facilitate the respective consensus.

In the second phase, the ordering service orders and packages the transactions into blocks. These blocks, which consist of endorsed and ordered transactions, make up the Fabric blockchain ledger.

In the third phase, the orderer distributes the ordered blocks to the peers. Each peer validates the transaction by checking the signatures and the read-write sets. If the transaction is valid, each peer commits it to the channel ledger, and sends the client a commit status event with proof of the ledger update. The world state of the channel is updated with the results of valid transactions only. Once all peers have completed this process, the ledger is consistent across the network and the gateway service informs the relevant applications that the transaction has been committed.

The transaction flow in Hyperledger Fabric can be summarised, as seen in Fig. 4.2, in the following steps:

1. A client sends a transaction proposal to one or more peers in the network.

2. The receiving peer(s) evaluate the proposal against the chaincode's endorsement policy.

3. If the proposal is valid, the peer signs the proposal response with its endorsement.

4. The client collects the endorsement responses from the peers and submits the transaction to the ordering service.

5. The ordering service orders the transactions and adds them to a block.

6. The block is then broadcast to all peers in the network for validation.

7. Each peer validates the transactions in the block, checking the state of the world and the chaincode's endorsement policy. If the transactions are valid, the peers update their state databases accordingly. The block

is committed to the ledger on the peer and the updated state is now visible to the network.

Note that the above steps happen in a decentralized manner, with each peer acting as an independent validator and maintaining its own copy of the ledger. This helps ensure the integrity of the network and eliminates the need for a central authority.

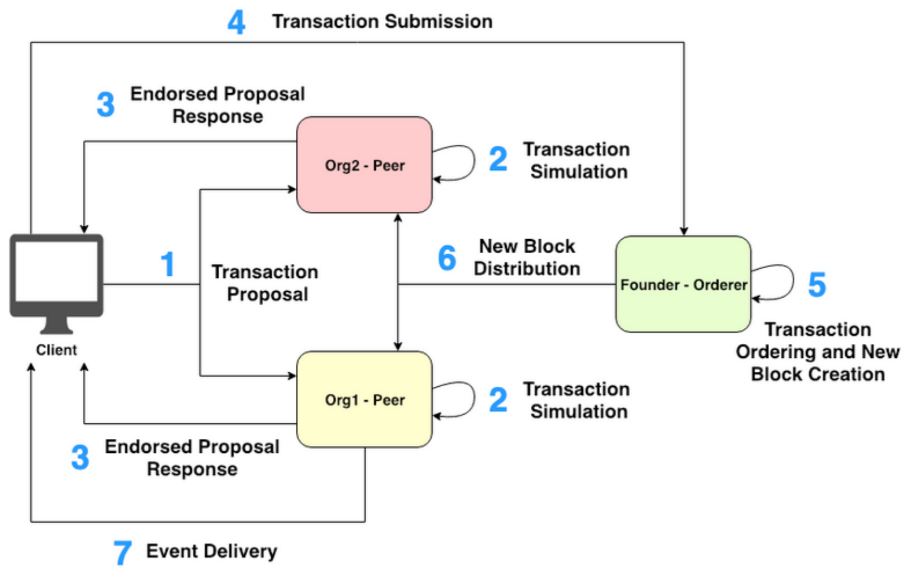


Figure 4.2: *Transaction flow*

Overall, the Fabric Gateway SDKs make it easy for developers to interact with the ledger by implementing these phases seamlessly, and the Fabric network ensures that the ledger is kept consistent and current by peers and orderers.

Chapter 5

Proposed Framework

5.1 Overview

The proposed blockchain-assisted framework uses the MANET described in Chapter 1 to incorporate the Blockchain protocol. Specifically, we will be using Hyperledger Fabric to implement the ledger.

Most commonly used blockchains are centered around managing and transferring digital assets. This gives them the ability to maintain the accuracy and authenticity of the stored data through the use of public key cryptography and digital signatures. For instance, in the case of Bitcoin, the transfer of bitcoins is secured through a PoW consensus mechanism. But when it comes to physical assets and commodities, the blockchain acts as a digital ledger of information related to these assets. However, since the information recorded on the blockchain only represents physical events and is not necessarily verified, the trustworthiness of the data becomes questionable, leading to questions about the data's integrity on the blockchain.

Even though blockchain can be a useful tool for keeping track of information, it is not enough to ensure the trustworthiness of the data about entities. If malicious actors input false information, it becomes permanent once recorded on the blockchain. To enhance the reliability of the data, accountability and incentive systems can be employed. These systems would penalize untruthful participants while rewarding trustworthy ones.

In this project, we will be using the MANET simulation of Chapter 1 to evaluate the reliability of the data recorded on the blockchain, to implement penalties and incentives, and ensure that the solution does not negatively impact the scalability of the platform. Here, we have made an effort to put together a blockchain-assisted implementation using Hyperledger Fabric. We opted to use Hyperledger Fabric [28] for deployment because it's easy to deploy and has various tools available for blockchain deployment, management, data querying, implementing smart contracts, and facilitating cross-organizational collaboration.

5.2 Blockchain-assisted MANET

As previously mentioned, maintaining traceability and integrity are significant hurdles in MANET systems. To address these issues, firstly, the information that provides traceability and integrity of reputation events must be recorded in a way that is resistant to tampering, and secondly, the recorded data must be genuine and reflect the actual observations of the mobile nodes. The blockchain technology satisfies the first requirement with its decentralized and secure ledger. The second requirement is fulfilled by the already developed mechanisms that establish trust in the data at its source and guarantee that the data recorded on the blockchain is trustworthy by the MANET network. Our work focuses on combining these two and reporting our observations.

Our framework is organized into two sections: the chaincode and the application. The chaincode is used to handle the ledger, while the application executes our simulation along with the transactions. In the following sections, we describe the functionality in detail.

5.2.1 Chaincode

We will be using the chaincode, presented in Appendix A, to handle the ledger. The initial state of the ledger holds the profiles of the 100 nodes that we will be using in our MANET simulation. These profiles are merely entries in the ledger that show the state of each node at any moment. Therefore, they store the node's id, the trust score of the node, the health status of the node, the cluster id of the node, the CH of the cluster the node belongs to and an array of all the transactions these node has been taking part in, either as a client or a server.

In the chaincode we can see the transactions that the application may invoke.

- queryNode: With this query any node can search and retrieve the profile of another node with a specified key.
- queryNodeCH: With this query a node that is a cluster head may search and retrieve the profile of another node with a specified key.

- queryAllNodes: This query will return all the nodes from the ledger.
- setCluster: This transaction is used to specify the cluster id and the cluster head of each node.
- changeNodeTrust: This transaction changes the trust score of a node on the ledger.
- changeNodeHS: This transaction changes the health status of a node on the ledger.
- isCH: This query checks if a node is cluster head.
- addTransactionClient: This adds a new transaction to the client node. The transaction, here, represents the transactions that occurs between nodes in the simulation. More on that in the next section.
- addTransactionServer: This adds a new transaction to the server node.
- queryNodeTransactions: This may be used to return all the transactions a specific node has been involved in.

Note that the cluster head check on the transactions is there to simulate the restriction that only a cluster head may access the ledger.

The chaincode will be deployed using Hyperledger Fabric test network. The test-network consists of multiple peers, that stores and processes transactions. These peers are organized into organizations, each representing a separate entity in the network. The organizations can be run by different entities, such as businesses, governments, or individuals, and each organization has its own set of peers.

The configuration of the Hyperledger Fabric test-network is specified using a set of configuration files. These files define the properties of the network, including the identities of the organizations and peers, the rules for processing transactions, and the network topology. More on that in the next section.

The Hyperledger Fabric test-network is run using Docker containers, which provide a secure and isolated environment for running the network components. This makes it easy to set up and run the network, and ensures that the network components are isolated from other processes running on the same machine.

5.2.2 Application Layer - MANET Simulation

The application layer will execute the MANET simulation that was presented in Chapter 1 and address queries and transaction requests for the chaincode to execute. The simulation, shown in Appendix B, begins by initialising and setting up a file system-based wallet and a Gateway connection to a blockchain network. The wallet is used to store and manage the identities that will be used to interact with the blockchain network, while the Gateway provides a connection to the network and enables the identities stored in the wallet to interact with it. The blockchain network is represented by a "Connection Profile" (CCP) file, which contains information about the network, including the configurations of the organizations that are part of it.

The identity for the Gateway is specified using the wallet and the "appUser" identity stored in the wallet. The CCP for the network is specified using the networkConfigPath. The "discovery" property is set to "true", which means that the Gateway will discover and connect to peers in the network, and the "commitHandler" property is set to "NONE", which means that the Gateway will not wait for commit events after submitting a transaction. These settings ensure that the code is set up to interact with the blockchain network as efficiently as possible.

After some initial node set up, the simulation will try to create a connection to the specified gateway in order to access the blockchain network and interact with a specific smart contract within the network. This includes submitting transactions to the network, querying the state of the contract, and executing smart contract functions. The simulation using time instances will begin right after.

At the start of each iteration, every node undergoes a self-assessment process, evaluating its own cost of analysis and measuring the proximity between itself and its neighbouring nodes. This self-check is crucial to ensure that there have been no alterations to these parameters. If any node that had previously made a positive impact on the overall cohesion of the network suddenly exits, an election process is initiated to determine a replacement. This method of self-evaluation and monitoring ensures the stability and continued cohesiveness of the network.

When an election is initiated, an election table is established to facilitate the voting process. Nodes with no neighbouring nodes will cast a vote in favor of themselves. However, if the node has neighbours, it will consider the cost of analysis of the neighbouring node. If the cost of analysis of the neighbour is lower, the node will cast its vote in favor of the neighbour. If the cost of analysis is higher, the node will vote for itself.

Once the voting is completed, the election table is sorted, and the cluster creation process begins. All nodes that have the first node of the election table as their first neighbour are deemed to have voted for that node and are assigned the same cluster ID as the first node, who becomes the CH. Any node with the CH as its first neighbour becomes a member of the CH's cluster. This process continues until all nodes have been assigned to a cluster.

Once the cluster creation process is completed, a transaction is submitted to the deployed contract to set the cluster ID and cluster head ID for each node. This information is used to update the cluster neighbours array for each node, removing any outdated information. It's important to note that these neighbours are 1-hop neighbours.

Next, the trust arrays are updated. If the CH has no neighbours, its LTV remains unchanged. However, if the CH has neighbours, it updates its own LTV based on the opinions of its neighbours for each node.

Finally, every CH submits a transaction that updates the trust score of each of its CMs according to its own LTV, which is the equivalent of the GTV. This process ensures the accuracy and reliability of the trust scores within the network.

The transaction process begins by checking if each node has sufficient energy to proceed. If there is any waiting time remaining, the node will wait until it is ready to proceed.

Once the node is available and has cluster neighbours, a random neighbour is selected as a candidate server. If the server is available, the client consults the CH. The CH retrieves the trust score of the server from the ledger and reports back to the client. If the trust score is low, the CH may give the server a second chance to prove its trustworthiness.

The client takes the trust score provided by the CH and combines it with its own assessment to produce a final trust score vector. This final trust score is then compared to a threshold. If the trust score is above the threshold, the transaction begins. If the trust score is below the threshold, the transaction is aborted. In this case, the node could either be malicious or healthy with low trust score.

If the transaction is approved, the duration of the transaction and the time it began are stored. Both the client and the server are made unavailable to prevent other nodes from starting a transaction with them. In the event of a malicious server, the client stores the time the health status was last changed.

If the server is not available, it means it must be in a transaction, and the client can identify the server it is transacting with based on the stored server ID. If the server is within range, the transaction duration is reduced by 1 and the energy of both the client and the server is also reduced.

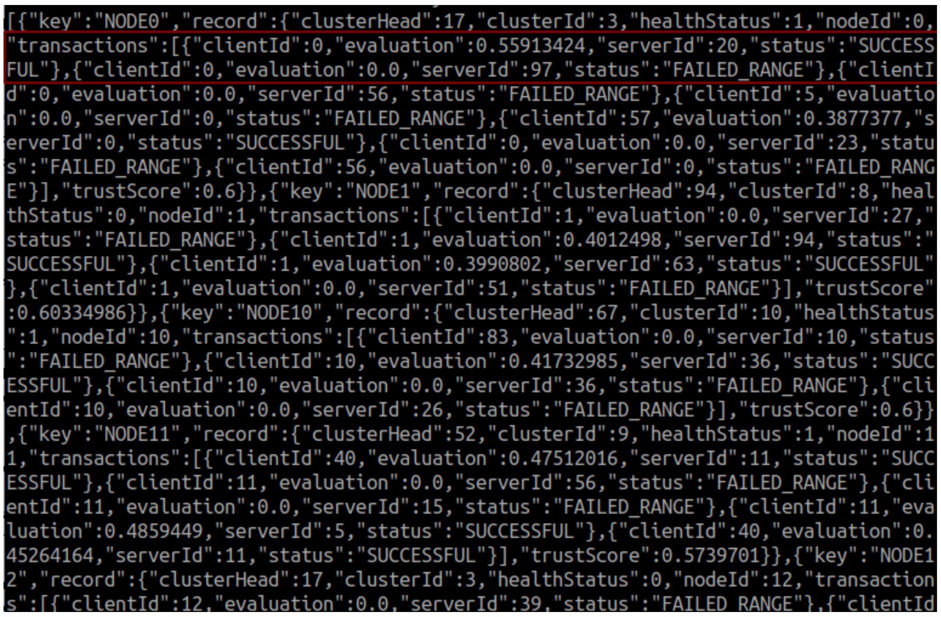
The transaction is stopped and the nodes are reset if the server is out of range or if either the client or the server runs out of energy during the transaction. These situations result in a failed transaction. If the server becomes malicious during the transaction, the malicious node and the time it became malicious are recorded.

The transaction is considered successful if both the client and the server remain within range for the duration of the transaction and have enough energy to complete it. Upon completion, an evaluation takes place to assess the success of the transaction.

The process of evaluating the performance of a node, after a transaction is completed, is a crucial aspect of the simulation. Each node is given a score based on its behavior during the transaction, with a positive score indicating honest behavior and a negative score indicating malicious behavior. This score, known as the evaluation, is calculated by comparing the node's behavior with predetermined standards.

If the node behaved honestly, its evaluation will be positive, ranging from 0 to 1. On the other hand, if the node engaged in malicious behavior, it will be penalized by -0.5 to its evaluation. Regardless of whether the transaction was successful, failed due to energy or range constraints, or was aborted, the transaction is recorded in the ledger for both the client and the server and added to the list of all transactions that each node has taken part in. Figure 5.1 shows the information retrieved after the ledger was queried to obtain the nodes' attributes along with the list of transactions associated with the nodes

Once the transaction is recorded, the client reports its evaluation for the server to the CH, who will combine this evaluation with its own opinion and update the ledger accordingly. This information is then used to update the trust scores of the nodes and to detect any malicious behavior. Finally, after the evaluation, the nodes are reset and the simulation produces its results.



```
[{"key":"NODE0","record":{"clusterHead":17,"clusterId":3,"healthStatus":1,"nodeId":0,
"transactions":[{"clientId":0,"evaluation":0.55913424,"serverId":20,"status":"SUCCESS
FUL"},{"clientId":0,"evaluation":0.0,"serverId":97,"status":"FAILED_RANGE"},{"clientI
d":0,"evaluation":0.0,"serverId":56,"status":"FAILED_RANGE"},{"clientId":5,"evaluatio
n":0.0,"serverId":0,"status":"FAILED_RANGE"},{"clientId":57,"evaluation":0.3877377,"s
erverId":0,"status":"SUCCESSFUL"},{"clientId":0,"evaluation":0.0,"serverId":23,"statu
s":"FAILED_RANGE"},{"clientId":56,"evaluation":0.0,"serverId":0,"status":"FAILED_RANG
E"}],"trustScore":0.6}},{"key":"NODE1","record":{"clusterHead":94,"clusterId":8,"heal
thStatus":0,"nodeId":1,"transactions":[{"clientId":1,"evaluation":0.0,"serverId":27,"
status":"FAILED_RANGE"},{"clientId":1,"evaluation":0.4012498,"serverId":94,"status":"
SUCCESSFUL"},{"clientId":1,"evaluation":0.3990802,"serverId":63,"status":"SUCCESSFUL"
},{"clientId":1,"evaluation":0.0,"serverId":51,"status":"FAILED_RANGE"}],"trustScore"
:0.60334986}},{"key":"NODE10","record":{"clusterHead":67,"clusterId":10,"healthStatus
":1,"nodeId":10,"transactions":[{"clientId":83,"evaluation":0.0,"serverId":10,"status
":"FAILED_RANGE"},{"clientId":10,"evaluation":0.41732985,"serverId":36,"status":"SUCC
ESSFUL"},{"clientId":10,"evaluation":0.0,"serverId":36,"status":"FAILED_RANGE"},{"cli
entId":10,"evaluation":0.0,"serverId":26,"status":"FAILED_RANGE"}],"trustScore":0.6}}
,{"key":"NODE11","record":{"clusterHead":52,"clusterId":9,"healthStatus":1,"nodeId":1
1,"transactions":[{"clientId":40,"evaluation":0.47512016,"serverId":11,"status":"SUCC
ESSFUL"},{"clientId":11,"evaluation":0.0,"serverId":56,"status":"FAILED_RANGE"},{"cli
entId":11,"evaluation":0.0,"serverId":15,"status":"FAILED_RANGE"},{"clientId":11,"eva
luation":0.4859449,"serverId":5,"status":"SUCCESSFUL"},{"clientId":40,"evaluation":0.
45264164,"serverId":11,"status":"SUCCESSFUL"}],"trustScore":0.5739701}},{"key":"NODE1
2","record":{"clusterHead":17,"clusterId":3,"healthStatus":0,"nodeId":12,"transaction
s":[{"clientId":12,"evaluation":0.0,"serverId":39,"status":"FAILED_RANGE"},{"clientId
```

Figure 5.1: *Nodes' Query Result*

5.3 Simulation and Results

5.3.1 Simulation

The network starts with setting up the infrastructure, which includes two organizations, two peers, one orderer, and one channel. The first step is to create a blockchain network using Hyperledger Fabric, which is a permissioned blockchain network. This network has two organizations, each with its own peer. The peers in each organization are responsible for maintaining the ledger and verifying transactions.

Next, an orderer node is added to the network to act as a centralized entity responsible for ordering transactions and ensuring consistency across all peers. This orderer node receives transactions from the peers and orders them into a single, consistent view of the ledger.

Once the network infrastructure is in place, a channel is created on the network. A channel is a private network within the overall fabric network, and it allows multiple parties to transact privately and securely on the same network.

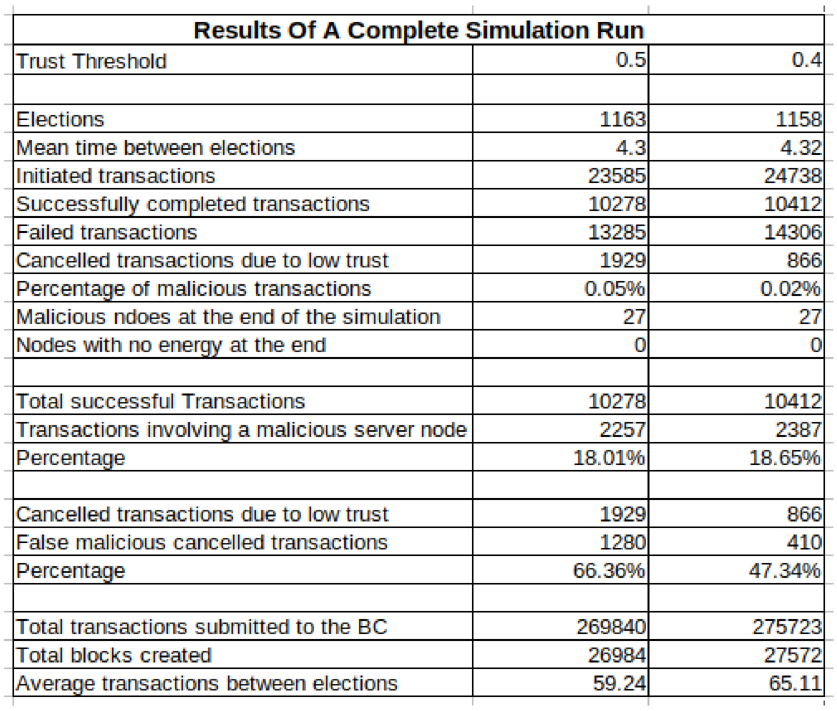
After setting up the network infrastructure, the next step is to deploy a chaincode, as shown in Appendix A, which is a smart contract that runs on the network. This chaincode defines the rules and logic of the network and contains the ledger of all transactions.

Finally, an application client is set up with one admin and one user, as shown in Appendix B. The application client is the simulation of the network and is used to submit transactions and interact with the ledger. The admin and the user represent the different entities that can participate in the network and submit transactions.

With the network infrastructure, chaincode, and application client in place, the network is now ready to start executing transactions and ensuring consensus on trust issues. The simulation runs on the application client and uses the ledger on the chaincode to keep track of transactions and evaluate the trustworthiness of nodes in the network. The network uses a combination of voting and trust scores to achieve consensus on trust issues and maintain a secure and trustworthy network.

5.3.2 Results

Multiple simulations with different initial conditions were conducted, and the results were found to be, more or less, consistent across all simulations. Figure 5.2 is a table of two simulation runs, where at first we set the trust threshold to 0.5 and then to 0.4. It is important to note that at the end of the simulation, some transactions were still ongoing and had not yet been completed. These transactions were considered initiated but were not included in the count of successful or failed transactions (about 20 transactions out of roughly 24000, a discrepancy which we consider negligible).



Results Of A Complete Simulation Run		
Trust Threshold	0.5	0.4
Elections	1163	1158
Mean time between elections	4.3	4.32
Initiated transactions	23585	24738
Successfully completed transactions	10278	10412
Failed transactions	13285	14306
Cancelled transactions due to low trust	1929	866
Percentage of malicious transactions	0.05%	0.02%
Malicious ndoes at the end of the simulation	27	27
Nodes with no energy at the end	0	0
Total successful Transactions	10278	10412
Transactions involving a malicious server node	2257	2387
Percentage	18.01%	18.65%
Cancelled transactions due to low trust	1929	866
False malicious cancelled transactions	1280	410
Percentage	66.36%	47.34%
Total transactions submitted to the BC	269840	275723
Total blocks created	26984	27572
Average transactions between elections	59.24	65.11

Figure 5.2: *Results of a Complete Simulation Run*

As stated before, the A1 metric tracks the time it takes for a client-node to detect a server-node's switch from a healthy state to a malicious state during a transaction. If this change occurs for over 50% of the transaction's duration, the client-node will penalize the server-node's trust at the end of the transaction, making the malicious state known to other nodes. This metric is important because it shows how long it takes for a client-node to realize that the server-node has become malicious, even though the server-node appeared healthy at the beginning of the transaction.

In Fig. 5.2, we can see the total number of successful transactions and the total number of transactions that involved a malicious server. Fig. 5.3 displays the frequency of the time span between a healthy node becoming malicious and when it was detected by another node. Finally, the mean time required to spot a malicious node with A1 metric is 25.81.

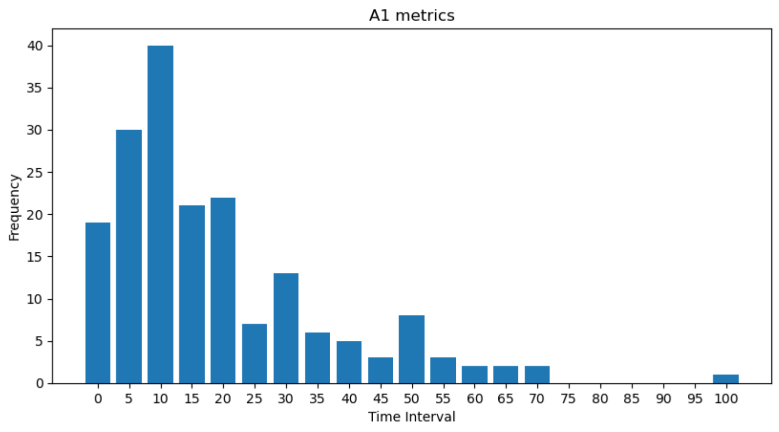


Figure 5.3: *A1 metric frequency distribution*

According to the analysis, out of a total of 10278 transactions that were successfully completed, the 2257 involved a malicious server-node, which is 18.01%. This means that less than two in ten transactions involved a malicious node, which initially appeared to be perfectly healthy. However, the evaluation of the trust of a node through a transaction carries a certain risk, as it requires a transaction to take place before the trust can be evaluated and there is no guarantee that a node is actually malicious based solely on the trust evaluation.

The A2 metric measures the time from when a previously healthy node turns malicious to when a client-node discovers this change and cancels the transaction before it starts. It concerns nodes that have been previously assessed and have a trust level below the trust threshold. This means that they have had at least one interaction with another client-node, hence why the recorded time in the A2 metric is higher. In Fig. 5.2, we see the results of a simulation run for this metric. The frequency distribution of the A2 metric is shown in Fig. 5.4. The mean time of the A2 metric is 29.48.

It's found that a significant proportion (66.36%) of nodes with low trust are actually healthy and transactions with them are cancelled. This can happen because they were previously evaluated by a high-standard node or because they were once malicious but have recovered. Therefore, provisions must be made for restoring the trust of these nodes.

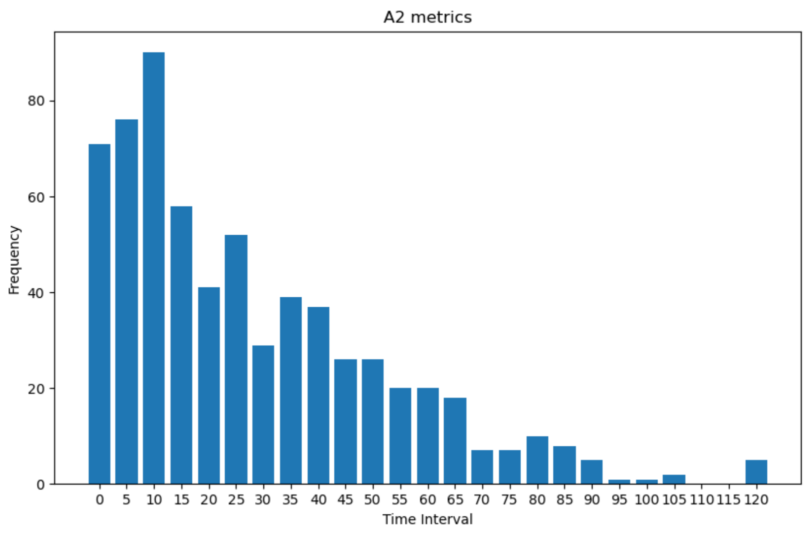


Figure 5.4: *A2 metric frequency distribution*

The A3 metric focuses on addressing the issue of restoring trust to nodes that have previously been evaluated as malicious or have a low trust level. It measures the time it takes for a node's trust to be restored after transitioning from a healthy state to a malicious state. This is done through intervention by the CH, who increases the node's trust value by adjusting its GTV component, although there is inherent risk in this decision. It is the responsibility of the CH to assess and manage this risk. Fig. 5.5 shows the time distribution of the trust restorations. The mean time interval for a node's trust to be restored is 14.11.

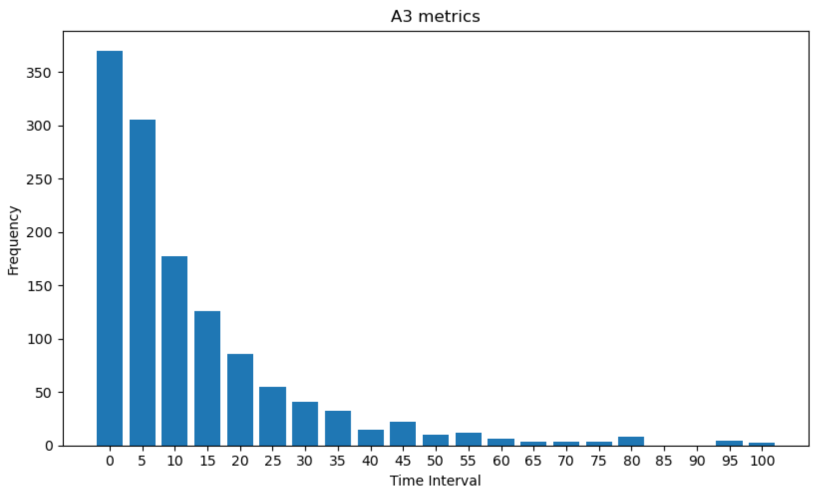


Figure 5.5: *A3 metric frequency distribution*

At this point, we should mention that the behavior of the CH is determined by the α parameter in the low pass filter. In our simulation, a value of 0.4 was used, but this doesn't take into account the timeliness of the information or the trustworthiness of the reporting node. A more recent evaluation from a trustworthy node is more reliable than an old evaluation from a low-trust node. This is something to implement in future works.

5.3.3 Caliper

Caliper is a blockchain performance benchmark tool developed by the Hyperledger project. It provides a means of measuring the performance of a blockchain network and its components, such as the ledger, consensus algorithms, and smart contract execution. Caliper is used to evaluate the scalability, efficiency, and stability of a blockchain network, providing valuable insights into the network's strengths and weaknesses. With Caliper, various types of tests can be performed, including transactions per second (TPS) tests, latency tests, and resource consumption tests. This information can be used to optimize the performance of a blockchain network and ensure that it can meet the demands of real-world applications.

We used Caliper to run a series of tests on our network, which was comprised of two organizations, two peers, one orderer, one communication channel, and a chaincode. The chaincode, the configuration file and some of the transactions files that were used to execute the tests are shown in Appendix C.

The test involved 5 worker nodes, executing 7 rounds of transactions. Each round involves a different set of operations such as creating a node, setting a cluster, changing the node's trust and health status, adding client and server transactions and querying the nodes. The transactions in each round were executed at a fixed rate of 10 transactions per second to 80 transactions per second. The configuration includes 500 assets, which are used as arguments in the different workload modules. The results for the ChangeNodeTrust transaction and –even though we are not currently using it in the simulation– the CreateNode transaction, are shown below. The rest produce, more or less, the same results.

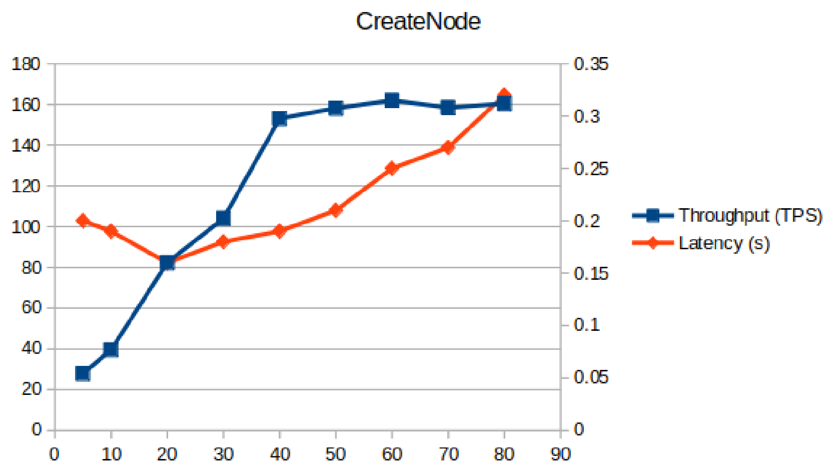


Figure 5.6: *CreateNode Transaction's Latency, Throughput*

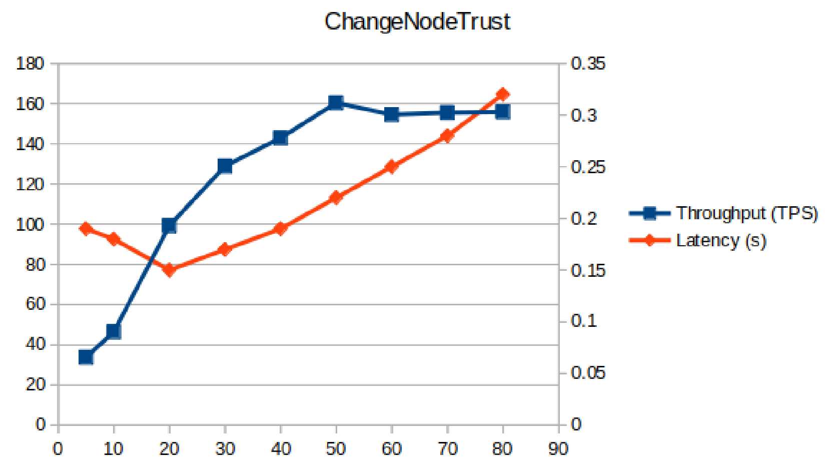


Figure 5.7: *ChangeNodeTrust Transaction's Latency, Throughput*

The results of the Caliper benchmark test show the relationship between transaction load, throughput, and latency. Here is a brief explanation of each of these metrics regarding the ChangeNodeTransaction:

- Transaction load: This refers to the number of transactions that were sent to the blockchain network during the benchmark test. In this test, we varied the transaction load from 10 to 80, with each increment of 10 representing an increase in the number of transactions.
- Throughput (TPS): This is the number of transactions that were processed by the blockchain network per second. In this test, the throughput increased as the transaction load increased, indicating that the network was able to handle more transactions per second. The highest throughput was 160.4 transactions per second, achieved with a transaction load of 50.
- Latency: This is the amount of time it takes for a transaction to be processed by the blockchain network. From [29], it can be concluded that the transaction latency is made up of various components such as endorsement latency, broadcast latency, commit latency, and ordering latency. However, Caliper only provides the total delay and does not break it down into these individual parts. In any way, in this test, the latency was relatively low, ranging from 0.15 to 0.32 seconds, even as the transaction load increased. This indicates that the network was able to process transactions quickly, without experiencing significant delays or bottlenecks.

Overall, the results of the Caliper benchmark test suggest that the blockchain network and chaincode are performing well, with a high throughput and low latency even under increased transaction load. However, it's important to note that benchmark test results may vary depending on the specific network and hardware configuration, as well as the type and complexity of the transactions being processed.

Note that the simulation and the performance tests are carried out on a Dell PC (Intel Core i7, 3.60 GHz, 8 GB memory).

5.4 Future Work

Now, we will explore several ways to improve the efficiency, security, scalability, and consensus of our simulation. As the number of nodes in our simulation grows, so does the complexity of managing the interactions between them. To ensure the trustworthiness of our simulation, it is important to ensure that it is secure against malicious actors and to maintain consensus among the nodes. Furthermore, as the simulation grows, we must also consider its scalability to ensure that it remains manageable and performant. With these considerations in mind, we will explore various techniques for improving the efficiency, security, scalability, and consensus of our simulation, such as optimizing the data model, improving the validation of transactions, enhancing privacy, increasing fault tolerance, and implementing more robust consensus algorithms. By implementing these improvements, we can create a simulation that is both realistic and secure, allowing us to gain valuable insights into how distributed systems can be used to model complex interactions between nodes.

5.4.1 Security

In order to improve security in our system, we could introduce access control rules in the chaincode. These can be implemented in several ways, here are a few approaches:

- Role-based access control: We could define roles, such as CH and CM, in the chaincode and assign permissions to each role. For example, we could allow CHs to write to the ledger and restrict other nodes from writing.
- Condition-based access control: We could specify conditions in the chaincode that determine if a node can perform a specific action, such as writing to the ledger. For example, we could check if a node is a CH before allowing it to write to the ledger.
- Use of smart contracts: We could write smart contracts in the chaincode that enforce access control rules. The smart contract can define the conditions that must be met before a node can interact with the ledger. For example, we could specify that before a node can add a transaction to the ledger, it must meet certain conditions such as having a minimum trust score or being a member of a specific cluster.
- Authentication and authorization: We could use authentication and authorization mechanisms, such as digital signatures, private keys, and encryption algorithms, to control access to the ledger. The chaincode can validate the authenticity and authorization of a node before allowing it to interact with the ledger.

Fabric's ACLs can be used to enhance the security and access control of this system by specifying who can perform certain actions or access certain resources within the network.

One way we could exploit Fabric's ACLs to make our system better is by using them to enforce fine-grained access control. This means that we can define access control policies for each resource or action in the network, and limit access to only those parties that need it. For example, we could use ACLs to restrict access to sensitive data, such as the trust scores of nodes, to only authorized parties such as CHs. This can help prevent unauthorized access or tampering with sensitive data by malicious actors.

Another way we could exploit Fabric's ACLs is to implement a layered access control model. By defining different access levels for different roles within the network, we can limit access to only those resources and actions that are necessary for each role to perform their duties. For example, we could define an access control policy that allows CHs to modify trust scores, but only allows regular nodes, CMs, to view their own trust score. This can help prevent accidental or intentional modifications to data by unauthorized parties.

We could also use Fabric's ACLs to restrict access to authorized parties. This can help prevent unauthorized nodes from joining the network and potentially compromising its security. Additionally, we could use ACLs to define policies for managing the addition and removal of nodes from the network, further enhancing the security and scalability of the system.

In order to implement this, first, we would need to define the access rules that specify who can read or write the data on the ledger. This would involve setting up a list of entities that are allowed to access the ledger and specifying the actions they can perform. Next, we would send the message digest of the data to the blockchain layer in the form of transactions. The transactions would be stored on the ledger and processed according to the access rules defined by the ACL. To ensure that the transactions are valid and do not violate the access rules, the validating peers would verify them before adding them to the ledger. This would involve checking whether the entity submitting the transaction is allowed to perform the requested action.

Once the transactions are added to the ledger, they would invoke smart contracts that generate reputation and trust values for entities and quality ratings for commodities using the reputation and trust module. The smart contracts would also emit warning events depending on predefined conditions, i.e. if a node becomes malicious. The reputation and trust values would be stored on the digital profiles of the nodes on the blockchain. The digital profiles would be mapped to the nodes using their unique IDs, such as their public keys.

Finally, the application layer would interact with the blockchain layer through queries about the trust scores. The CHs would query about the trust scores of nodes in their clusters, and the quality of the interactions and transactions involving those nodes. Based on the retrieved scores, the CHs would take actions such as rewarding nodes with high scores by providing them with more privileges or authority within the cluster, and penalizing nodes with low scores by reducing their privileges or even removing them from the cluster. These actions would help to ensure the overall health and trustworthiness of the network.

Overall, by utilizing Fabric's ACLs, we can enhance the security, access control, and scalability of the network, making it more robust and resistant to malicious attacks or unauthorized access.

One more thing that could improve the security of the network are regular audits. We could conduct regular audits of the ledger to identify any security issues or anomalies. By monitoring the ledger for unusual activity, we could detect potential security breaches and take appropriate action to address them.

5.4.2 Efficiency-Scalability

The system can become more efficient and scalable by having the chaincode handle the election process. First, the requirements for becoming a CH can be clearly defined within the chaincode, allowing for a clear and consistent understanding of what is required. This can simplify the election process and help to minimize confusion or uncertainty. Additionally, the chaincode can automatically initiate elections based on a set schedule or certain criteria, such as the current CH's trust score dropping below a certain threshold. This can improve the speed and reliability of the election process, as well as ensure that elections are conducted in a fair and transparent manner.

Furthermore, having the chaincode handle the election process can reduce the amount of manual intervention required, which can free up resources and allow for more efficient and streamlined operations. Overall, having the chaincode handle the election process can help to improve the scalability and efficiency of the system, as well as provide greater control and transparency over the election process.

One other way to improve scalability and efficiency in the system is to transfer the trust reputation calculation to the chaincode. This way, we can ensure the security of the simulation by preventing nodes from tampering with their own trust scores. The trust score of each node would be calculated using a validation function defined in the chaincode. This function would be based on the interactions stored in the ledger and would take into account each node's behavior and interactions with other nodes. By relying on the chaincode to calculate trust scores, we would reduce the risk of fraud and increase the accuracy of the results. Furthermore, this would allow the CHs to easily update the trust scores of their nodes without the need for manual calculation, freeing up resources and speeding up the overall process.

The chaincode could also handle the evaluation process. Then, the simulation would become more automated and efficient. The chaincode would define the criteria for evaluating a node's trust score, such as the number of successful transactions and positive feedback from other nodes. This eliminates the need for CHs to manually adjust trust scores, freeing them up to focus on other tasks that require their attention. Automating the evaluation process through the chaincode also ensures that the calculations are consistent and unbiased, improving the overall accuracy and reliability of the trust score evaluation. Additionally, moving the evaluation process to the chaincode adds an extra layer of security, as it prevents nodes from tampering with their own trust scores. The use of a chaincode in this manner helps to improve both the scalability and efficiency of the simulation.

5.4.3 Ensuring Consensus

The proposed framework is designed to provide a basis for a sturdy trust management system by utilizing a combination of voting algorithms, cluster formation, trust evaluation mechanisms and the ability to detect and prevent malicious nodes. The consensus on trust issues is ensured through the use of trusted intermediaries (CHs) and the enforcement of rules and conditions through smart contracts.

This approach provides a good foundation for ensuring consensus on trust issues in a limited-capabilities node network using the blockchain protocol. However, it may not be sufficient to fully address all the challenges that come with trust management in blockchain networks.

It's important to keep in mind that blockchain networks are decentralized and rely on a large number of nodes to maintain their security and trust. The use of a trust score and election of CHs can provide some level of trust management, but it may not be sufficient to fully address all the potential security and trust issues that may arise in a decentralized network.

For example, in a network with limited-capabilities nodes, the trust score assigned to a node may not accurately reflect its true trustworthiness, as the node may not have the necessary resources or capabilities to participate in transactions. This could lead to security vulnerabilities and trust issues in the network.

Additionally, relying on a single trusted intermediary, such as the CH, may not be enough to ensure consensus on trust issues in a large and complex network. The CH could be compromised, or its decisions could be influenced by malicious actors. This could undermine the trust and security of the network.

Another possibility could be to implement a mechanism for verifying the trustworthiness of the CHs themselves. This could be done by having a secondary layer of nodes that oversee the actions of the CHs and provide additional oversight to ensure that the trust scores being reported are accurate. Additionally, implementing a mechanism for dispute resolution in case of conflicting trust scores could help to further ensure consensus on trust issues.

To improve the trust management system, additional factors could be considered when calculating the trust score, such as the node's past performance in transactions or its reputation in the community. By implementing a reputation system that tracks the past behavior of each node it would lead to more accurate calculations of the trust scores and provide more accurate recommendations for the CH elections. Assuming that the trust score of each candidate CH was taken into account in the election process.

Furthermore, using multiple ledgers, where each ledger is managed by a different CH or another trusted entity, can improve the consensus on trust issues. Each ledger can have its own method of trust evaluation, which can then be combined to produce a final trust score.

Taking it one step further, incorporating machine learning algorithms can help identify and detect malicious nodes based on their behavior patterns. This can help prevent malicious nodes from affecting the overall trust evaluation of the network.

Therefore, while this approach provides a good starting point, it may be necessary to supplement it with additional security measures and trust management techniques to ensure consensus on trust issues in a limited-capabilities node network using the blockchain protocol. By implementing these measures, we can ensure that consensus on trust is maintained across the entire system and that the system is able to identify and mitigate any trust issues that may arise.

Conclusions

This thesis has explored the integration of blockchain technology into a MANET that employs a clustering scheme and a trust mechanism to detect malicious nodes. The proposed system, which utilizes the Hyperledger Fabric framework, presents a promising foundation for achieving consensus on trust issues within a resource-constrained network.

We conducted a review of related work which proposed different approaches to incorporating the blockchain protocol, demonstrating the versatility of the technology. Our proposed system builds on these approaches, utilizing a unique clustering scheme based on the cost of analysis/processing concept, specifically designed for MANETs. The system also incorporates a trust mechanism to ensure network integrity and address malicious nodes.

In order to fully understand the Blockchain protocol we conducted a thorough analysis of its features and key concepts. We also delved into Hyperledger Fabric, which was selected as the framework to implement the blockchain protocol. The analysis of Hyperledger Fabric covered various aspects, such as its structure, the chaincode, the transaction flow. This enabled us to develop our proposed system that merges the blockchain technology with the given MANET simulation.

Next, the proposed framework was presented along with the results it generated. As previously discussed, it is designed to ensure consensus on trust issues by utilizing a combination of voting algorithms, cluster formation, and trust evaluation mechanisms. A clustering scheme is utilized based on cost of analysis/processing, along with a trust mechanism, to ensure transparent and objective cluster formation. Each cluster is headed by a CH who is responsible for maintaining the trust scores of nodes within the cluster. The trust scores are stored on the ledger, along with other node attributes such as health status, cluster ID, and transaction history, and used to determine node trustworthiness.

To ensure only trustworthy nodes participate in transactions, the proposed system uses the CH as a trusted intermediary. The CH consults the ledger about the trust scores of the nodes involved in the transaction and only approves it if the trust score of the other node is above a certain threshold. By relying on the trust scores maintained by the CHs, the system achieves consensus on trust and maintains the integrity of the network. If a malicious node is discovered, the trust score of that node can be adjusted accordingly, and the CH is informed. This helps to prevent malicious nodes from participating in transactions.

The ledger is updated accordingly after each transaction through the use of smart contracts. Smart contracts ensure that transactions are recorded in a tamper-proof manner on the ledger, and that all parties involved in the transaction can agree on the outcome. This adds an additional layer of trust and security to the network, as all peers can independently verify the state of the ledger.

In conclusion, the proposed system stands out due to its unique design and provides a good starting point to start building a robust trust management solution for addressing trust and consensus issues in MANETs. The combination of trust scores, the election process, smart contracts, and the ability to detect and prevent malicious nodes makes the proposed system unique and promising for future work in this area. While there is still room for improvement in terms of performance optimization, scalability, and enhanced security features, the proposed framework represents a solid foundation for integrating blockchain technology into MANETs. As blockchain technology continues to evolve, there will undoubtedly be more opportunities to explore and enhance the capabilities of this proposed system.

References

[1] Michail Chatzidakis and Stathes Hadjiefthymiades. “Trust management in mobile ad hoc networks”. In: *2014 16th International Telecommunications Network Strategy and Planning Symposium (Networks)*. 2014, pp. 1–6. DOI: 10.1109/NETWKS.2014.6958525.

[2] Michail Chatzidakis and Stathes Hadjiefthymiades. “Location Aware Clustering and Epidemic Trust Management in Mobile Ad Hoc Network”. In: *2019 28th International Conference on Computer Communication and Networks (ICCCN)*. 2019, pp. 1–7. DOI: 10.1109/ICCCN.2019.8847166.

[3] Michail Chatzidakis and Stathes Hadjiefthymiades. “A trust change detection mechanism in mobile ad-hoc networks”. In: *Computer Communications* 187 (2022), pp. 155–163. ISSN: 0140-3664. DOI: https://doi.org/10.1016/j.comcom.2022.02.007. URL: https://www.sciencedirect.com/science/article/pii/S0140366422000445.

[4] W.A. Jabbar, M. Ismail, R. Nordin and S. Arif. “Power-efficient routing schemes for MANETs: a survey and open issues”. In: *Wireless Networks* 23 (2017), pp. 1917–1952. DOI: 10.1007/s11276-016-1263-6. URL: https://doi.org/10.1007/s11276-016-1263-6.

[5] Kajal S. Patel and J. S. Shah. “Detection and avoidance of malicious node in MANET”. In: *2015 International Conference on Computer, Communication and Control (IC4)*. 2015, pp. 1–4. DOI: 10.1109/IC4.2015.7375729.

[6] Kirti Gupta and Pardeep Mittal. “An Overview of Security in MANET”. In: *International Journal of Advanced Research in Computer Science and Software Engineering* 7 (June 2017), pp. 151–156. DOI: 10.23956/ijarcsse/V7I6/0254.

[7] Noman Mohammed et al. “Mechanism Design-Based Secure Leader Election Model for Intrusion Detection in MANET”. In: *IEEE Transactions on Dependable and Secure Computing* 8.1 (2011), pp. 89–103. DOI: 10.1109/TDSC.2009.22.

[8] S. M. Mousavi et al. “MobiSim: A Framework for Simulation of Mobility Models in Mobile Ad-Hoc Networks”. In: *Third IEEE International Conference on Wireless and Mobile Computing, Networking and Communications (WiMob 2007)*. 2007, pp. 82–82. DOI: 10.1109/WIMOB.2007.4390876.

[9] *Blockchain Overview*. URL: https://en.wikipedia.org/wiki/Blockchain.

[10] *Blockchain Overview*. URL: https://www.investopedia.com/terms/b/blockchain.asp.

[11] *Public, Private, Permissioned Blockchains*. URL: https://www.investopedia.com/news/public-private-permissioned-blockchains-compared/.

[12] *Blockchain Features*. URL: https://101blockchains.com/introduction-to-blockchain-features/.

[13] Asma Lahbib et al. “Blockchain based trust management mechanism for IoT”. In: *2019 IEEE Wireless Communications and Networking Conference (WCNC)*. 2019, pp. 1–8. DOI: 10.1109/WCNC.2019.8885994.

[14] Rakesh Shrestha et al. “A new type of blockchain for secure message exchange in VANET”. In: *Digit. Commun. Networks* 6 (2020), pp. 177–186.

[15] *Digital Signature*. URL: https://blockgeeks.com/what-is-hashing-digital-signature-in-the-blockchain/.

[16] *Immutable Ledger*. URL: https://www.solulab.com/what-is-immutable-ledger-in-blockchain-and-its-benefits/.

[17] *P2P Networks*. URL: https://www.blockchain-council.org/blockchain/blockchain-role-of-p2p-network/.

[18] Arati Baliga. “Understanding Blockchain Consensus Models”. In: 2017.

[19] *Consensus Algorithms*. URL: https://www.decipherzone.com/blog-detail/consensus-algorithms.

[20] *Mining Transactions*. URL: https://blog.goodaudience.com/how-a-miner-adds-transactions-to-the-blockchain-in-seven-steps-856053271476.

[21] *Blockchain Mining*. URL: https://intellipaat.com/blog/tutorial/blockchain-tutorial/what-is-bitcoin-mining/.

[22] Ralph C. Merkle. “A Digital Signature Based on a Conventional Encryption Function”. In: *Advances in Cryptology — CRYPTO ’87*. Ed. by Carl Pomerance. Berlin, Heidelberg: Springer Berlin Heidelberg, 1988, pp. 369–378. ISBN: 978-3-540-48184-3.

[23] Zhe Yang et al. “Blockchain-Based Decentralized Trust Management in Vehicular Networks”. In: *IEEE Internet of Things Journal* 6.2 (2019), pp. 1495–1505. DOI: 10.1109/JIOT.2018.2836144.

[24] Cong Pu. “A Novel Blockchain-Based Trust Management Scheme for Vehicular Networks”. In: *2021 Wireless Telecommunications Symposium (WTS).* 2021, pp. 1–6. DOI: 10.1109/WTS51064.2021.9433711.

[25] Sidra Malik et al. “TrustChain: Trust Management in Blockchain and IoT Supported Supply Chains”. In: *2019 IEEE International Conference on Blockchain (Blockchain).* 2019, pp. 184–193. DOI: 10.1109/Blockchain.2019.00032.

[26] Xu Wu and Junbin Liang. “A blockchain-based trust management method for Internet of Things”. In: *Pervasive and Mobile Computing* 72 (2021), p. 101330. ISSN: 1574-1192. DOI: https://doi.org/10.1016/j.pmcj.2021.101330. URL: https://www.sciencedirect.com/science/article/pii/S1574119221000079.

[27] *Hyperledger Fabric Documentation.* URL: https://hyperledger-fabric.readthedocs.io/en/release-2.5/.

[28] C. Cachin. “Architecture of the hyperledger blockchain fabric”. In: *Workshop on Distributed Cryptocurrencies and Consensus Ledgers* 310 (2016).

[29] Parth Thakkar, Senthil Nathan, and Balaji Viswanathan. “Performance Benchmarking and Optimizing Hyperledger Fabric Blockchain Platform”. In: *2018 IEEE 26th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS).* 2018, pp. 264–276. DOI: 10.1109/MASCOTS.2018.00034.

Appendix A

Chaincode

A.1 Node

```java
import org.hyperledger.fabric.contract.annotation.DataType;
import org.hyperledger.fabric.contract.annotation.Property;
import com.owlike.genson.annotation.JsonProperty;
import org.hyperledger.fabric.contract.annotation.Transaction;
import java.util.ArrayList;
import java.util.List;
import java.util.Objects;

@DataType()
public final class Node {

    @Property()
    private int nodeId;

    @Property()
    private double trustScore;

    @Property()
    private int healthStatus;

    @Property()
    private int clusterId;

    @Property()
    private int clusterHead;

    @Property
    private List<NodeTransaction> transactions;

    public Node(@JsonProperty("nodeId") final int nodeId, @JsonProperty("trustScore") final double
                trustScore, @JsonProperty("healthStatus") final int healthStatus, @JsonProperty("clusterId")
                final int clusterId, @JsonProperty("clusterHead") final int clusterHead,
                @JsonProperty("transactions") final List<NodeTransaction> transactions) {
        this.nodeId = nodeId;
        this.healthStatus = healthStatus;
        this.trustScore = trustScore;
        this.clusterId = clusterId;
        this.clusterHead = clusterHead;
        this.transactions = transactions;
    }

    public int getClusterHead() {
        return clusterHead;
    }

    public int getClusterId() {
        return clusterId;
    }

    public int getNodeId() {
        return nodeId;
    }

    public double getTrustScore() {
        return trustScore;
    }

    public int getHealthStatus() {
        return healthStatus;
    }

    public List<NodeTransaction> getTransactions() {
        return transactions;
    }

    public void addTransaction(NodeTransaction tx){
        this.transactions.add(tx);
    }

    @Override
    public boolean equals(final Object obj) {
        if (this == obj) {
            return true;
        }

        if ((obj == null)  (getClass() != obj.getClass())) {
            return false;
        }

        Node other = (Node) obj;

        return Objects.deepEquals(new String[] {String.valueOf(getNodeId()), String.valueOf(getTrustScore()),
                    String.valueOf(getHealthStatus()),String.valueOf(getClusterId()),
                    String.valueOf(getClusterHead())}, new String[] {String.valueOf(other.getNodeId()),
                    String.valueOf(other.getTrustScore()), String.valueOf(other.getHealthStatus()),
                    String.valueOf(other.getClusterId()), String.valueOf(other.getClusterHead())});
    }

    @Override
    public int hashCode() {
        return Objects.hash(getNodeId(), getTrustScore(), getHealthStatus(), getClusterId(), getClusterHead());
    }

    @Override
    public String toString() {
        return this.getClass().getSimpleName() + "@" + Integer.toHexString(hashCode()) + " [nodeId="
                + nodeId + ", trustScore=" + trustScore + ", healthStatus=" + healthStatus + ", clusterId="
                + clusterId + ", clusterHead=" + clusterHead + ", transactions = " + transactions + " ]";
    }
}
```

A.2 NodeTransaction

```java
import com.owlike.genson.annotation.JsonProperty;
import org.hyperledger.fabric.contract.annotation.Property;

public class NodeTransaction {
    @Property
    private int clientId;
    @Property
    private int serverId;
    @Property
    private double evaluation;
    @Property
    private TransactionStatus status;

    public NodeTransaction(@JsonProperty("clientId") final int clientId, @JsonProperty("serverId")
                           int serverId, @JsonProperty("evaluation") final double evaluation,
                           @JsonProperty("status") final TransactionStatus status) {
        this.clientId = clientId;
        this.serverId = serverId;
        this.evaluation = evaluation;
        this.status = status;
    }

    public int getClientId() {
        return clientId;
    }

    public int getServerId() {
        return serverId;
    }

    public double getEvaluation() {
        return evaluation;
    }

    public TransactionStatus getStatus() {
        return status;
    }
}
```

A.3 TransactionStatus

```java
public enum TransactionStatus {
    SUCCESSFUL,
    FAILED_ENERGY,
    FAILED_RANGE,
    ABORTED;
}
```

A.4 NodeQueryResult

```java
import com.owlike.genson.annotation.JsonProperty;
import org.hyperledger.fabric.contract.annotation.Property;
import java.util.Objects;

```

```java
public class NodeQueryResult {
    @Property()
    private final String key;

    @Property()
    private final Node record;

    public NodeQueryResult(@JsonProperty("Key") final String key, @JsonProperty("Record")
                           final Node record) {
        this.key = key;
        this.record = record;
    }

    public String getKey() {
        return key;
    }

    public Node getRecord() {
        return record;
    }

    @Override
    public boolean equals(final Object obj) {
        if (this == obj) {
            return true;
        }

        if ((obj == null)  (getClass() != obj.getClass())) {
            return false;
        }

        NodeQueryResult other = (NodeQueryResult) obj;

        Boolean recordsAreEquals = this.getRecord().equals(other.getRecord());
        Boolean keysAreEquals = this.getKey().equals(other.getKey());

        return recordsAreEquals && keysAreEquals;
    }

    @Override
    public int hashCode() {
        return Objects.hash(this.getKey(), this.getRecord());
    }

    @Override
    public String toString() {
        return this.getClass().getSimpleName() + "@" + Integer.toHexString(hashCode()) +
                " [key=" + key + ", record=" + record + "]";
    }
}
```

A.5 Manet

```java
import java.util.ArrayList;
import java.util.List;
import java.util.regex.Matcher;
import java.util.regex.Pattern;
import java.util.stream.Collectors;
import java.util.stream.Stream;
import org.hyperledger.fabric.contract.Context;
import org.hyperledger.fabric.contract.ContractInterface;
import org.hyperledger.fabric.contract.annotation.Contract;
import org.hyperledger.fabric.contract.annotation.Default;
import org.hyperledger.fabric.contract.annotation.Info;
import org.hyperledger.fabric.contract.annotation.Transaction;
import org.hyperledger.fabric.shim.ChaincodeException;
import org.hyperledger.fabric.shim.ChaincodeStub;
import org.hyperledger.fabric.shim.ledger.KeyValue;
import org.hyperledger.fabric.shim.ledger.QueryResultsIterator;
import com.owlike.genson.Genson;

/**
 * Java implementation of the Manet Contract
 */
@Contract(
        name = "Manet",
        info = @Info(
                title = "Manet contract",
                description = "The hyperlegendary node contract",
                version = "0.0.1-SNAPSHOT"
                ))

@Default
public final class Manet implements ContractInterface {

    public static final int MAX_NODES = 100;
    public static final double INIT_TRUST = 0.6;
    private final Genson genson = new Genson();

    private enum ManetErrors {
        NODE_NOT_FOUND,
        NODE_NOT_CH,
        TRANSACTION_NOT_FOUND,
        TRANSACTION_ALREADY_EXISTS,
        INVALID_STATUS
    }

    /**
     * Creates initial Nodes on the ledger.
     *
     * @param ctx the transaction context
     */
    @Transaction()
    public void initLedger(final Context ctx) {
        ChaincodeStub stub = ctx.getStub();

        for (int i = 0; i < MAX_NODES; i++) {
            String key = String.format("NODE%d", i);
            Node node = new Node(i, INIT_TRUST,1, 0, 0, new ArrayList<NodeTransaction>());
            String nodeState = genson.serialize(node);
            stub.putStringState(key, nodeState);
        }
    }

    /**
     * Retrieves a node with the specified key from the ledger.
     *
     * @param ctx the transaction context
     * @param key the key
     * @return the Node found on the ledger if there was one
     */
    @Transaction()
    public Node queryNode(final Context ctx, final String key) {
        ChaincodeStub stub = ctx.getStub();
        String nodeState = stub.getStringState(key);

        if (nodeState.isEmpty()) {
            String errorMessage = String.format("Node %s does not exist", key);
            System.out.println(errorMessage);
            throw new ChaincodeException(errorMessage, ManetErrors.NODE_NOT_FOUND.toString());
        }

        Node node = genson.deserialize(nodeState, Node.class);

        return node;
    }

    /**
     * Retrieves a node with the specified key from the ledger.
     *
     * @param ctx the transaction context
     * @param key the key
     * @param ch_key the key of the cluster head that makes the query
     * @return the Node found on the ledger if there was one
     */
    @Transaction()
    public Node queryNodeCH(final Context ctx, final String ch_key, final String key) {
        if(!isCH(ctx, ch_key)) {
            String errorMessage = String.format("Node %s is not clusterhead", ch_key);
            System.out.println(errorMessage);
            throw new ChaincodeException(errorMessage, ManetErrors.NODE_NOT_CH.toString());
        }
        Node node = queryNode(ctx, key);
        return node;
    }

    /**
     * Retrieves all nodes from the ledger.
     *
     * @param ctx the transaction context
     * @return array of Nodes found on the ledger
     */
    @Transaction()
    public String queryAllNodes(final Context ctx) {
        ChaincodeStub stub = ctx.getStub();

        final String startKey = "NODE0";
        final String endKey = "NODE99";
        List<NodeQueryResult> queryResults = new ArrayList<NodeQueryResult>();

        QueryResultsIterator<KeyValue> results = stub.getStateByRange(startKey, endKey);

        for (KeyValue result: results) {
            Node node = genson.deserialize(result.getStringValue(), Node.class);
            queryResults.add(new NodeQueryResult(result.getKey(), node));
        }

        final String response = genson.serialize(queryResults);

        return response;
    }

    /**
```

```java
     * Sets a node's cluster on the ledger.
     *
     * @param ctx the transaction context
     * @param key the key
     * @param clusterId the id of the node's cluster
     * @param clusterHead the id of the cluster's cluster head
     * @return the updated Node
     */
    @Transaction()
    public Node setCluster(final Context ctx, final String key, final String clusterId,
                           final String clusterHead) {
        ChaincodeStub stub = ctx.getStub();

        String nodeState = stub.getStringState(key);

        if (nodeState.isEmpty()) {
            String errorMessage = String.format("Node %s does not exist", key);
            System.out.println(errorMessage);
            throw new ChaincodeException(errorMessage, ManetErrors.NODE_NOT_FOUND.toString());
        }

        Node node = genson.deserialize(nodeState, Node.class);

        Node newNode = new Node(node.getNodeId(), node.getTrustScore(), node.getHealthStatus(),
                                Integer.parseInt(clusterId), Integer.parseInt(clusterHead),
                                node.getTransactions());
        String newNodeState = genson.serialize(newNode);
        stub.putStringState(key, newNodeState);

        return newNode;
    }

    /**
     * Changes the trust score of a node on the ledger.
     *
     * @param ctx the transaction context
     * @param key the key
     * @param ch_key the key of the cluster head that makes the query
     * @param newTS the new trust score
     * @return the updated Node
     */
    @Transaction()
    public Node changeNodeTrust(final Context ctx, final String ch_key, final String key,
                                final String newTS) {
        ChaincodeStub stub = ctx.getStub();

        if(!isCH(ctx, ch_key)) {
            String errorMessage = String.format("Node %s is not clusterhead", ch_key);
            System.out.println(errorMessage);
            throw new ChaincodeException(errorMessage, ManetErrors.NODE_NOT_CH.toString());
        }

        Node node = queryNode(ctx, key);
        Node newNode = new Node(node.getNodeId(), Double.parseDouble(newTS), node.getHealthStatus(),
                                node.getClusterId(), node.getClusterHead(), node.getTransactions());
        String newNodeState = genson.serialize(newNode);
        stub.putStringState(key, newNodeState);

        return newNode;
    }

    /**
```

```java
     * Changes the health status of a node on the ledger.
     *
     * @param ctx the transaction context
     * @param key the key
     * @param newHS the new reputation
     * @return the updated Node
     */
    @Transaction()
    public Node changeNodeHS(final Context ctx, final String key, final String newHS) {
        ChaincodeStub stub = ctx.getStub();

        Node node = queryNode(ctx, key);
        Node newNode = new Node(node.getNodeId(), node.getTrustScore(), Integer.parseInt(newHS),
                                node.getClusterId(), node.getClusterHead(), node.getTransactions());
        String newNodeState = genson.serialize(newNode);
        stub.putStringState(key, newNodeState);

        return newNode;
    }

    /**
     * Check CH status.
     *
     * @param ctx the transaction context
     * @param key the key
     * @return true if node is ClusterHead. else false
     */
    @Transaction()
    public Boolean isCH(final Context ctx, final String key) {
        Node node = queryNode(ctx, key);

        if(node.getClusterHead() == node.getNodeId()){
            return true;
        }
        return false;
    }

    /**
     * Adds new transaction to the client node
     *
     * @param ctx the transaction context
     * @param ch_key the key of the cluster head that makes the query
     * @param client the clientId
     * @param server the serverId
     * @param evaluation the evaluation the client gave to the server
     * @param status the transaction status
     * @return the updated Node
     */
    @Transaction()
    public Node addTransactionClient(final Context ctx, final String ch_key,
                                     final String client, final String server,
                                     final String evaluation, final String status) {
        ChaincodeStub stub = ctx.getStub();

        if(!isCH(ctx, ch_key)) {
            String errorMessage = String.format("Node %s is not clusterhead", ch_key);
            System.out.println(errorMessage);
            throw new ChaincodeException(errorMessage, ManetErrors.NODE_NOT_CH.toString());
        }

        Pattern p = Pattern.compile("[a-z]+|\\d+");
        Matcher m = p.matcher(client);
        String clientId = "";
        while (m.find()) {
            clientId = m.group();
        }

        NodeTransaction transaction = new NodeTransaction(Integer.parseInt(clientId),
                                      Integer.parseInt(server), Double.parseDouble(evaluation),
                                      TransactionStatus.valueOf(status));

        Node node = queryNode(ctx, client);
        node.addTransaction(transaction);
        Node newNode = new Node(node.getNodeId(), node.getTrustScore(), node.getHealthStatus(),
                                node.getClusterId(), node.getClusterHead(), node.getTransactions());
        String newNodeState = genson.serialize(newNode);
        stub.putStringState(client, newNodeState);

        return newNode;
    }

    /**
     * Adds new transaction to the server node
     *
     * @param ctx the transaction context
     * @param ch_key the key of the cluster head that makes the query
     * @param client the clientId
     * @param server the serverId
     * @param evaluation the evaluation the client gave to the server
     * @param status the transaction status
     * @return the updated Node
     */
    @Transaction()
    public Node addTransactionServer(final Context ctx, final String ch_key,
                                     final String client, final String server,
                                     final String evaluation, final String status) {
        ChaincodeStub stub = ctx.getStub();

        if(!isCH(ctx, ch_key)) {
            String errorMessage = String.format("Node %s is not clusterhead", ch_key);
            System.out.println(errorMessage);
            throw new ChaincodeException(errorMessage, ManetErrors.NODE_NOT_CH.toString());
        }

        Pattern p = Pattern.compile("[a-z]+|\\d+");
        Matcher m = p.matcher(server);
        String serverId = "";
        while (m.find()) {
            serverId = m.group();
        }

        NodeTransaction transaction = new NodeTransaction(Integer.parseInt(client),
                                      Integer.parseInt(serverId), Double.parseDouble(evaluation),
                                      TransactionStatus.valueOf(status));

        Node node = queryNode(ctx, server);
        node.addTransaction(transaction);
        Node newNode = new Node(node.getNodeId(), node.getTrustScore(), node.getHealthStatus(),
                                node.getClusterId(), node.getClusterHead(), node.getTransactions());
        String newNodeState = genson.serialize(newNode);
        stub.putStringState(server, newNodeState);

        return newNode;
    }

    /**
     * Query a transaction
     *
     * @param ctx the transaction context
     * @param key the key of the node
     * @return the updated Node
     */
    @Transaction()
    public List<NodeTransaction> queryNodeTransactions(final Context ctx, final String key) {
        ChaincodeStub stub = ctx.getStub();
        Node node = queryNode(ctx, key);
        return node.getTransactions();
    }
}
```

Appendix B

Client Application

B.1 Node

```java
import java.util.Random;

public class Node {

    //----------------general node parameters----------------------
    Random rnd = new Random();
    int x = -1;
    int xPrev = -1;
    int y = -1;
    int yPrev = -1;
    int k = 0;
    int i = 0;
    int nodeNo = 100;
    int nodeId = 0;
    float cohesion = 0;
    float initialTrust = 0.6f;
    int healthStatus = 1; //1=healthy, 0=malicious
    float probMal = 0.01f; //probability to become malicious
    float probHealth = 0.03f; //probability to become healthy
    float aLPF = 0.4f;
    //for the markovian
    float[][] healthProbArr = {{probHealth, (1 - probHealth)},
            {probMal, (1 - probMal)}};
    int malTimeStamp = 0; //last timestamp that the node was malicious
    int nIndex = 0;
    int wTime = 0; //mean time between transactions
    float lamda = 0.1f;
    public float nodeRep = 0.5f;
    //random or else for t=1 there is a problem
    int nodeEnergy = rnd.nextInt(1000) + 3000;
    int nodeEnergyThreshold = 10;
    public double costOfAnalysis;
    public int clusterId = 0;
    public int clusterHeadId = 0;
    public boolean isClHead = true;
    // new/old neighbour arrays and temp for sorting
    //if there is a change, calls for election
    int[] electors = new int[nodeNo];
    double[][] neighbours = new double[nodeNo][5];
    //neighbourhood array
    //for every neighbour: id and cost of analysis
    //for Cluster Head election
    double[][] oldNeighbours = new double[nodeNo][4];
    //array with neighbours' ids for the previous timestamp
    //if previous neighbours are different from current, calls for election
    int[] clusterNeighbours = new int[nodeNo];
    double[][] temp = new double[1][5];
    //array for trust vector sorting
    float[][] trustVector = new float[nodeNo][2];

    //----------------client node parameters--------------
    float eLTV = 0.8f;
    boolean avClient = true; //if available for client
    int transStart = 0; //timestamp when the transaction starts
    //duration of server being malicious
    int malTimeCounter = 0;
    //timestamp when the server became malicious
    int serverMalTS = -1;
    int serverId = -1; //if there is a server
    int dur = 0;
    int durLeft = -1;
    //----------------server node parameters--------------
    boolean avServer = true; //if available for server
    //=====================================================

    void setTransStart(int time) {
        this.transStart = time;
    }

    void node(int nodId, int x, int y) {
        this.nodeId = nodId;
        this.x = x;
        this.y = y;
    }

    void initNeighbourArray() {
        for (k = 0; k < nodeNo; k++) {
            for (i = 0; i < 5; i++) {
                neighbours[k][i] = -1;
            }
            //resets the pointer to the beginning so that the next time
            //the addneighbour is executed will rewrite from the beginning
            this.nIndex = 0;
        }
    }

    void clearClNeighbours() {
        for (i = 0; i < nodeNo; i++) {
            clusterNeighbours[i] = -1;
        }
    }

    void transferNeighbours() {
        for (i = 0; i < nodeNo; i++) {
            System.arraycopy(neighbours[i], 0, oldNeighbours[i], 0, 4);
        }
    }

    void addNeighbour(int neighbourId, double costOfAnalysis, float dist, float distPrev) {
        this.neighbours[nIndex][0] = neighbourId;
        this.neighbours[nIndex][1] = costOfAnalysis;
        this.neighbours[nIndex][2] = dist;
        this.neighbours[nIndex][3] = distPrev;
        if (this.neighbours[nIndex][2] < this.neighbours[nIndex][3]) {
            this.neighbours[nIndex][4] = 1;
        } else {
            neighbours[nIndex][4] = 0;
        }
        nIndex++;
    }

    void sortNeighbours() { //sort by cost of analysis
        for (k = 1; k < (nodeNo - 1); k++) {
            for (i = 0; i < (nodeNo - k); i++) {
                if (neighbours[i + 1][0] == -1) {
                    break;
                }
                if (neighbours[i + 1][1] < neighbours[i][1]) {
                    temp[0][0] = neighbours[i][0];
                    temp[0][1] = neighbours[i][1];
                    temp[0][2] = neighbours[i][2];
                    temp[0][3] = neighbours[i][3];
                    temp[0][4] = neighbours[i][4];
                    neighbours[i][0] = neighbours[i + 1][0];
                    neighbours[i][1] = neighbours[i + 1][1];
                    neighbours[i][2] = neighbours[i + 1][2];
                    neighbours[i][3] = neighbours[i + 1][3];
                    neighbours[i][4] = neighbours[i + 1][4];
                    neighbours[i + 1][0] = temp[0][0];
                    neighbours[i + 1][1] = temp[0][1];
                    neighbours[i + 1][2] = temp[0][2];
                    neighbours[i + 1][3] = temp[0][3];
                    neighbours[i + 1][4] = temp[0][4];
                }
            }
        }
    }

    void initElectCopy() {
        for (i = 0; i < nodeNo; i++) {
            electors[i] = -1;
        }
    }

    int countNeighbours() {
        i = 0;
        while (this.neighbours[i][0] != -1) {
            i++;
        }
        return i;
    }

    int countClNeighbours() {
        i = 0;
        while (this.clusterNeighbours[i] != -1) {
            i++;
        }
        return i;
    }

    boolean isClNeighbour(int serverNode) {
        for (k = 0; k < nodeNo; k++) {
            if (this.clusterNeighbours[k] == serverNode) {
                return true;
            }
        }
        return false;
    }

    void setWTime() { //sets an exponential waiting time
        this.wTime = (int) ((Math.log(1 - rnd.nextFloat()))
                / ((-1) * lamda)) + 1;
    }

    void decrDelay() {
        this.wTime = this.wTime - 1;
    }

    void costCalc(float totRep) {
        this.costOfAnalysis = (this.nodeRep / totRep) / this.nodeEnergy;
    }

    void setClusterId(int id) {
        this.clusterId = id;
    }

    void setHead() {
        this.isClHead = true;
    }

    void setClusterHeadId(int id) {
        this.clusterHeadId = id;
    }

    void electionReset() {
        this.clusterId = 0;
        this.clusterHeadId = -1;
        this.isClHead = false;
    }

    void initTrustVector() { //initialise the trust vector
        for (i = 0; i < nodeNo; i++) {
            trustVector[i][0] = initialTrust;
            trustVector[i][1] = 1;
        }
    }

    void setDur() {
        //keep the duration to be used in trust
        this.dur = this.durLeft;
    }

    //changes or keeps the same health status
    void healthProbe(int time) {
        if (rnd.nextDouble() < this.healthProbArr[healthStatus][0]) {
            this.healthStatus = (this.healthStatus + 1) % 2;
            //when it becomes malicious, store the timestamp
            if (this.healthStatus == 0) {
                this.malTimeStamp = time;
            }
        }
    }

    int getMalTimeStamp() {
        return this.malTimeStamp;
```

```java
    }

    void malCounterReset() {
        this.malTimeCounter = 0;
    }

    float CH_Report(int node) {
        if (this.trustVector[node][0] < 0.5) {
            //2nd chance
            return this.trustVector[node][0] + rnd.nextFloat() / 4f;
        }
        return this.trustVector[node][0];
    }

    float CH_Timestamp(int node) {
        return this.trustVector[node][1];
    }

    void report(int server, float trust, int time) {
        this.trustVector[server][0] = aLPF * trust + (1 - aLPF)
                * this.trustVector[server][0];
        this.trustVector[server][1] = time;
    }

    void LTVupdate(int node, float trust, int time, float timestamp) {
        this.trustVector[node][0] = eLTV * trust + (1 - eLTV)
                * this.trustVector[node][0];
        this.trustVector[node][1] = time;
    }

    void eLTV_restore() {
        this.eLTV = 1f;
    }

    void resetClient() {
        this.avClient = true;
        this.transStart = 0;
        this.malTimeCounter = 0;
        this.serverMalTS = -1;
        this.serverId = -1;
        this.dur = 0;
        this.durLeft = -1;
    }

    void resetServer() {
        this.avServer = true;
    }

    void decEnergy() {
        this.nodeEnergy--;
    }
}
```

B.2 ClHeadElection

```java
public class ClHeadElection {

    int nodeNo = 100;
    int i = 0, k = 0, l = 0;
    int elecNo = 0;
    int[][] elecTable = new int[nodeNo][2];
    int[] elecTableSorted = new int[nodeNo];
    int[][] temp = new int[1][2];

    void initElect() {
        for (i = 0; i < nodeNo; i++) {
            elecTable[i][0] = i;
            elecTable[i][1] = 0;
        }
    }

    void sortElecTable() {
        for (k = 0; k < nodeNo; k++) {
            elecTableSorted[k] = elecTable[0][0];
            for (l = 0; l < nodeNo; l++) {
                if (elecTable[l][1] > elecTable[elecTableSorted[k]][1]) {
                    elecTableSorted[k] = elecTable[l][0];
                }
            }
            elecTable[elecTableSorted[k]][1] = -1;
        }
    }

    void delSRow(int cl) {
        for (i = 0; i < nodeNo; i++) {
            if (this.elecTableSorted[i] == cl) {
                this.elecTableSorted[i] = -1;
            }
        }
    }

    void printElecTable() {
        for (i = 0; i < nodeNo; i++) {
            System.out.println(this.elecTable[i][0] + " "
                    + this.elecTable[i][1]);
        }
    }
```

B.3 RegisterUser

```java
import java.nio.file.Paths;
import java.security.PrivateKey;
import java.util.Properties;
import java.util.Set;

import org.hyperledger.fabric.gateway.Wallet;
import org.hyperledger.fabric.gateway.Wallets;
import org.hyperledger.fabric.gateway.Identities;
import org.hyperledger.fabric.gateway.Identity;
import org.hyperledger.fabric.gateway.X509Identity;
import org.hyperledger.fabric.sdk.Enrollment;
import org.hyperledger.fabric.sdk.User;
import org.hyperledger.fabric.sdk.security.CryptoSuite;
import org.hyperledger.fabric.sdk.security.CryptoSuiteFactory;
import org.hyperledger.fabric_ca.sdk.HFCAClient;
import org.hyperledger.fabric_ca.sdk.RegistrationRequest;

public class RegisterUser {

    static {
        System.setProperty("org.hyperledger.fabric.sdk.service_discovery.as_localhost", "true");
    }

    public static void main(String[] args) throws Exception {

        // Create a CA client for interacting with the CA.
        Properties props = new Properties();
        props.put("pemFile",
            "../../test-network/organizations/peerOrganizations/org1.example.com/
                ca/ca.org1.example.com-cert.pem");
        props.put("allowAllHostNames", "true");
        HFCAClient caClient = HFCAClient.createNewInstance("https://localhost:7054", props);
        CryptoSuite cryptoSuite = CryptoSuiteFactory.getDefault().getCryptoSuite();
        caClient.setCryptoSuite(cryptoSuite);

        // Create a wallet for managing identities
        Wallet wallet = Wallets.newFileSystemWallet(Paths.get("wallet"));

        // Check to see if we've already enrolled the user.
        if (wallet.get("appUser") != null) {
            System.out.println("An identity for the user \"appUser\" already exists in
                                the wallet");
            return;
        }

        X509Identity adminIdentity = (X509Identity)wallet.get("admin");
        if (adminIdentity == null) {
            System.out.println("\"admin\" needs to be enrolled and added to the wallet first");
            return;
        }
        User admin = new User() {

            @Override
            public String getName() {
                return "admin";
            }

            @Override
            public Set<String> getRoles() {
                return null;
            }

            @Override
            public String getAccount() {
                return null;
            }

            @Override
            public String getAffiliation() {
                return "org1.department1";
            }

            @Override
            public Enrollment getEnrollment() {
                return new Enrollment() {

                    @Override
                    public PrivateKey getKey() {
                        return adminIdentity.getPrivateKey();
                    }

                    @Override
                    public String getCert() {
                        return Identities.toPemString(adminIdentity.getCertificate());
                    }
                };
            }

            @Override
            public String getMspId() {
                return "Org1MSP";
            }

        };

        // Register the user, enroll the user, and import the new identity into the wallet.
        RegistrationRequest registrationRequest = new RegistrationRequest("appUser");
        registrationRequest.setAffiliation("org1.department1");
        registrationRequest.setEnrollmentID("appUser");
        String enrollmentSecret = caClient.register(registrationRequest, admin);
        Enrollment enrollment = caClient.enroll("appUser", enrollmentSecret);
        Identity user = Identities.newX509Identity("Org1MSP", enrollment);
        wallet.put("appUser", user);
        System.out.println("Successfully enrolled user \"appUser\" and imported it
                                into the wallet");
    }
}
```

B.4 EnrollAdmin

```java
import java.nio.file.Paths;
import java.util.Properties;

import org.hyperledger.fabric.gateway.Wallet;
import org.hyperledger.fabric.gateway.Wallets;
import org.hyperledger.fabric.gateway.Identities;
import org.hyperledger.fabric.gateway.Identity;
import org.hyperledger.fabric.sdk.Enrollment;
import org.hyperledger.fabric.sdk.security.CryptoSuite;
import org.hyperledger.fabric.sdk.security.CryptoSuiteFactory;
import org.hyperledger.fabric_ca.sdk.EnrollmentRequest;
import org.hyperledger.fabric_ca.sdk.HFCAClient;

public class EnrollAdmin {

    static {
        System.setProperty("org.hyperledger.fabric.sdk.service_discovery.as_localhost", "true");
    }

    public static void main(String[] args) throws Exception {

        // Create a CA client for interacting with the CA.
        Properties props = new Properties();
        props.put("pemFile",
            "../../test-network/organizations/peerOrganizations/org1.example.com/
                    ca/ca.org1.example.com-cert.pem");
        props.put("allowAllHostNames", "true");
        HFCAClient caClient = HFCAClient.createNewInstance("https://localhost:7054", props);
        CryptoSuite cryptoSuite = CryptoSuiteFactory.getDefault().getCryptoSuite();
        caClient.setCryptoSuite(cryptoSuite);

        // Create a wallet for managing identities
        Wallet wallet = Wallets.newFileSystemWallet(Paths.get("wallet"));

        // Check to see if we've already enrolled the admin user.
        if (wallet.get("admin") != null) {
            System.out.println("An identity for the admin user \"admin\" already exists
                                in the wallet");
            return;
        }

        // Enroll the admin user, and import the new identity into the wallet.
        final EnrollmentRequest enrollmentRequestTLS = new EnrollmentRequest();
        enrollmentRequestTLS.addHost("localhost");
        enrollmentRequestTLS.setProfile("tls");
        Enrollment enrollment = caClient.enroll("admin", "adminpw", enrollmentRequestTLS);
        Identity user = Identities.newX509Identity("Org1MSP", enrollment);
        wallet.put("admin", user);
        System.out.println("Successfully enrolled user \"admin\" and imported it into the wallet");
    }
}
```

B.5 ClientApp

```java
import java.io.BufferedWriter;
import java.io.File;
import java.io.FileWriter;
import java.math.BigDecimal;
import java.nio.file.Path;
import java.nio.file.Paths;
import java.util.Random;
import java.util.Scanner;
import org.hyperledger.fabric.gateway.*;
import org.json.simple.JSONObject;
import org.json.simple.parser.JSONParser;

public class ClientApp {

    static {
        System.setProperty("org.hyperledger.fabric.sdk.service_discovery.as_localhost",
                           "true");
    }

    public static void main(String[] args) throws Exception {
        // Load a file system based wallet for managing identities.
        Path walletPath = Paths.get("wallet");
        Wallet wallet = Wallets.newFileSystemWallet(walletPath);
        // load a CCP
        Path networkConfigPath = Paths.get("..", "..", "test-network",
                                           "organizations", "peerOrganizations",
                                           "org1.example.com", "connection-org1.yaml");

        Gateway.Builder builder = Gateway.createBuilder();
        builder.identity(wallet, "appUser").networkConfig(networkConfigPath).discovery(true);
        //Do not wait for any commit events to be received from peers
        //after submitting a transaction since we use only one client
        builder.commitHandler(DefaultCommitHandlers.valueOf("NONE"));

        int i, j, k, l; //counters
        boolean electionFlag = true;
        int time;       //the number of time snapshots
        int nodeNo = 100;
        int maxTime = 5000;
        float dist;    // node euclidean distance
        float distPrev; //previous distance
        int range = 100;  //distance of interest
        int boxr;   //box range
        int candServer; //variable for canditare server
        float lamda = 0.1f;
        float totRep = 0; //total reputation
        int clusterId;
        int clusterNodeNo = 0; //number of cluster members (without CH)
        float aLPF = 0.4f; //low pass filter parameter
        float trustThreshold = 0.4f; //node trust threshold
        int cH;
        int tStarted = 0; //initiated transactions
        int sTrans = 0; //successful transactions
        int fTrans = 0; //failed transactions
        int mTrans = 0; //transactions with malicious server
        int falseMal = 0;
        int trustFailed = 0; //failed due to trust
        int clusterCount;
        float minTrust = 0;
        float maxTrust = 1;
        float evaluation;
        float[] tempTrustVector = new float[nodeNo];
        int[] blackList = new int[nodeNo];
        byte[] result;
        JSONParser parser = new JSONParser();
        int lastHS ;

        //files for metrics
        FileWriter fstreamA1 = new FileWriter("A1_metrics.txt");
        FileWriter fstreamA2 = new FileWriter("A2_metrics.txt");
        FileWriter fstreamA3 = new FileWriter("A3_metrics.txt");
        //FileWriter fstreamNodes = new FileWriter("allNodes.txt");
        FileWriter fstreamWt = new FileWriter("wtime.txt");
        FileWriter fstreamDur = new FileWriter("Dur.txt");
        FileWriter fstream = new FileWriter("sig.txt");
        FileWriter fstreamGen = new FileWriter("gen.txt");

        BufferedWriter outA1 = new BufferedWriter(fstreamA1);
        BufferedWriter outA2 = new BufferedWriter(fstreamA2);
        BufferedWriter outA3 = new BufferedWriter(fstreamA3);
        //BufferedWriter outNodes = new BufferedWriter(fstreamNodes);
        BufferedWriter outSig = new BufferedWriter(fstream);
        BufferedWriter outGen = new BufferedWriter(fstreamGen);
        BufferedWriter outWt = new BufferedWriter(fstreamWt);
        BufferedWriter outDur = new BufferedWriter(fstreamDur);

        ClHeadElection elect = new ClHeadElection(); //for CH election

        Random rnd = new Random();

        Scanner s = new Scanner(new File("100.txt"));
        Scanner sPrev = new Scanner(new File("100.txt"));

        Node[] nodeArr = new Node[nodeNo];  //creation of nodes
        for (i = 0; i < nodeNo; i++) {
            nodeArr[i] = new Node();
            //initialisation of node's trust vector
            nodeArr[i].initTrustVector();
            //so that everyone does not start transactions in t=1
            nodeArr[i].setWTime();
            outWt.write("\n" + String.valueOf(nodeArr[i].wTime));
        }

        for (i = 0; i < nodeNo; i++) {
            blackList[i] = -1;
        }
        // create a gateway connection
        try (Gateway gateway = builder.connect()) {

            // get the network and contract
            Network network = gateway.getNetwork("mychannel");
            Contract contract = network.getContract("manet");
//          network.getChannel().registerBlockListener(blockEvent -> {
//                      System.out.println("Block submitted");
//          });

            for (time = 1; time <= maxTime; time++) {
                //filling Node with id, x, kai y (x,y change)
                for (i = 0; i < nodeNo; i++) {
                    s.nextInt();
                    nodeArr[i].nodeId = s.nextInt();
                    nodeArr[i].x = s.nextInt();
                    nodeArr[i].y = s.nextInt();
                    //skips unnecessary mobisim data
                    s.nextInt();
                    s.nextInt();
                }
                //filling node with previous data. May not be used in the new version
                //version where election occurs based on cohesion
                for (i = 0; i < nodeNo; i++) {
                    if (time == 1) {
                        nodeArr[i].xPrev = nodeArr[i].x;
                        nodeArr[i].yPrev = nodeArr[i].y;
                        continue;
                    }
                    sPrev.nextInt();
                    sPrev.nextInt();
                    nodeArr[i].xPrev = sPrev.nextInt();
                    nodeArr[i].yPrev = sPrev.nextInt();
                    //skips unnecessary mobisim data
                    sPrev.nextInt();
                    sPrev.nextInt();
                }
                //health status. With some probability it changes in some nodes
                for (i = 0; i < nodeNo; i++) {
                    lastHS = nodeArr[i].healthStatus;
                    nodeArr[i].healthProbe(time);
                    if(!(lastHS == nodeArr[i].healthStatus)) {
                        contract.submitTransaction("changeNodeHS",
                            "NODE" + nodeArr[i].nodeId, String.valueOf(nodeArr[i].healthStatus));
                    }
                }
                //calculate the cost of analysis of every node
                for (i = 0; i < nodeNo; i++) {
                    if(time>1)
                        nodeArr[i].nodeRep =
                                nodeArr[nodeArr[i].clusterHeadId].trustVector[nodeArr[i].nodeId][0];
                    totRep += nodeArr[nodeArr[i].clusterHeadId].trustVector[i][0];
                }
                //the analysis cost is calculated by each node
                //separately because private information is required.
                //cost of analysis is not private
                for (i = 0; i < nodeNo; i++) {
                    nodeArr[i].costCalc(totRep);
                }

                System.out.println("\n[Time: " + time + "]");
    //print node energy deactivation: 50% faster!
    //            for (i = 0; i < nodeNo; i++) {
    //                System.out.println(nodeArr[i].nodeId
    //                        + " (" + nodeArr[i].nodeEnergy + ") ");
    //            }

                //It transports the old neighbors to the old neighbor array
                //so that they can be compared with the new neighbors in case of changes
                for (i = 0; i < nodeNo; i++) {
                    nodeArr[i].transferNeighbours();
                    nodeArr[i].initNeighbourArray();
                }

                //search for nodes that are within given range
                boxr = range; //it doesn't make sense for them to be different
                for (i = 0; i < nodeNo; i++) { //i is the current node
                    for (j = 0; j < nodeNo; j++) { //is the neighbor under consideration
                        if (i == j) {      //so that they are not looking for themselves
                            continue;
                        }

                        //compute dist only if node is in box
                        if (Math.abs(nodeArr[i].x - nodeArr[j].x) < boxr
                                && Math.abs(nodeArr[i].y) - nodeArr[j].y < boxr) {

                            dist = (float) Math.sqrt(Math.pow((nodeArr[i].x
                                    - nodeArr[j].x), 2)
                                    + Math.pow((nodeArr[i].y
                                    - nodeArr[j].y), 2));

                            distPrev = (float) Math.sqrt(Math.pow((nodeArr[i].xPrev
                                    - nodeArr[j].xPrev), 2)
                                    + Math.pow((nodeArr[i].yPrev
                                    - nodeArr[j].yPrev), 2));

                            //adds j node to i node's neighbor list
                            if (dist <= range) {
                                nodeArr[i].addNeighbour(nodeArr[j].nodeId,
                                        nodeArr[j].costOfAnalysis,
                                        dist, distPrev);
                            }
                        }//end of boxr
                    }//telos j
                }//telos i
                if (time == 1) {
                    continue;
                }
                //so far each node has been informed about its own position
                //and their neigbour array for available neighbours
                for (i = 0; i < nodeNo; i++) {
                    nodeArr[i].sortNeighbours();
                }
                //so far the neighbor tables are sorted by the cost of analysis
                //if there is an exit of any node that initially had a positive contribution
                //in the cohesion an election begins
                outerloop:
                for (i = 0; i < nodeNo; i++) {
                    if (time == 2) {
                        System.out.println("\nInitial CH election");
                        break;
                    }
                    //then each CH checks if any of the electors (those who initially
                    //contributed positively to the cohesion of the cluster), gets out of CH's range
                    //if it comes out, a new election round begins
                    if (nodeArr[i].isClHead == true) {
                        j = 0;
                        while ((int) nodeArr[i].electors[j] != -1) {
                            if ((float) Math.sqrt(Math.pow((nodeArr[i].x
                                - nodeArr[nodeArr[i].electors[j]].x), 2)
                                + Math.pow((nodeArr[i].y - nodeArr[nodeArr[i].electors[j]].y), 2))
                                < ((float) range)) {
                                j++;
                                continue;
                            }
                            electionFlag = true;
                            System.out.println("\nNode " + nodeArr[i].nodeId
                               + " calls for elections");
                            break outerloop;
                        }
                    }
                }
    //=============================== election ========================//
                //flag = true when neighbours change
                if (electionFlag == true) {
                    //resets in order to not continue from the previous value
                    clusterId = 1;
                    elect.initElect(); //initialise election table
                    //initialize nodes in case of any old values
                    for (i = 0; i < nodeNo; i++) {
                        nodeArr[i].electionReset();
                    }

                    for (i = 0; i < nodeNo; i++) {
                        //if it has no neighbors it votes for itself
                        if (nodeArr[i].countNeighbours() == 0) {
                            elect.elecTable[nodeArr[i].nodeId][1]++;
                            continue;
                        }
                        //if the cost of the previous neighbor(sorted)
                        //is less than its own
                        //it votes for the neighbor, otherwise it votes for itself
                        if (nodeArr[(int) nodeArr[i].neighbours[0][0]].costOfAnalysis
                                < nodeArr[i].costOfAnalysis) {
                            elect.elecTable[(int) nodeArr[i].neighbours[0][0]][1]++;
                        } else {
                            elect.elecTable[nodeArr[i].nodeId][1]++;
                        }
                    }

                    elect.sortElecTable(); //sort election table

                    //cluster creation: all nodes that have as first neighbor
                    //the first node of the elekTable means that they voted
                    //for him therefore they will get the same cluster id
                    //with the first node as the clusterhead
```

```
                    for (k = 0; k < nodeNo; k++) {
                        //in combination with delRow
                        if (elect.elecTableSorted[k] == -1) {
                            //ignored as they already belong in a cluster
                            continue;
                        }
                        cH = elect.elecTableSorted[k]; //CH election
                        nodeArr[cH].setClusterHeadId(cH); //defining itself
                        nodeArr[cH].setHead(); //sethead = true
                        //defining CH's clusterid
                        nodeArr[cH].setClusterId(clusterId);

                        //whoever has as first neighbour the CH
                        //becomes a member of the cluster
                        for (i = 0; i < nodeNo; i++) {
                            if ((int) nodeArr[i].neighbours[0][0] == cH
                                    && nodeArr[i].isClHead == false) {
                                nodeArr[i].setClusterId(clusterId);
                                nodeArr[i].setClusterHeadId(cH);
                            }
                        }

                        for (i = 0; i < nodeNo; i++) {
                            if (nodeArr[i].clusterId == clusterId) {
                                for (j = 0; j < nodeNo; j++) {
                                    if (nodeArr[j].clusterId == clusterId
                                        && nodeArr[j].isClHead == true) {
                                        //set cluster in the BC
                                    contract.submitTransaction("setCluster",
                                        "NODE"+nodeArr[j].nodeId, String.valueOf(clusterId),
                                        String.valueOf(nodeArr[j].nodeId));
                                        break;
                                    }
                                }
                                for (i = 0; i < nodeNo; i++) {
                                    if (nodeArr[i].clusterId == clusterId
                                            && nodeArr[i].isClHead == false) {
                                        //set cluster in the BC
                                        contract.submitTransaction("setCluster",
                                            "NODE" + nodeArr[i].nodeId, String.valueOf(clusterId),
                                            String.valueOf(nodeArr[i].clusterHeadId));
                                    }
                                }
                            }
                        } //end of defining clusters

//    ------------------------Print cluster------------------------
//                        for (i = 0; i < nodeNo; i++) { //ektiposi clusterhead
//                            if (nodeArr[i].clusterId == clusterId) {
//                                System.out.println("\nCluster Id: " + clusterId);
//                                for (j = 0; j < nodeNo; j++) {
//                                    if (nodeArr[j].clusterId == clusterId
//                                            && nodeArr[j].isClHead == true) {
//                                        System.out.println("Clusterhead Id: "
//                                                + nodeArr[j].nodeId);
//                                        break;
//                                    }
//                                }
//                                //print cluster members
//                                System.out.print("Cluster member Ids: ");
//                                for (i = 0; i < nodeNo; i++) {
//                                    if (nodeArr[i].clusterId == clusterId
```

```java
//                                            && nodeArr[i].isClHead == false) {
//                                        System.out.print(nodeArr[i].nodeId + " ");
//                                    }
//                                }
//                                System.out.print("\n");
//                            }
//                        }

                        //delete the election table of the specific nodes of the cluster
                        //in order to be ready for the next election
                        for (i = 0; i < nodeNo; i++) {
                            if (nodeArr[i].clusterId == clusterId) {
                                elect.delSRow(nodeArr[i].nodeId);
                            }
                        }
                        clusterId++;
                    }

                    //update cluster neighbour array for each node
                    //1-hop neighbours
                    //delete previous cluster neighbours
                    for (i = 0; i < nodeNo; i++) {
                        nodeArr[i].clearClNeighbours();
                    }
                    //update the array with the new cluster neighbours
                    //i index for the node to update
                    for (i = 0; i < nodeNo; i++) {
                        j = 0; //j index for the cluster neighbour array
                        //k index for the neighbour under consideration
                        for (k = 0; k < nodeNo; k++) {
                            if (nodeArr[i].neighbours[k][0] == -1) {
                                break;
                            }
                            if (nodeArr[(int) nodeArr[i].neighbours[k][0]].clusterId
                                    == nodeArr[i].clusterId) {
                                nodeArr[i].clusterNeighbours[j]
                                        = (int) nodeArr[i].neighbours[k][0];
                                j++;
                            }
                        }
                    }

                    //end of cluster array update
                    //elect.printElecTable();
                    elect.elecNo++; //to check how many elections occur
                    electionFlag = false;

    //from here on out update the trust array
    //=============================================================================
                    clusterCount = 0;
                    //how many clusters
                    for (i = 0; i < nodeNo; i++) {
                        if (nodeArr[i].isClHead) {
                            //to ensure that the updated CH are in a block
                            //failsafe for the queries
                            while (!"true".equals(new String(contract.evaluateTransaction("isCH",
                                                    "NODE" + nodeArr[i].nodeId)))) {
                                contract.submitTransaction("setCluster", "NODE" + nodeArr[i].nodeId,
                                            String.valueOf(nodeArr[i].clusterId),
                                            String.valueOf(nodeArr[i].nodeId));
                            }
                            clusterCount++;
                        }
                    }

                    for (i = 1; i <= clusterCount; i++) { //for every cluster i
                        //tempTrustVector = 0
                        for (l = 0; l < nodeNo; l++) {
                            tempTrustVector[l] = 0;
                        }
                        //counts nodes (minus CH) for the current i cluster
                        for (l = 0; l < nodeNo; l++) {
                            if (nodeArr[l].clusterId == i && nodeArr[l].isClHead == true) {
                                //CH's cl neighbours
                                clusterNodeNo = nodeArr[l].countClNeighbours();
                                break;
                            }
                        }
                        //if CH has no neighbours, its LTV remmains the same
                        if (clusterNodeNo != 0) {
                            //examines node k if it belongs in i cluster
                            for (k = 0; k < nodeNo; k++) {
                                if (nodeArr[k].clusterId == i && nodeArr[k].isClHead == false) {
                                    //LTV components to tempTrustVector
                                    for (l = 0; l < nodeNo; l++) {
                                        tempTrustVector[l] += nodeArr[k].trustVector[l][0];
                                    } //end of transfer of k node's trust
                                } //end of loop if it is cl member but no CH
                            } //end of loop for all cluster members
                            //tempTrustVector to CH LTV
                            for (k = 0; k < nodeNo; k++) {
                                if (nodeArr[k].clusterId == i && nodeArr[k].isClHead == true) {
                                    for (l = 0; l < nodeNo; l++) {
                                        nodeArr[k].trustVector[l][0]
                                                = (1.0f - aLPF) * nodeArr[k].trustVector[l][0]
                                                + (aLPF * (tempTrustVector[l]
                                                / clusterNodeNo));
                                        //no more than maxtrust
                                        if (nodeArr[k].trustVector[l][0] > maxTrust) {
                                            nodeArr[k].trustVector[l][0] = maxTrust;
                                        }
                                        //no less than mintrust
                                        if (nodeArr[k].trustVector[l][0] < minTrust) {
                                            nodeArr[k].trustVector[l][0] = minTrust;
                                        }
                                        nodeArr[k].trustVector[l][1] = time; //timestamp
                                    }
                                    //initiates tempTrustVector
                                    for (l = 0; l < nodeNo; l++) {
                                        tempTrustVector[l] = 0;
                                    }

                                    for(int n : nodeArr[k].clusterNeighbours){
                                        if(n == -1)
                                            break;
                                        //equivalent of GTV
                                        contract.submitTransaction("changeNodeTrust",
                                                "NODE"+nodeArr[k].nodeId, "NODE"+nodeArr[n].nodeId,
                                                String.valueOf(nodeArr[k].trustVector[n][0]));
                                    }
                                }
                            }
                        }
                    }//end of loop for all clusters
```

```
                    for (i = 0; i < nodeNo; i++) {
                        if (nodeArr[i].isClHead == true) {
                            nodeArr[i].initElectCopy();
                            int elec = 0;
                            k = 0;
                            while (nodeArr[i].neighbours[k][0] != -1) {
                                if (nodeArr[i].clusterId ==
                                        nodeArr[(int) nodeArr[i].neighbours[k][0]].clusterId
                                        && nodeArr[i].neighbours[k][4] == 1) {
                                    nodeArr[i].electors[elec] = (int) nodeArr[i].neighbours[k][0];
                                    k++;
                                    elec++;
                                    continue;
                                }
                                k++;
                            }
                        }
                    }
                }
    //-------------------------- end of election ------------------------//
                System.out.println("Start of transaction phase");
//                result = contract.evaluateTransaction("queryAllNodes");
//                System.out.println(new String(result));
    //-------------------------- start transaction------------------------//
                for (i = 0; i < nodeNo; i++) { //for every node i

                    if (nodeArr[i].nodeEnergy <= nodeArr[i].nodeEnergyThreshold) {
                        System.out.println("Node " + nodeArr[i].nodeId
                                + " is out of energy");
                        continue; //not enough energy
                    }
                    //if there is any remaining wTime, the node is waiting
                    if (nodeArr[i].wTime != 0) {
                        nodeArr[i].decrDelay(); //reduce wTime by 1
                        //if there is waiting time, next node
                        if (nodeArr[i].wTime != 0) {
                            continue;
                        }
                    }

                    //if available, with cluster neighbours
                    //(and the waiting time has passed)
                    if (nodeArr[i].avClient == true
                            && nodeArr[i].countClNeighbours() != 0) {
                        //a random neighbour is selected as a candidate
                        //checks if available
                        candServer = nodeArr[i].clusterNeighbours[rnd.nextInt(nodeArr[i].countClNeighbours())];
                        if (nodeArr[candServer].avServer == false
                                 nodeArr[candServer].nodeEnergy
                                < nodeArr[candServer].nodeEnergyThreshold) {
                            System.out.print(nodeArr[i].nodeId + " --X--> "
                                    + candServer + "\tServer unavailable.");
                            nodeArr[i].setWTime();
                            outWt.write("\n" + String.valueOf(nodeArr[i].wTime));
                            System.out.println("Delay set for " + nodeArr[i].nodeId
                                    + ": " + nodeArr[i].wTime);
                            continue;
                        } else { //server is available so a CH consulatation is in order
//                            nodeArr[i].LTVupdate(candServer,
//                                    nodeArr[nodeArr[i].clusterHeadId].
//                                            CH_Report(candServer), time,
//                                    nodeArr[nodeArr[i].clusterHeadId].CH_Timestamp(candServer));
```

```java

                //query BC
                result = contract.evaluateTransaction("queryNode",
                                "NODE"+nodeArr[nodeArr[i].clusterHeadId].nodeId,
                                "NODE"+nodeArr[candServer].nodeId);
                JSONObject json = (JSONObject) parser.parse(new String(result));
                float trustScore =
                        BigDecimal.valueOf((double) json.get("trustScore")).floatValue();

                if (trustScore < 0.5) {
                    //2nd chance
                    trustScore = nodeArr[nodeArr[i].clusterHeadId].trustVector[candServer][0]
                                                                + rnd.nextFloat() / 4f;
                    contract.submitTransaction("changeNodeTrust",
                                "NODE"+nodeArr[nodeArr[i].clusterHeadId].nodeId,
                                "NODE"+nodeArr[candServer].nodeId,
                                String.valueOf(trustScore));
                }

                nodeArr[i].LTVupdate(candServer, trustScore, time,
                        nodeArr[nodeArr[i].clusterHeadId].CH_Timestamp(candServer));

                if (nodeArr[i].trustVector[candServer][0]
                        >= trustThreshold) {
                    nodeArr[i].serverId = candServer;
                    nodeArr[i].durLeft
                            = (int) ((Math.log(1 - rnd.nextFloat()))
                            / ((-1) * lamda)) + 1;
                    //stores duration for trust calculations
                    nodeArr[i].setDur();
                    outDur.write("\n" + String.valueOf(nodeArr[i].dur));
                    //stores the time the transaction begun
                    nodeArr[i].setTransStart(time);
                    //binds ton server
                    nodeArr[nodeArr[i].serverId].avServer = false;
                    //binds ton client
                    nodeArr[i].avClient = false;
                    if (blackList[candServer] != -1) {
                        outA3.write("\n"
                                + String.valueOf(time - blackList[candServer]));
                        blackList[candServer] = -1;
                    }
                    System.out.println("\n" + nodeArr[i].nodeId + " -----> "
                            + nodeArr[i].serverId
                            + "\tTransaction started. Duration: "
                            + nodeArr[i].durLeft);

                    //in case of malicious server, stores the time
                    //that health was last changed
                    if (nodeArr[nodeArr[i].serverId].healthStatus == 0
                            && nodeArr[i].serverMalTS == -1) {
                        nodeArr[i].serverMalTS = nodeArr[nodeArr[i].serverId].getMalTimeStamp();
                    }
                    tStarted++;
                    continue;
                } else {
                    System.out.println("Server trust too low, "
                            + "aborting transaction");

                    contract.submitTransaction("addTransactionClient",
                                    "NODE"+nodeArr[nodeArr[i].clusterHeadId].nodeId,
                                    "NODE"+nodeArr[i].nodeId,
                                    String.valueOf(nodeArr[candServer].nodeId), "0", "ABORTED");
                    contract.submitTransaction("addTransactionServer",
                                    "NODE"+nodeArr[nodeArr[i].clusterHeadId].nodeId,
                                     String.valueOf(nodeArr[i].nodeId),
                                    "NODE"+nodeArr[candServer].nodeId, "0", "ABORTED");

                    blackList[candServer] = time;
                    if (nodeArr[candServer].healthStatus == 0) {
                        outA2.write("\n" + String.valueOf(time
                                - nodeArr[candServer].malTimeStamp));
                    } else {
                        falseMal++;
                    }

                    trustFailed++;
                    nodeArr[i].setWTime();
                    nodeArr[i].eLTV_restore();
                    outWt.write("\n" + String.valueOf(nodeArr[i].wTime));
                }
            } //end of available server
        } //end of available client

        //if unavailable server or client, then they are in a transaction
        //id there is serverId, the node is client
        //so if its server is within range,
        //reduces the transaction duration by 1.
        //otherwise, it stops the transaction and resets the nodes
        if (nodeArr[i].avClient == false && nodeArr[i].serverId
                != -1) {
            if (nodeArr[i].isClNeighbour(nodeArr[i].serverId)) {
                //reduces the remaining duration by 1
                nodeArr[i].durLeft--;
                //reduces the client's energy
                nodeArr[i].decEnergy();
                //reduces the server's energy
                nodeArr[nodeArr[i].serverId].decEnergy();
                if (nodeArr[i].nodeEnergy < nodeArr[i].nodeEnergyThreshold
                         nodeArr[nodeArr[i].serverId].nodeEnergy
                        < nodeArr[nodeArr[i].serverId].nodeEnergyThreshold) {
                    System.out.println("Transaction failled "
                            + "due to energy lack");

                    contract.submitTransaction("addTransactionClient",
                                    "NODE"+nodeArr[nodeArr[i].clusterHeadId].nodeId,
                                    "NODE"+nodeArr[i].nodeId,
                                    String.valueOf(nodeArr[i].serverId),
                                    String.valueOf(0), "FAILED_ENERGY");

                    rcontract.submitTransaction("addTransactionServer",
                                    "NODE"+nodeArr[nodeArr[i].clusterHeadId].nodeId,
                                    String.valueOf(nodeArr[i].nodeId),
                                    "NODE"+nodeArr[i].serverId,
                                    String.valueOf(0), "FAILED_ENERGY");

                    fTrans++;
                    nodeArr[nodeArr[i].serverId].resetServer();
                    nodeArr[i].resetClient();
                    nodeArr[i].eLTV_restore();
                    continue;
                }
                //to record malicious server IF it becomes malicious
                //during the transaction
```

```java
                            if (nodeArr[nodeArr[i].serverId].healthStatus == 0
                                    && nodeArr[i].serverMalTS == -1) {
                                nodeArr[i].serverMalTS = nodeArr[nodeArr[i].serverId].getMalTimeStamp();
                            }
                            if (nodeArr[nodeArr[i].serverId].healthStatus == 0) {
                                //increase malicious time counter by 1
                                nodeArr[i].malTimeCounter++;
                            }
                            //dur=0 => end of transaction
                            if (nodeArr[i].durLeft == 0) {
                                System.out.print("\n" + nodeArr[i].nodeId
                                        + "--X-->" + nodeArr[i].serverId
                                        + "\tTransaction completed successfully.");
                                System.out.println("\nEnergy levels: "
                                        + nodeArr[i].nodeId + ": "
                                        + nodeArr[i].nodeEnergy + "\t"
                                        + nodeArr[i].serverId + ": "
                                        + nodeArr[nodeArr[i].serverId].nodeEnergy);

//==================A1 metrics==============================//
                                if (nodeArr[i].serverMalTS != -1
                                        && (float) (nodeArr[i].malTimeCounter
                                        / nodeArr[i].dur) > 0.5) {
                                //we count the time from the most recent G2B
                                    outA1.write("\n" + String.valueOf(time
                                            - nodeArr[i].serverMalTS));
                                    mTrans++;
                                }

//-------------Evaluation----------
                                evaluation = (float) (rnd.nextGaussian() * 0.1 + 0.5);
                                if (evaluation > 1) { //no more than 1
                                    evaluation = 1;
                                }
                                if (evaluation < 0) { //no les than 0
                                    evaluation = 0;
                                }

                                if (((float) nodeArr[i].malTimeCounter
                                        / nodeArr[i].dur) > 0.5) {
                                }
                                if (((float) nodeArr[i].malTimeCounter
                                        / nodeArr[i].dur) > 0.5
                                        && evaluation > 0.5) {
                                    evaluation -= 0.5; //penalty for malicious
                                }

                                contract.submitTransaction("addTransactionClient",
                                                    "NODE"+nodeArr[nodeArr[i].clusterHeadId].nodeId,
                                                    "NODE"+nodeArr[i].nodeId,
                                                    String.valueOf(nodeArr[i].serverId),
                                                    String.valueOf(evaluation), "SUCCESSFUL");
                                contract.submitTransaction("addTransactionServer",
                                                    "NODE"+nodeArr[nodeArr[i].clusterHeadId].nodeId,
                                                    "NODE"+nodeArr[i].serverId,
                                                    String.valueOf(evaluation), "SUCCESSFUL");

                                outSig.write("\n" + String.valueOf(evaluation));
//-----------------------------------------------

                                nodeArr[i].LTVupdate(nodeArr[i].serverId, evaluation, time, time);
                                nodeArr[nodeArr[i].clusterHeadId].report(nodeArr[i].serverId, evaluation, time);
```

```

                                    contract.submitTransaction("changeNodeTrust",
                                                    "NODE" + nodeArr[nodeArr[i].clusterHeadId].nodeId,
                                                    "NODE" + nodeArr[i].serverId,
                                                    String.valueOf(nodeArr[nodeArr[i].clusterHeadId]
                                                                .trustVector[nodeArr[i].serverId][0]));
                                    //
                                    // To update all CH using some epidemic algorithm
                                    //
                                    for (j = 1; j < nodeNo; j++) {
                                        if (j == i) continue;
                                        nodeArr[nodeArr[j].clusterHeadId].report(nodeArr[i].serverId,
                                                                             evaluation, time);
                                    }

                                    sTrans++;
                                    //reset
                                    nodeArr[nodeArr[i].serverId].resetServer();
                                    nodeArr[i].resetClient();
                                    nodeArr[i].eLTV_restore();
                                    nodeArr[i].setWTime();
                                    outWt.write("\n" + String.valueOf(nodeArr[i].wTime));
                                    System.out.println("Delay set for "
                                            + nodeArr[i].nodeId + ": " + nodeArr[i].wTime);

                                }
                            } else {//out of range...
                                System.out.print(nodeArr[i].nodeId
                                        + " --X--> " + nodeArr[i].serverId
                                        + "\tTransaction failed (server out of reach) ");

                                contract.submitTransaction("addTransactionClient",
                                                    "NODE"+nodeArr[nodeArr[i].clusterHeadId].nodeId,
                                                    "NODE"+nodeArr[i].nodeId,
                                                    String.valueOf(nodeArr[i].serverId),
                                                    "0", "FAILED_RANGE");
                                contract.submitTransaction("addTransactionServer",
                                                    "NODE"+nodeArr[nodeArr[i].clusterHeadId].nodeId,
                                                    String.valueOf(nodeArr[i].nodeId),
                                                    "NODE"+nodeArr[i].serverId, "0", "FAILED_RANGE");

                                fTrans++;
                                if (nodeArr[i].serverMalTS != -1) {
                                    System.out.println("Node " + i + " should know that node "
                                            + nodeArr[i].serverId + " was malicious"
                                            + (time - nodeArr[i].malTimeStamp)
                                            + " time slots ago...");
                                }
                                //reset
                                nodeArr[nodeArr[i].serverId].resetServer();
                                nodeArr[i].resetClient();
                                nodeArr[i].setWTime();
                                outWt.write("\n" + String.valueOf(nodeArr[i].wTime));
                                System.out.println(" Delay set for " + nodeArr[i].nodeId
                                        + ": " + nodeArr[i].wTime);
                            }
                        }
                    }
                }//end time loop
                //for debugging when time <3000
                //after that the message beacomes too large
                //result = contract.evaluateTransaction("queryAllNodes");
```

```
            //outNodes.write("\n" + new String(result));
        }

        //statistics
        System.out.println("Range: " + range);
        System.out.println("Elections: " + elect.elecNo);
        System.out.printf("Mean time between elections: %.2f",
                ((float) (time - 1) / elect.elecNo));
        System.out.println("\nTransactions started: " + tStarted);
        System.out.println("Succesfull Transactions: " + sTrans);
        System.out.println("Failed Transactions: " + fTrans);
        System.out.println("Didn't start due to low trust: "
                + trustFailed);
        System.out.println("Percentage of malicious transactions "
                + (float) ((mTrans / (sTrans + fTrans))));
        System.out.println("Malicious nodes at the end of the simulation: ");
        for (i = 0; i < nodeNo; i++) {
            if (nodeArr[i].healthStatus == 0) {
                System.out.println(nodeArr[i].nodeId);
            }
        }
        /*System.out.println("Healthy nodes at the end of the simulation: ");
        for (i = 0; i < nodeNo; i++) {
            if (nodeArr[i].healthStatus == 1) {
                System.out.println(nodeArr[i].nodeId);
            }
        }*/
        System.out.println("Nodes out of energy");
        for (i = 0; i < nodeNo; i++) {
            if (nodeArr[i].nodeEnergy <= 10) {
                System.out.println(nodeArr[i].nodeId);
            }
        }
        System.out.println("\nA1 metrics");
        System.out.println("Succesfull Transactions: " + sTrans);
        System.out.println("Total Malicious Transactions: "
                + String.valueOf(mTrans));
        System.out.println("Percentage: "
                + String.valueOf((float) mTrans
                / (mTrans + sTrans) * 100) + "%");
        System.out.println("\nA2 metrics");
        System.out.println("Total transactions that didn't start "
                + "due to low trust: "
                + trustFailed);
        System.out.println("False Malicious: " + String.valueOf(falseMal));
        System.out.println("Percentage: "
                + String.valueOf((float) falseMal
                / trustFailed * 100) + "%");
        outGen.write("\nRange: " + String.valueOf(range));
        outGen.write("\nElections: " + String.valueOf(elect.elecNo));
        outGen.write("\nMean time between elections: "
                + String.valueOf(((float) (time - 1) / elect.elecNo)));
        outGen.write("\nTransactions started: "
                + String.valueOf(tStarted));
        outGen.write("\nSuccesfull Transactions: "
                + String.valueOf(sTrans));
        outGen.write("\nFailed Transactions: "
                + String.valueOf(fTrans));
        outGen.write("\nDidn't start due to low trust: "
                + String.valueOf(trustFailed));
        outGen.write("\nPercentage of malicious transactions "
                + String.valueOf((float) mTrans / (sTrans + fTrans)));
```

```java
        outGen.write("\nMalicious nodes at the end of the simulation: ");
        for (i = 0; i < nodeNo; i++) {
            if (nodeArr[i].healthStatus == 0) {
                outGen.write("\n" + String.valueOf(nodeArr[i].nodeId));
            }
        }
        outGen.write("\nHealthy nodes at the end of the simulation: ");
        for (i = 0; i < nodeNo; i++) {
            if (nodeArr[i].healthStatus == 1) {
                outGen.write("\n" + String.valueOf(nodeArr[i].nodeId));
            }
        }
        outGen.write("\nNodes out of energy");
        for (i = 0; i < nodeNo; i++) {
            if (nodeArr[i].nodeEnergy <= 10) {
                outGen.write("\n" + String.valueOf(nodeArr[i].nodeId));
            }
        }
        outGen.write("\nA1 metrics");
        outGen.write("\nSuccesfull Transactions: "
                + String.valueOf(sTrans));
        outGen.write("\nTotal Malicious Transactions: "
                + String.valueOf(mTrans));
        outGen.write("\nPercentage: "
                + String.valueOf((float) mTrans
                / (sTrans + mTrans) * 100) + "%");
        outGen.write("\nA2 metrics");
        outGen.write("\nTotal transactions that didn't start "
                + "due to low trust: "
                + String.valueOf(trustFailed));
        outGen.write("\nFalse Malicious: " + String.valueOf(falseMal));
        outGen.write("\nPercentage: " + String.valueOf((float) falseMal
                / trustFailed * 100) + "%");

        System.out.println("\nEnd");
        outA1.close();
        outA2.close();
        outA3.close();
        //outNodes.close();
        outWt.close();
        outDur.close();
        outSig.close();
        outGen.close();
        s.close();
        sPrev.close();
    } //end main
}//end class
```

B.6 ClientTest

```java
import org.hyperledger.fabric.gateway.Contract;
import org.hyperledger.fabric.gateway.ContractException;
import org.hyperledger.fabric.gateway.Gateway;
import org.hyperledger.fabric.gateway.Network;
import org.junit.Test;
import java.nio.charset.StandardCharsets;
import java.util.concurrent.TimeoutException;

public class ClientTest {
    @Test
    public void testManet() throws Exception {
        EnrollAdmin.main(null);
        RegisterUser.main(null);
        ClientApp.main(null);
    }
}
```

Appendix C

Benchmark Files

C.1 Test Configuration

```yaml
test:
  workers:
    number: 5
  rounds:
    - label: Set cluster.
      txDuration: 30
      rateControl:
          type: fixed-load
          opts:
            transactionLoad: 5
      workload:
        module: benchmarks/samples/fabric/manet/setCluster.js
        arguments:
          assets: 500
    - label: ChangeNodeTrust.
      txDuration: 30
      rateControl:
          type: fixed-load
          opts:
            transactionLoad: 5
      workload:
        module: benchmarks/samples/fabric/manet/changeNodeTrust.js
        arguments:
          assets: 500
    - label: ChangeNodeHS.
      txDuration: 30
      rateControl:
          type: fixed-load
          opts:
            transactionLoad: 5
      workload:
        module: benchmarks/samples/fabric/manet/changeNodeHS.js
        arguments:
          assets: 500
    - label: Query a node.
      txDuration: 30
      rateControl:
          type: fixed-load
          opts:
```

```
          transactionLoad: 5
      workload:
        module: benchmarks/samples/fabric/manet/queryNode.js
        arguments:
          assets: 500
    - label: Add client transaction.
      txDuration: 30
      rateControl:
          type: fixed-load
          opts:
            transactionLoad: 5
      workload:
        module: benchmarks/samples/fabric/manet/addTransactionClient.js
        arguments:
          assets: 500
    - label: Add server transaction.
      txDuration: 30
      rateControl:
          type: fixed-load
          opts:
            transactionLoad: 5
      workload:
        module: benchmarks/samples/fabric/manet/addTransactionServer.js
        arguments:
          assets: 500
```

C.2 QueryNode

```
'use strict';

const { WorkloadModuleBase } = require('@hyperledger/caliper-core');

/**
 * Workload module for the benchmark round.
 */
class QueryNodeWorkload extends WorkloadModuleBase {
    /**
     * Initializes the workload module instance.
     */
    constructor() {
        super();
        this.txIndex = 0;
        this.limitIndex = 0;
    }

    /**
     * Initialize the workload module with the given parameters.
     * @param {number} workerIndex The 0-based index of the worker instantiating the workload module.
     * @param {number} totalWorkers The total number of workers participating in the round.
     * @param {number} roundIndex The 0-based index of the currently executing round.
     * @param {Object} roundArguments The user-provided arguments for the round from the benchmark config file.
     * @param {BlockchainInterface} sutAdapter The adapter of the underlying SUT.
     * @param {Object} sutContext The custom context object provided by the SUT adapter.
     * @async
     */
    async initializeWorkloadModule(workerIndex, totalWorkers, roundIndex, roundArguments,
                                    sutAdapter, sutContext) {
        await super.initializeWorkloadModule(workerIndex, totalWorkers,
                                roundIndex, roundArguments, sutAdapter, sutContext);

        this.limitIndex = this.roundArguments.assets;
    }

    /**
     * Assemble TXs for the round.
     * @return {Promise<TxStatus[]>}
     */
    async submitTransaction() {
        this.txIndex++;
        let nodeNumber = 'Client' + this.workerIndex + '_NODE' + this.txIndex.toString();

        let args = {
            contractId: 'manet',
            contractVersion: 'v1',
            contractFunction: 'queryNode',
            contractArguments: [nodeNumber],
            timeout: 30,
            readOnly: true
        };

        if (this.txIndex === this.limitIndex) {
            this.txIndex = 0;
        }

        await this.sutAdapter.sendRequests(args);
    }
}

/**
 * Create a new instance of the workload module.
 * @return {WorkloadModuleInterface}
 */
function createWorkloadModule() {
    return new QueryNodeWorkload();
}

module.exports.createWorkloadModule = createWorkloadModule;
```

C.3 ChangeNodeTrust

```
'use strict';

const { WorkloadModuleBase } = require('@hyperledger/caliper-core');

const trustNo = ["0.1", "0.2", "0.3", "0.4", "0.45", "0.5", "0.55",
                    "0.6", "0.65", "0.7", "0.8" , "0.9"];

class ChangeNodeTrustWorkload extends WorkloadModuleBase {

    constructor() {
        super();
        this.txIndex = 0;
    }

    async submitTransaction() {
        this.txIndex++;
        let nodeNumber = 'Client' + this.workerIndex + '_NODE' + this.txIndex.toString();
        let newNodeTrust = trustNo[Math.floor(Math.random() * trustNo.length)];

        let args = {
            contractId: 'manet',
            contractVersion: 'v1',
            contractFunction: 'changeNodeTrust',
            contractArguments: [nodeNumber, newNodeTrust],
            timeout: 60
        };

        if (this.txIndex === this.roundArguments.assets) {
            this.txIndex = 0;
        }

        await this.sutAdapter.sendRequests(args);
    }
}

function createWorkloadModule() {
    return new ChangeNodeTrustWorkload();
}

module.exports.createWorkloadModule = createWorkloadModule;
```

C.4 AddTransactionClient

```
'use strict';

const { WorkloadModuleBase } = require('@hyperledger/caliper-core');

const evals = ["0.1", "0.2", "0.3", "0.4", "0.45", "0.5", "0.55",
                    "0.6", "0.65", "0.7", "0.8", "0.9"];
const stats = ["SUCCESSFUL", "FAILED_ENERGY", "FAILED_RANGE", "ABORTED"];
const ids = ["1","5","10","28","15","16","22","27","46","74","87",
                "93","48","34","65","85","43","12","39","21"];

class AddTransactionClientWorkload extends WorkloadModuleBase {

    constructor() {
        super();
        this.txIndex = 0;
    }

    async submitTransaction() {
        this.txIndex++;
        let nodeNumber = 'Client' + this.workerIndex + '_NODE' + this.txIndex.toString();
        let evaluation = evals[Math.floor(Math.random() * evals.length)];
        let status = stats[Math.floor(Math.random() * stats.length)];
        let id = ids[Math.floor(Math.random() * ids.length)];

        let args = {
            contractId: 'manet',
            contractVersion: 'v1',
            contractFunction: 'addTransactionClient',
            contractArguments: [nodeNumber, id, evaluation, status],
            timeout: 60
        };

        if (this.txIndex === this.roundArguments.assets) {
            this.txIndex = 0;
        }

        await this.sutAdapter.sendRequests(args);
    }
}

function createWorkloadModule() {
    return new AddTransactionClientWorkload();
}

module.exports.createWorkloadModule = createWorkloadModule;
```