Turkish Journal of Electrical Engineering & Computer Sciences

http://journals.tubitak.gov.tr/elektrik/

Research Article

Turk J Elec Eng & Comp Sci
(2022) 30: 450 – 468
© TÜBİTAK
doi:10.3906/elk-2103-105

Data immutability and event management via blockchain in the Internet of things

Hakan ALTAŞ[1,2], **Gökhan DALKILIÇ**[2], **Umut Can ÇABUK**[3,*]
[1]Graduate School of Natural and Applied Sciences, Dokuz Eylül University, İzmir, Turkey
[2]Department of Computer Engineering, Dokuz Eylül University, İzmir, Turkey
[3]International Computer Institute, Ege University, İzmir, Turkey

Received: 23.03.2021 • **Accepted/Published Online:** 25.12.2021 • **Final Version:** 04.02.2022

Abstract: The Internet of things (IoT) is the key enabler of the smart systems used in many areas, from agriculture to aviation, industrial automation to autonomous vehicles. Most IoT deployments employ cost-efficient lightweight devices with limited resources (e.g., bandwidth, energy, storage). Although an IoT network must be built in its simplest form, engineers include more sophisticated devices like gateways and servers to provide web-based services and benefit from cloud systems. So, although the nodes can be widely distributed geographically or topologically, the system becomes centralized, which causes bottlenecks and single-points-of-failure. Furthermore, providing data integrity, nonrepudiation, and event management becomes tricky. In most IoT scenarios, data usually flow from sensors to storage and processing units, whereas event-driven commands and triggers flow from these units to actuators, if any. Therefore, an attacker who gained access to parts of the centralized systems may leak, alter, or remove critical data and may exploit event handling features. This is where blockchain technology can be extremely useful. Using a decentralized ledger as the data storage unit provides integrity, immutability, and nonrepudiation for any IoT deployment. And a customized smart contract lets the IoT deployment benefit from decentralized and immutable (i.e. nonmanipulatable) event management features, too. Further, decentralization provides resilience against availability attacks to a large extent. With this motivation, we introduced a novel IoT architecture that incorporates an Ethereum-based private (Quorum) blockchain running a unique ad-hoc smart contract and a message queue telemetry transport (MQTT) based communication scheme between sensor and actuator nodes. The scheme, the ledger, and the smart contract have also been implemented with several nodes, a broker, and a server all on a PC using Docker containers, where the server was running a forest fire risk detection algorithm as the use case scenario. The proof-of-concept successfully validates the abovesaid functionality, scalability, and efficiency for the given IoT scenario (and some others). Moreover, performance tests showed that an instance of the system with 1000 nodes could stably process and record incoming (sensor) data up to 12.5 transactions per second (TPS) and distribute commands up to 4 TPS, whereas higher TPS is achievable depending on the network conditions and tolerance to losses. The scheme was shown to have polynomial message and time complexity.

Key words: Blockchain, Ethereum, immutability, IoT security, smart contract

1. Introduction

In many Internet of things (IoT) ecosystems, the integration of intermediary subsystems, also called IoT platforms, has become prevalent. They are the intersection points where data reporting IoT devices (e.g., sensors) send their messages, and data collecting devices (e.g., actuators) pull/query new messages, representing a sort of specialized gateway. Such sensor data are becoming the main enabler of handling event management, activity triggers, advanced event processing, or even big data analyses [1].

*Correspondence: umut.can.cabuk@ege.edu.tr

This work is licensed under a Creative Commons Attribution 4.0 International License.

Analyzing big data collections, handling event management, and advanced event processing become integral functions of IoT deployments, while these activities are also supported by the IoT platforms. Within IoT platforms, dealing with various events and further processing of such are being done (i) to drive or trigger other IoT devices with actuator capabilities by sending appropriate messages, and (ii) to perform big data analysis for predicting the future behaviors of systems and nature, or obtaining meaningful patterns on the collected data set. Therefore, it can be said that sensor values are processed by an IoT platform to generate actuation messages that regulate the actuator devices' further operation.

Event management processes bring new challenges to the field, including severe security threats; thus, the entire system (as well as other interconnected systems) may be at risk of infiltration, eavesdropping, failures, or malfunctioning. Furthermore, modern IoT implementations with extended capabilities (driven by digital transformation efforts) may require larger network bandwidths. Data inaccuracy and hardware bottlenecks should also be mentioned as general problems in the IoT ecosystems. Among the aforementioned issues, infiltration attacks (of various kinds) stand at a different point from a data-oriented perspective because they, in theory, can cause manipulation or loss of the original data that is being transferred through the network or stored in a database unit without even being detected for a long while. Whereas malfunctioning is usually detectable, and eavesdropping does not cause loss of original data. Hence, such risks require specialized solutions that involve immutability and nonrepudiation features.

Most large IoT deployments are all about scalability. Therefore, addressing security issues must be done via scalable and efficient solutions. Widespread adoption of fifth-generation cellular communication systems (5G) may help mitigate scalability and efficiency problems [2]. Nevertheless, the developers and the producers of IoT systems did not yet agree on universally applicable approaches and standards for security challenges. Hence, some costly countermeasures may be implemented on-demand, such as trusted third parties, private certificate authorities, and additional hardware layers.

This work focuses on addressing portions of these security challenges by utilizing blockchain technology as a communication and data storage medium. That is to say, we explain and demonstrate how the risks of facing manipulation or loss of sensor data, as well as loss of control in the event management process caused by a potential infiltration attack, can be mitigated through the adoption of a blockchain base within a specially built IoT scheme. Additionally, this methodology provides means to mitigate data inaccuracy and inconsistency problems to a large extent. We demonstrated that implementing blockchain technologies can resolve or minimize these requirements at a reasonable cost. Integrating a blockchain-based communication and data storage mechanism to the existing IoT platforms provides secure two-way communication between an IoT platform and the corresponding IoT devices within the network. The blockchain implementation is experimentally realized by programming a unique smart contract to bring efficiency and scalability to the blockchain side. Our contributions can be listed as follows:

1. We introduce a unique IoT deployment and a corresponding communication scheme empowered with an intermediary message queue that increases resilience against availability attacks, an IoT server that enables integration of a blockchain-based data storage, and a private blockchain that provides data integrity, immutability, and non-repudiation for the data to be stored.

2. To achieve the aforementioned functionality, we have implemented a private Ethereum-based (Quorum) blockchain and developed a corresponding smart contract to store sensor data securely and handle event management altogether.

3. Through experiments, we have validated our claims regarding mentioned functionalities and conducted performance analyses, which show that the entire scheme is working at a reasonable speed and is, therefore, scalable.

The rest of this paper is organized as follows: Section II evaluates the existing related studies, Section III provides preliminaries regarding the proposed solution, Section IV explains the solution in detail, Section V introduces our tests and discusses the results, and finally, Section VI concludes the paper and discusses the future works.

2. Related works

There are a few timely related studies in the literature, which introduce methods to let IoT systems benefit from blockchain technology in addressing some security issues. This section provides insight into these studies and briefly introduces their extents.

Ali et al. [3] unveil some critical drawbacks and potential problems of centralized IoT platforms used within various IoT ecosystems, as follows:

- A successful denial of service (DoS) attack can make a centralized platform unusable and prevents its services for the clients. So that, there is a single point of failure that affects the entire network.
- In an IoT ecosystem, a centralized platform stores users' (or devices') sensitive information whose leakage would be a significant risk. That is to say, the owners or producers of the information have very limited control over their data.
- When data are stored in a centralized cloud platform, the system faces potential accountability and traceability problems. Generally, such risks involve sensitive data to be removed or manipulated.

They also stated that these risks regarding the use of centralized systems are increasing as the IoT systems get more and more crowded, induced by the digital transformation trends. This growing number of IoT devices needs the scalability of connected platforms. Yet, building a centralized approach makes it complicated for scalable jobs. In addition, the study shows the following potential solutions of decentralized platforms about the potential problems:

- A decentralized platform can remove the risk of a single-point-of-failure, different from a centralized approach. An additional advantage of the decentralized approach is that it does not need an extra third-party application for controlling to store user data.
- In the blockchain network, checking user identity is already integrated into the system. Also, the blockchain approach verifies the data that are sent by platform clients.
- For accountability and traceability, the blockchain can store event logs and data immutably. This feature provides high benefits to the IoT ecosystem against related potential problems.
- The smart contract is the most useful tool for programmable logic to provide access control, confidentiality, and authentication to increase the security of IoT setups using blockchain.

Another study, by Kathayayani et al. [4], suggested a method to provide a tamper resistance system for IoT devices by using the Hyperledger Fabric blockchain, considering a smart power meter scenario. The authors

have implemented a prototype platform where the IoT nodes produce some data and send them to a web server, whereas the web server posts the data to the connected blockchain. The peers of the Hyperledger Fabric network are made responsible for storing the smart contracts and ledgers. When a smart contract enforces a transaction, the corresponding transaction(s) are kept in the ledgers. During a block process transaction, the peers are responsible for running the predefined smart contracts and sending responses to the client applications, if any. Since all peers keep a local copy of the ledger, every peer communicates with each other to check and sustain the blockchain. Likewise, Hang et al. [5] introduced a novel decentralized platform to handle identity and data security challenges caused by cyber-attacks and analyze the performance issues regarding their platform. The study focused on business logic applications that use smart contracts to conduct event management activities. Furthermore, it provided a solution for the constrained devices to join (and benefit from) the blockchain network efficiently.

Moudoud et al. [6] focused on integrating a blockchain structure to the supply chain management processes to increase the overall security of the systems. The paper suggested a new lightweight consensus algorithm approach to satisfy the speed requirements of IoT deployments. Their consensus algorithm, called "Lightweight Consensus for IoT" is important, since it is a solid proof-of-concept regarding blockchain's applicability within IoT systems. Although we followed similar approaches to make blockchain adoption more efficient in an IoT deployment, we considered different methodologies for the development phases, so that we focus on securing the IoT platform via blockchain, and we aim to provide secure data storage on the platform-side to let constrained devices to work flawlessly over MQ telemetry transport (MQTT) protocol (the MQ part of the name is not an abbreviation per the latest official documentation). Another difference is that we preferred smart contracts to contain rules and conditions to handle event management (for controlling actuator devices), whereas the related study uses the contracts for providing transparency and efficiency and for managing the operations of the stakeholders. Ali et al. [7] aimed to transfer and store IoT (e.g., sensor) data on a blockchain using networked devices equipped with security-aware smart contracts. The study mentions IoT-specific challenges caused by delays and overheads within the blockchain. The authors preferred to use Hyperledger Fabric as the blockchain ledger with the motivation of establishing secure communication between the IoT nodes and within the decentralized ledger. Their motivation is parallel to ours, but their field of application is exclusively different. Throughout the study, the authors elaborated on the smart home applications and potential use cases within that field. Lastly, they presented the results of their comparative analysis, including their proposal and the Quorum blockchain as an alternative.

Huang et al. [8] worked on smart contract security as well. Their motivation was to uncover the potential security gaps within Hyperledger Fabric and Ethereum smart contracts. The study discusses smart contracts as specific software products that can be secured to some extent using predefined software life-cycle models. The paper compares the smart contract mechanisms of Ethereum and Hyperledger Fabric blockchains from a security perspective and considers three different classes of vulnerabilities, particularly for smart contracts, namely issues of specific development languages, features of specific blockchain platforms, and misunderstandings of common practices. They have noted several security problems, including the "self-destruct" bug of the Solidity language. This is caused by a specific bug within the Ethereum platform. More on the languages, they stated that non-determinism-related issues exist in the general-purpose programming language Go used in Hyperledger Fabric. However, this bug does not exist in Solidity and Ethereum thereof. An example of features of specific blockchain platform-based issues is the transaction order dependence (when using Gas-based accounting operations), as revealed. It is, however, not valid for Hyperledger Fabric. Likewise, Ethereum is not vulnerable to "read your

write" attacks. Problems involving misunderstanding of common practices are threatening for both platforms.

Baliga et al. [9] analyzed security features of Quorum, a unique open source blockchain platform. A Quorum node is divided into two sub-modules: the transaction manager and the enclave. The transaction manager is responsible for managing the private transactions with cooperation. On the other hand, the enclave performs encryption and decryption operations related to blockchain transmissions. The Quorum platform supports two different consensus algorithms: Raft [10], and Istanbul Byzantine fault tolerance (IBFT) [11]. The Raft consensus algorithm has been found to have small advantages over IBFT, such as better performance at higher transaction rates (i.e. higher than 1650 transactions per second). Therefore, we preferred to use Raft for our IoT setup, where Quorum is utilized. Xu et al. [12] have elaborated the security of wireless networks that utilize a Raft-based consensus mechanism against jamming attacks. The study suggests a Raft-based wireless blockchain network divided into two different parts, namely, a client and a wireless blockchain consensus network. In Raft-based blockchain consensus networks, any node can be a leader or a follower. The selected leader sends downlink messages to its followers. Whenever a follower receives a related message from its leader, it makes a confirmation operation and sends feedback to the corresponding leader. After these steps, the leader node collects followers' feedback. This collection process aims to provide the absolute majority of the followers' responses. The study also makes analyses on performance and success rates.

Mihelj et al. [13] proposed a system that detects (and responds to) traffic incidents using some crowd-sourced data that are generated by users or preinstalled monitoring devices. They have utilized a "Turing-complete" blockchain platform as a means for data storage. This decentralized platform provides resilience to institutional data manipulation and supposedly provides reliability. In the study, researchers mentioned that the event detection platform was tested through well-defined simulations and data that are generated from users. They implemented an event detection and resource reputation assessment system on a smart contract running over an Ethereum instance. Their system architecture and motivation were inspiring for our study, while our scheme provides modularity and generalizability with a focus on IoT scenarios.

3. Preliminaries

This section introduces our IoT model and the system component preferences.

3.1. IoT model

The following assumptions define our understanding of an IoT ecosystem:

- An IoT ecosystem may consist of (but not limited to) any combination of the following devices: clients (e.g., sensors, actuators, or hybrid devices), special-purpose routers (e.g., brokers), gateways, message queuing interfaces, servers, databases, monitoring devices, etc.
- All nodes in the IoT ecosystem implement the Internet protocol (IP) suite to some extent. The clients (e.g., sensors, actuators) are able to run the MQTT protocol, whereas an IoT server supports transmission control protocol (TCP) and user datagram protocol (UDP), as well as transport layer security (TLS).
- A private blockchain implementation running in real-time is integrated into the IoT network to provide decentralized data storage and event management via smart contracts.
- An attacker who infiltrates to a (centralized) storage unit serving an IoT setup can bypass other authentication mechanisms and record faux data, read, remove, alter, or manipulate all existing data. Additionally,

a (once) legitimate client device (or a user of it) may deny some data entries submitted to the storage earlier. Attacks aiming at other units (e.g., sensors) are out of the scope of this study. Potential risks sourced by smart contract bugs and weaknesses of consensus protocols are neglected as well.

3.2. System components

This section provides preliminary information regarding the essential components of the proposed system.

3.2.1. Virtual environment

Docker, a computer and operating system virtualization solution, is used to implement the IoT nodes and other network elements "virtually". Docker allows building and packing applications with necessary dependencies and running multiple instances virtually independently on a host computer (by sharing some resources among the software). It lets packed applications run in isolated environments called containers, which enables running the instances in parallel, thus providing scalability. This approach decreases the complexity of such studies (docs.docker.com/get-started/overview). Developing a platform, however, is not only a programming challenge but also a question of building an architecture. Docker helps to implement the intended architecture easily. In our solution, all IoT nodes (and other system units) are created as a Docker container, therefore, "dockerized".

3.2.2. Blockchain

Some strict requirements of this project under development limited our choices regarding the adoption of a suitable blockchain ledger among tens of available options existing in the literature and the commercial domain. The first and most important requirement was the native support for smart contracts (in order to program the network). That limited our options to Ethereum, Hyperledger Fabric, Multichain, Lisk, Hyundai digital asset currency (HDAC), and Quorum (although there may be others, we cared for popularity and community support to some extent) [14]. Another design decision was using Raft as the consensus protocol as it is IoT-compatible (justified in the next sub-section). Further, although not mandatory, blockchain technology would better have a specific cryptocurrency or token in circulation, which enables real-time live tests and guarantees continuous operation/development. Also, blockchain structures provide tamper resistance for recordkeeping systems. Attackers cannot alter parts of stored data unless they claim control of 50%+1 of the nodes in the network [15]. Last but not least, transaction processing delays and throughputs are of crucial importance. Hence, Quorum appears as a viable option with necessary features and decent performance [16].

Quorum is a distinct implementation of a private blockchain based on the well-known Ethereum platform (consensys.net/quorum), which was developed by J. P. Morgan Chase and Ethereum enterprise alliance. Ethereum provides a public permissionless blockchain platform to implement decentralized applications in various domains. It also supports smart contracts, essentially decentralized applications running via blockchain, for different purposes in various domains. The idea behind our work is centered on a smart contract that can be used in event management to handle required IoT actuation(s) within the ecosystem. Quorum is preferred as the distributed ledger technology to benefit from an open source Ethereum-based platform. It is a decent choice for storing IoT-related data within a decentralized structure and can easily be favored over a regular private Ethereum network for the following reasons:

- It enables management of network and peer permissions.
- It provides enhanced transaction and contract privacy.

- It supports a voting-based consensus mechanism.
- It offers better performance.

The development notes found in (github.com/51nodes/quorum-local-raft-network) present a Raft-based Quorum blockchain network, implemented using Docker containers. In addition, the related containers run the Raft consensus algorithm without the transaction manager sub-module (e.g., Tessera), which is useful in making encrypted transactions within the blockchain. On the other hand, in our proposed solution, the structure is more secure and robust since accessing the blockchain from outside of the original network is restricted by design. In addition, we considered that making non-private transactions in a blockchain increases the efficiency in terms of network transactions per second (TPS). This does not bring risks regarding privacy because the blockchain network cannot be reached by outsiders in our scenario.

3.2.3. Consensus algorithm

A consensus algorithm is an essential element of blockchain-based systems. It regulates which node shall write what information to the distributed ledger. In our setup, Raft [9, 10] is decided as the consensus algorithm for the integrated Quorum blockchain. There are actually not many options since Quorum only supports Raft and IBFT. Even if they were supported, many other algorithms (e.g., proof-of-work) would be infeasible for constrained IoT systems [17]. As reported in [9], Raft outperforms IBFT for higher transaction rates (which may be the case in live and crowded IoT ecosystems) in terms of latency and throughput.

Additionally, Raft has proven compatibility with Docker containers as well. When Raft is in use, nodes in a blockchain network delegate a leader among them, and all nodes are acknowledged regarding the leader node. Each new transaction is escalated by the leader and distributed to the nodes in a quasi-centralized manner. The escalated transaction is only written to the ledger if at least one more than half of the nodes confirm the transaction. The leader node broadcasts periodic heartbeat messages to sustain its leadership. When it leaves the network, another leader is delegated via election upon the detected timeout.

3.2.4. IoT server

Within an IoT setup, an IoT server is a quasi-centralized computing and storage unit that is usually exempt from some constraints (e.g., battery shortage), limits the operability of smaller IoT nodes, and provides an expanded set of features to the rest of the IoT network it resides in. An IoT server is an optional component that may theoretically act as a data storage, a gateway, an intelligence unit, an automated controller (e.g., smart home systems), or a combination thereof. Our exemplary IoT server does not store data but mediates the communication between the Kafka message queue subsystem and the Ethereum-based ledger. The server is expected to collect necessary messages tagged with a label of the topics of interest from the Kafka message queue, and then it can convert these collected data to the format required for the Ethereum ledger. During this process, the IoT server utilizes the Actor model architecture. The concurrency in the system is provided by design without employing the complexity of threads. In the development, we used the Go-based proto-actor library (https://proto.actor). To translate the Kafka messages to blockchain-compatible format, the official Go implementation of the Ethereum protocol, shortly Go Ethereum (github.com/ethereum/go-ethereum) is used. Whenever a "send" transaction is done, the server checks the smart contract addresses embedded in the message. If there exists a valid smart contract address in the message, the server sends the message to the

smart contract to update the corresponding smart contract values. Later, it interacts with the smart contract to get the desired output.

3.2.5. Message queuing interface

Apache Kafka is an open source distributed event streaming platform that is also capable of serving as a message queuing system. It was implemented by Apache Software Foundation using Scala and Java languages. Like MQTT, it supports a publish/subscribe architecture, where publishing messages on designated topics and subscribing to these topics to get only relevant messages are possible for the attending nodes. In this platform, producers (e.g., sensors) can send their fresh messages upon production, and consumers (e.g., IoT servers) can listen to the messages published from these producers [18] in real-time. The main purpose of setting a Kafka message queue is to provide an intermediary layer between an IoT (i.e., MQTT) broker and the IoT server. Since Apache Kafka has built-in fault tolerance features and provides high performance, it increases the overall resilience of the system against device (e.g., IoT server) failures and availability attacks (kafka.apache.org). For the development phase, we utilized the Confluent Kafka library (www.confluent.io).

3.2.6. Message transmission protocol

MQ telemetry transport (MQTT) is a popular lightweight message transmission protocol that is very beneficial for IoT ecosystems [19]. It also relies on a publish/subscribe architecture (i.e., brokers and clients) and is, therefore, natively compatible with Apache Kafka. All included MQTT clients can send and receive messages within a related channel, named a topic. The MQTT broker(s), on the other hand, is responsible for distributing messages coming from publisher clients to the subscriber clients (and to other devices, if any), depending on the topics they are interested in [20]. Mosca JS, an MQTT brokerage system, is preferred for broker implementation due to its simplicity and ease of use (as well as potential performance benefits). It passes MQTT client messages to the Kafka messaging queue. Mosca JS is a Node.js-based MQTT broker and allows easily scalable development (github.com/moscajs/mosca). For the client-side implementations, Mosquitto (for proof-of-concept) and Apache JMeter (for tests) are used.

4. Proposed scheme

The scheme's components, relations, and interactions are illustrated in Figure 1. As in the figure, all components are implemented in Docker containers. Nevertheless, other architectures may also be used since the tasks and responsibilities of each component are distinctly defined as they are stand-alone modules. An MQTT client (i.e. a sensor or actuator device that implements Mosquitto or JMeter) may transmit sensor or usage (i.e., analytics) data to the broker and receive actuator duties or other commands all over MQTT protocol. An MQTT broker (that implements Mosca JS) is responsible for distributing the messages from the clients to other clients and the Kafka queue depending on the message topics and each devices' topic preferences. It also handles the communication in the opposite direction of the flow. The message queue interface (that implements Kafka) transmits messages that it receives from the broker(s) to the IoT server (but not vice versa) over a produce/consume basis (similar to the publish/subscribe structure used in MQTT). It is also responsible for handling, pre-processing, and temporarily keeping the data when necessary.

An IoT server, apart from its other duties like locally storing data, post-processing data (e.g., for analytics purposes), or interacting with other systems, is responsible for establishing an interface between the rest of the

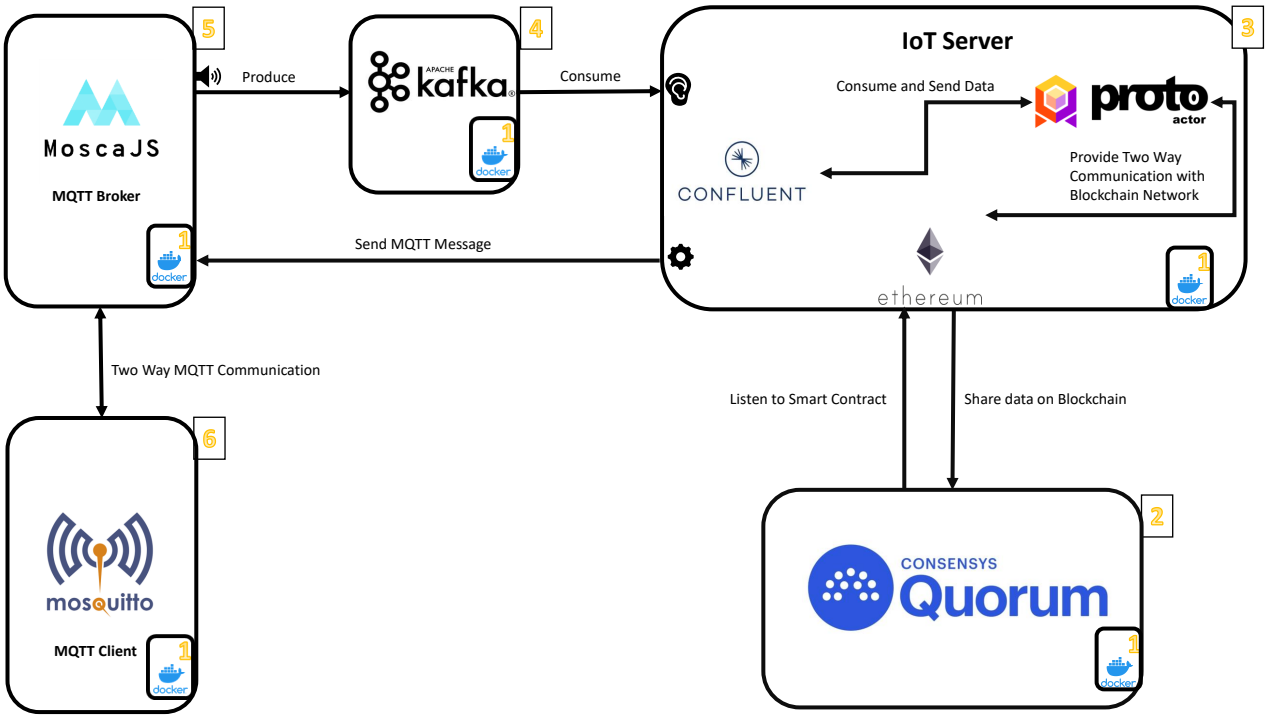


Figure 1. Diagram showing the system components, their interactions, data flows and the overall architecture.

IoT network and the blockchain network. The IoT server is implemented as two main functions using the Actor model (as in Figure 1). The first function (consuming part - Confluent - in the IoT server box) handles consumed messages from Kafka and pops them into the second function, namely blockchain client (Etherum part in the IoT server box), if necessary. The Ethereum part interacts with the blockchain ledger and produces messages to transmit to the broker accordingly. Such interaction includes (*i*) transmitting data (i.e. payload) to be stored in the ledger to Quorum and (*ii*) obtaining smart contract responses from Quorum. Lastly, the server transmits the received smart contract responses directly to the broker (Mosca JS) for client-side distribution. In this phase, the message queue (Kafka) is intentionally bypassed due to simplicity concerns. For the same reason, our implementation does not enforce other duties on the IoT server.

Quorum, as an Ethereum-based decentralized ledger, has two important functions within the proposed scheme. Upon request, it (*i*) stores the IoT data (e.g., sensor data, usage statistics, etc.) consigned by the IoT server, and (*ii*) conducts event management (i.e., decision) activities via pre-programmed smart contracts that it runs. The way it stores data in the ledger (i.e. via decentralized hash chains) allows immutability and non-repudiation, as long as neither of the elements in the network is compromised during production or initial delivery of the data. Whenever some data is written into the ledger, it is no longer possible for a node or an intruder to alter or refuse it owing to the distribution of hashed copies among blockchain peers. By the way, a smart contract intervenes (only) the relevant incoming messages and, thus, makes a decision upon IoT client behaviors that are (*i*) triggering or starting a new process, (*ii*) ceasing an ongoing process, or (*iii*) modifying an ongoing process. This decision is disseminated to the relevant IoT clients over the hierarchical data path shown in Figure 1.

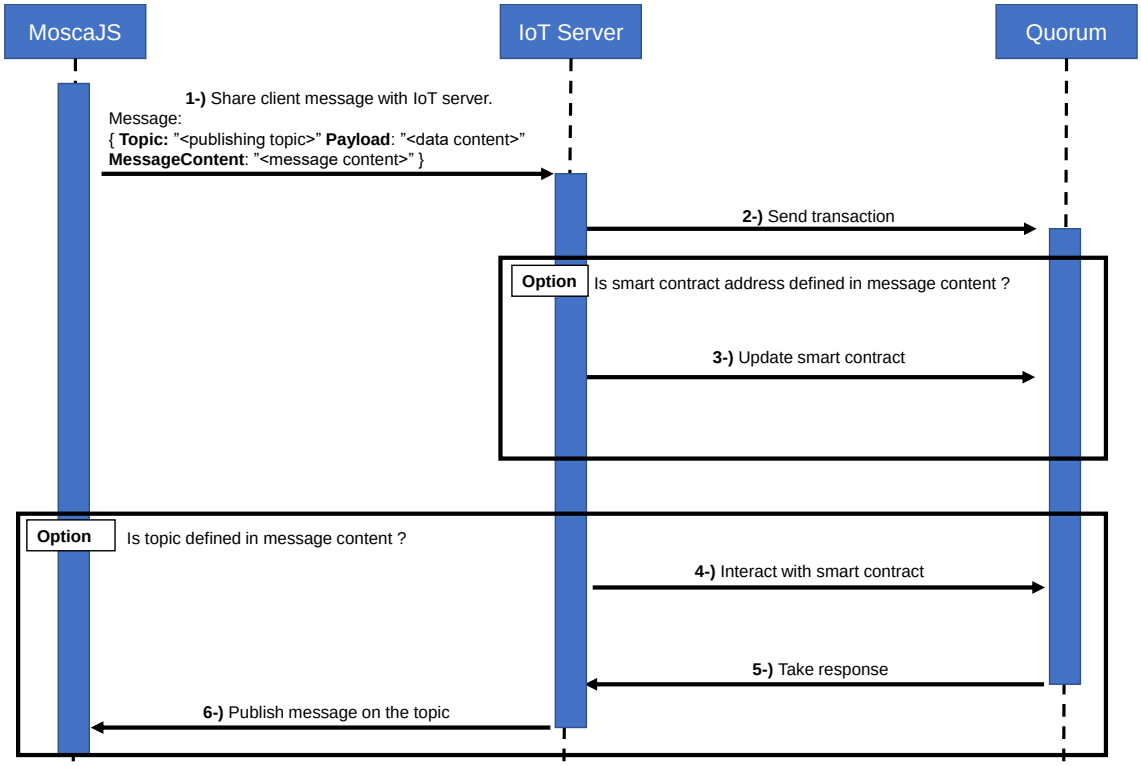


Figure 2. Message sequence chart showing the algorithmic workflow for handling new messages (e.g., sensor data).

As a design decision concerning ease of use, implementation, and integrity, any message that will be sent from the IoT server to the blockchain should be formed as a JavaScript object notation (JSON) object containing relevant data fields (i.e., keys). There are three predefined types of such fields, namely Payload, Topic, and MessageContent, as shown below:

```
{ "Payload":        "<Data to store in the blockchain ledger>"
  "Topic":          "<Topic to get a response from the smart contract>"
  "MessageContent": "<Smart contract values, address, and topic>"      }
```

The field "Payload" includes the data (e.g., sensor data) to be stored in the decentralized ledger. "Topic" includes the topic label of the message. This topic is typically (but not mandatorily) the same as the MQTT topic. The "MessageContent" field contains the information relevant to updating the smart contracts and interacting with them to take action or get any means of outcomes. Optionally, the keys can be set in different JSON messages instead of a single message.

Figure 2 illustrates the messaging procedures followed whenever the IoT server receives a message (as in the top-left box) from the Kafka queue. The figure defines the algorithmic workflow and the data paths via the arrows with sequence numbers. The first path stands for releasing an upcoming message originating from the Kafka queue on an arbitrary topic. The message is received (i.e., consumed) by the IoT server. The Kafka consumer (i.e., IoT server) then sends the transaction records embedded in the new message to the Quorum to store them in the distributed ledger, as shown by the second arrow. After the transmission is completed, the IoT server checks the "MessageContent" field to find if there is a valid smart contract address.

If there is such information in the message, the server updates smart contract values and interacts with the corresponding smart contract of Quorum as shown by the third path in Figure 2. As shown with the fourth path, if the "MessageContent" field has an additional "Publish_Topic" key, the actor layer then interacts with the addressed smart contract in Quorum, gets a decisive order (for event management purposes) as in the fifth path, and sends the resulting order to the relevant IoT clients with "Publish_Topic" eventually over MQTT protocol, as shown in the sixth path. Last but not least, the MQTT clients listen to the topics that the server gives feedback on. The publish feature of the IoT server is developed using the Paho MQTT client library (eclipse.org/paho/index.php?page=clients/golang/index.php). For the clients, Mosquitto is preferred since it has a lightweight and suitable library, especially for listener roles (mosquitto.org).

5. Validation

Given the use case scenario introduced in the following subsection, the proposed scheme is tested thoroughly to check if it provides the intended functionalities and how fast it works.

5.1. Case study

The proposed system architecture may not work flawlessly for particular IoT deployments, such as real-time industrial IoT systems. Because blockchain-based data storage may not allow transacting or processing data at very high speeds (due to the nature of blockchain used). Yet, under the assumptions given earlier, it is possible to generalize the proposed architecture for many IoT scenarios, especially where data push frequency is relatively lower, but the data is critically important (see the achieved data speeds given in this section). Accordingly, smart grid applications (e.g., electricity, water, etc.) and supply chain management are mostly suitable domains for the proposed architecture since the speed of data flow is relatively lower and the need for immutable secure storage is obvious. Edge and fog computing approaches may increase computing capabilities as well as speed [21]. But we focus on unmanipulated data and secure smart contracts. Therefore, the computing approaches are out of the scope of this study.

We have considered a unique scenario with security-critical storage requirements (but requiring moderate speeds), that is, a system detecting the current risk of forest fires in the area of deployment. This use case can easily be implemented with a wireless sensor network system, as stated in [22]. Its potential benefits were surveyed in [23] It can also be integrated into an IoT ecosystem by extending the network to the Internet. It is by no means necessary to have a steady stream of sensor data with a known frequency. By employing our proposal, the risk estimation and decision-making activities can be carried out by a smart contract via the collected real-time sensor data. Son et al. [22] have mentioned a risk detection measure, called the danger index (DI), originally published by the Korea Forest Service. It is found using the below formulae:

$$DI = 6.87 + (0.64 \times P) + (0.15 \times EH) + (1774.94 \div CS) \tag{1}$$

Where CS is the current solar radiation, P is the precipitation ratio, and EH is the effective humidity that is calculated as below:

$$EH = (1 - r) \times (H_0 + (r \times H_1) + (r^2 \times H_2) + (r^3 \times H_3) + (r^4 \times H_4)) \tag{2}$$

Where H_{0-4} are five different humidity measurements and r is a coefficient of fixed value (0.7). All constants are predefined by the original publisher and are of no significance to the system under development.

Yet, the calculated danger index (DI) is then classified using the following scale: $0 \leq DI < 60$ indicates a *Low Risk* state, $60 \leq DI < 80$ indicates a *High Risk* state, and $80 \leq DI \leq 100$ indicates an *Extreme Risk* state. The terms and abbreviations used in the validation section are collected in Table 1.

Table 1. Terms & Definitions

Abbreviation	Definition	Remarks
CS	Current solar radiation	-
DI	Danger index	-
EH	Effective humidity	-
H_{0-4}	Humiditiy values	Assumed 5 distinct values
P	Precipitation ratio	-
r	Coefficient	Set to 0.7 in example
AM	Action message	IoT message type
RM	Registry message	IoT message type

5.2. Implementation

In the simulation environment, we have implemented the features of the case study on our components. The clients provide CS, P, and H_{0-4} values to the broker and, eventually, to the IoT server. The values are preset for simulation purposes since their actual values do not matter. Calculations of EH in Equation 2 and DI in Equation 1 are done by the IoT server using the smart contract. The smart contract is implemented on the ledger, using Solidity language. The code of our implementation is given below. We used the OpenZeppelin math library (github.com/OpenZeppelin/openzeppelin-contracts) for Solidity to conduct arithmetic operations.

```
pragma solidity ^0.7.5;
import "openzeppelin-contracts/contracts/math/SafeMath.sol";

contract Inbox {
  uint256 DI;
  uint256 P;
  uint256 CS;
  uint256 H0HR;
  uint256 EH;
  uint256 r = uint256(70);

  function Set(uint256 _CS, uint256 _P, uint256 _H0, uint256 _H1, uint256 _H2, uint256 _H3, uint256
      _H4) view public returns(uint256) {
    P = SafeMath.div(SafeMath.mul(uint256(64), _P), uint256(10**2));
    CS = SafeMath.div(uint256(17749400), _CS);
    H0HR = SafeMath.add(SafeMath.add(SafeMath.add(SafeMath.add(_H0, SafeMath.div(SafeMath.mul(r,
        _H1), uint256(10**2))), SafeMath.div(SafeMath.mul(r**2, _H2), uint256(10**4))),
        SafeMath.div(SafeMath.mul(r**3, _H3), uint256(10**6))), SafeMath.div(SafeMath.mul(r**4,
        _H4), uint256(10**8)));
    EH = SafeMath.div(SafeMath.mul(uint256(15), SafeMath.mul(uint256(30), H0HR)), uint256(10**4));
    DI = SafeMath.div(SafeMath.add(SafeMath.add(SafeMath.add(uint256(687), P), EH), CS),
        uint256(10**2));
    return DI;
  }
  function Get() view public returns(uint256) {
    return DI;
  }
```

}

As a remark, Solidity version 0.7.5 causes some implementation challenges, since decimal multiplications are problematic. Our approach was multiplying the decimal numbers with powers of 10 to eliminate fractions and then dividing the results by the same factor to normalize per to the range of the danger index.

5.3. Testbed & configuration

For testing purposes, the MQTT clients are simulated via Apache JMeter (jmeter.apache.org). It allows the making of load tests and is used as a simulation tool [24]. More, we used the MQTT plugin to send MQTT messages from Apache JMeter instances. The host hardware has the following properties: Windows 10 (64-bit) operating system, 32 GB of RAM, and an Intel i7 processor.

The time measurements are repeated with two different message types, namely a registry message (RM) and an action message (AM). A registry message merely contains some information (e.g., sensor data) to be recorded on the ledger, whereas an action message includes an additional smart contract configuration (e.g., address, parameters, etc.) as well as some information to be recorded. The action message enforces the smart contract (pointed in the address) to run, make some decisions, and escalate an action. This triggering mechanism comes at an expected cost of larger messages and higher latency. In either case, the messages are sent over MQTT version 3.1 with quality of service (QoS) level 2 and with merely one topic. The process does not require retaining the messages. During the test runs, we calculated the IoT server's performance to assess the efficiency, usability, and scalability of the proposed scheme. Therefore, each MQTT message has a timestamp of its creation time. Additionally, when the blockchain operations are done, a new timestamp has been generated. The difference between those two indicates the completion time of the required actions implied by each MQTT message. This time measurement applies to all messages sent with different rates of transactions per second. Different TPS rates are achieved by scheduling 1000 concurrent threads accordingly.

5.4. Test of registry messages

The test is repeated three times, each at different rates, consecutively at 5 TPS, 10 TPS, and 12.5 TPS. Each case (i.e. speed) has been run with 1000 concurrent threads (representing the clients). Regarding each run, the individual measurements and the mean averages are depicted in Figures 3, 4, and 5 in the given order. The results are also summarized in Table 2. All the graphs reflect the confidence interval that was preset at 95% and the coefficient of determination (r-squared), as hinted in [25].

Table 2. Summary of test results

Test	Speed (TPS)	Time for 1000 Threads (s)	Avg. Latency (ms)
RM	5	200	22.727
RM	10	100	17.866
RM	12.5	80	19.597
AM	4*	250	53.089

**4 in clients, 8 in the blockchain.*

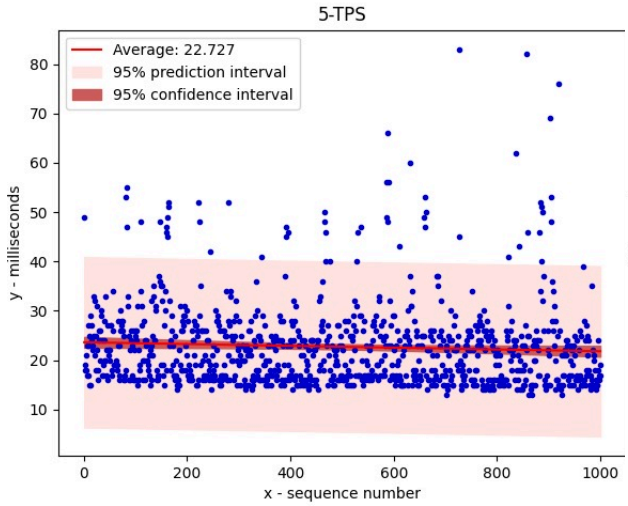


Figure 3. Registry messages (RM) @ 5 TPS.

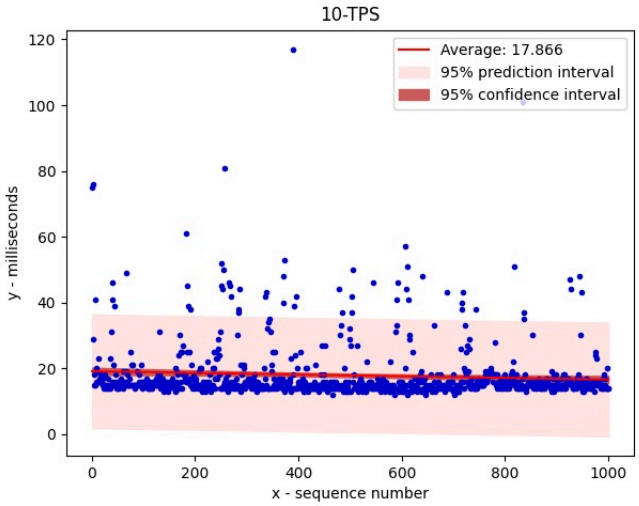


Figure 4. Registry messages (RM) @ 10 TPS.

5.5. Test of action messages

The test is only conducted once at a fixed speed of 4 TPS. Nevertheless, this is the speed set at the client-side. However, in our scheme, triggering a smart contract enforces the smart contract to respond to the clients. Hence, the speed of the blockchain is 8 TPS due to this request-and-response style of communication. During the test, the smart contract is triggered only once. On the other hand, the size of the messages sent within the two tests are different; in RM, the size of a packet is 108 bytes, whereas it becomes 159 bytes in AM. As in RM, 1000 MQTT threads were run concurrently, and an average is found out. The individual results are depicted in Figure 6.

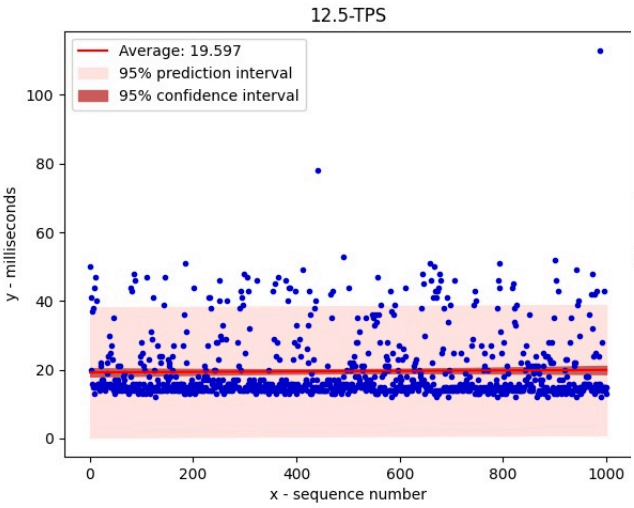


Figure 5. Registry messages (RM) @ 12.5 TPS.

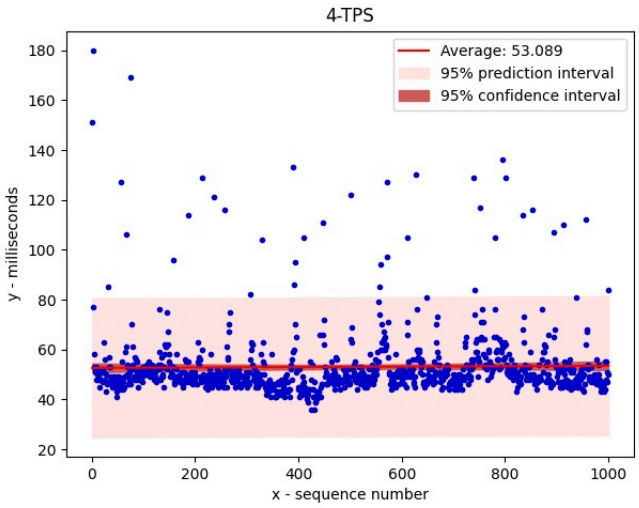


Figure 6. Action messages (AM) @ 4 TPS.

A proof-of-concept of the proposed scheme is, thus, developed following a specific use case. It was made to run flawlessly and validated thereafter. As inferred from the results, the registry messages can be processed

significantly (i.e., nearly three times) faster than the action messages (which also means a proportionally lower average latency) because the action messages include two subsequent operations on the blockchain (i.e. Quorum). Furthermore, the size of action messages is (nearly 1.5 times) bigger than that of registry messages. Therefore, increasing the size of the transmitted messages and the complexity of the smart contract decreases the system performance dramatically. As a side note, although the running times are purely acceptable for an IoT server, some performance improvements may be necessary for the blockchain implementation in order to make the scheme more scalable. Ultimately, the results indicate that if the collected data is sensitive and the data collection frequency is low, then the data storage and event management activities can easily be carried out via the proposed scheme.

5.6. QoS assesment

Quality of service (QoS) is a design and operation paradigm in information and communication technologies that involves defining and tracking a set of (most likely quantitative) measures and characteristics of a serving system. These measures help quantify how well the system-of-interest works and what actions should be taken to make it serve better, if any.

The proposed system in this paper is a Docker-powered microservices application that can easily be deployed online, preferably at a cloud server, to be offered as a public or private service. The client IoT devices shall use the cloud based MQTT broker for sending their data to the blockchain application. Therefore, the entire system can be seen as a software as a service (SaaS), where common QoS metrics used for SaaS are also applicable. Ezenwoke et al. [26] described some quantitative and qualitative attributes about QoS concerning cloud services. These attributes include response time, cost, availability, usability, security management, and flexibility. Considering our research, the following statements can be reached regarding the abovesaid QoS attributes of our proposal:

- Response time: The delay between a request (e.g., a data push) and the actual time it completes is the response time. Although it is application-dependent, increasing the capacity of resources (e.g., processing power, bandwidth, memory, etc.) of the cloud platform will decrease the response time [27]. The average response time of the Ethereum network, which is around 12 to 14 seconds, is a justifiable reference point for QoS determination (ethereum.org/en/developers/docs/blocks). We can guarantee even lower response times for the proposed setup, thanks to the consensus protocol chosen, namely Raft, which works faster. Further, as discussed in Section 3.2 that Raft works even faster without the transaction manager submodule. These performance improvements are carefully considered for the proposed setup.

- Cost: The cost in a cloud based service is a function of the allocated resources. There is a tradeoff between the cost and the resources. Requirements and feasibility change per application. More discussion can be found later in Section 5.7.

- Availability: The system's uptime in which it is active and responsive defines the availability. IoT devices generate data on a continuous basis. Therefore, the telco-grade availability rate of %99.99 [28] can be considered to represent the high availability requirements. Throughout our tests, no outage and no packet loss have been recorded.

The proposed system may be implemented to provide different QoS standards, including the strictest industry considerations.

5.7. Discussion of complexity

A brief complexity analysis concerning our study's exemplary use case (implemented within the smart contract) is given. We utilized the Big-O notation to find an execution time for the related algorithm irrespective of processor or programming language selection. The smart contract implementation algorithm executes two functions, namely, the effective humidity (EH) and the danger index (DI).

Within EH, the atomic base operation is the multiplication (as the addition is clearly dominated by the multiplication) and the input is the number of data sources (e.g., sensor nodes). As the number of such sources increases, multiplications increase quadratically since they exist among H values and the power terms of r. Accordingly, the correlation between the node count (n) and the multiplication count ($f(n)$) can be stated as $f(n) = 0.5n^2 - 0.5n + 1$. So that, $O(f(n))$ leads to $O(n^2)$. On the other hand, the computational complexity of DI is trivial and is determined by the EH function. Nevertheless, this analysis is only valid for our specific use case. A broader analysis of the proposed architecture can be obtained by evaluating the message complexity.

Again, the message complexity is strictly dependent on the number of nodes involved in the IoT deployment (n). Moreover, it also depends on the number of processing nodes within the blockchain network (m). During the log replication process, the Raft consensus algorithm creates a single two-way path between the leader node and each of the clients, which results in linearity in terms of message complexity. The (occasional) leader election phases, however, requires quadratic messaging between all the node pairs. But the dominance of this phase depends on the predefined leader election periods and can be negligible for longer periods. When sufficiently longer periods are considered, the overall message complexity of the system becomes $O(nm)$; otherwise, $O(nm^2)$ is obtained.

6. Conclusion

This study first discusses data integrity, immutability, and non-repudiation issues, as well as their adverse effects on event management processes within IoT ecosystems organized in a centralized way. Briefly, these issues arise with a successful infiltration attempt to the servers and database(s) of the system. An attacker is then able to alter (or even wipe out) the collected data and may interfere with event-response mechanisms resulting in malfunctioning of the entire system. Integrating a blockchain mechanism (i.e., a decentralized ledger) as a data storage and processing unit is found to be helpful in addressing these challenges.

Earlier works involving blockchain integration to IoT systems in the literature, to the best of our knowledge, either lack implementational details, or omit performance analyses, or are not generalizable (i.e., special-purpose), or simply do not perform well (i.e., not scalable, not efficient, etc.). On the other hand, we proposed a well-defined, implemented, and tested IoT architecture that combines MQTT-based IoT networks and an Ethereum-based blockchain in a modular way to mitigate the mentioned immutability and event management issues in a scalable and efficient manner with the help of a custom smart contract. The proposed scheme involves IoT clients (i.e., sensor and actuator nodes) that communicate through MQTT via Mosquitto, broker nodes that implement Mosca-JS, a message queue node that runs Apache Kafka, an IoT server that drives the proto-actor model, and an Ethereum-based blockchain node that inherits Quorum ledger with smart contract support as demonstrated in Figures 1 and 2. So, messages containing sensor data are stored in the blockchain network, and the smart contract runs to assess event management decisions (e.g., actuator triggers) when necessary.

A successful implementation was made via Docker containers on a PC. Without loss of generality, a forest fire monitoring system was considered as the use case for validation purposes. Mission-critical IoT data

can reliably be stored and processed using the proposed scheme, while for trivial data, there is still no need to undertake the operating costs of a blockchain ledger. In addition to the importance of the collected data, the frequency of it is also important. The scheme may be tackled by bottlenecks when data transmission frequency is very high (e.g., several hundreds of TPS, although this is mostly application-dependent). Quorum with Raft consensus protocol works well under the test case conditions so that in a very large network consisting of 1000 clients, sensor data can be recorded to the ledger by a rate of 12.5 TPS, and event trigger messages can be disseminated to the actuators by a rate of 4 TPS (including requests and 8 TPS considering one-way rate). A smaller network would perform even better as the scheme has polynomial time and message complexity.

There are hundreds of alternative open source blockchains and counting. Thus, further research may reveal better-performing ledgers and/or consensus algorithms for this purpose. Different IoT protocols such as constrained application protocol (CoAP) or lightweight machine-to-machine (LWM2M) may be better for various scenarios. Future works shall also include the integration of sidechains (as explained in [29]) to extend the network with additional layers of secure data storage. Throughout this study, we have discovered two possible barriers to widespread adoption of the proposed method: The first is the considerably low number of allowed transactions per second within most blockchain systems. The issue is still an open problem for crowded or highly active networks. It occurs when the IoT system transmits data much faster than its blockchain counterpart can record to the ledger. But this may not be the case as we already see that our TPS requirements can be met by an Ethereum network. The second is the lack of unique smart contract libraries for the IoT. We will be working on new IoT-oriented smart contract libraries to increase the applicability of blockchain into the IoT domain.

References

[1] Rogojanu T, Ghita M, Stanciu V, Ciobanu R, Marin R et al. Netiot: A versatile iot platform integrating sensors and applications. In: 2018 Global Internet of Things Summit (GIoTS); Bilbao, Spain; 2018. pp. 1–6.

[2] Escolar AM, Alcaraz-Calero JM, Salva-Garcia P, Bernabe JB, Wang Q. Adaptive network slicing in multi-tenant 5g iot networks. IEEE Access 2021; 9: 14048–14069. doi: 10.1109/ACCESS.2021.3051940

[3] Ali MS, Vecchio M, Pincheira M, Dolui K, Antonelli F et al. Applications of blockchains in the internet of things: A comprehensive survey. IEEE Communications Surveys and Tutorials 2019; 21 (2): 1676–1717. doi: 10.1109/COMST.2018.2886932

[4] Kathayayani N, Murthy JK, Naik VN. Hyperledger fabric blockchain for data security in iot devices. International Journal of Recent Technology and Engineering 2020; 9 (1): 2571–2577. doi: 10.35940/ijrte.A3040.059120

[5] Hang L, Kim DH. Design and implementation of an integrated IoT blockchain platform for sensing data integrity. Sensors 2019; 19 (10): 2228. doi: 10.3390/s19102228

[6] Moudoud H, Cherkaoui S, Khoukhi L. An iot blockchain architecture using oracles and smart contracts: the use-case of a food supply chain. In: 2019 IEEE 30th Annual International Symposium on Personal, Indoor and Mobile Radio Communications (PIMRC); Istanbul, Turkey; 2019. pp. 1–6.

[7] Ali J, Ali T, Musa S, Zahrani A. Towards secure iot communication with smart contracts in a blockchain infrastructure. International Journal of Advanced Computer Science and Applications 2018; 9 (10): 578-585.

[8] Huang Y, Bian Y, Li R, Zhao JL, Shi P. Smart contract security: a software lifecycle perspective. IEEE Access 2019; 7: 150184–150202. doi: 10.1109/ACCESS.2019.2946988

[9] Baliga A, Subhod I, Kamat P, Chatterjee S. Performance evaluation of the quorum blockchain platform. arXiv:2018; 1809.03421.

[10] Ongaro D, Ousterhout J. In search of an understandable consensus algorithm. In: 2014 USENIX Annual Technical Conference, USENIX ATC'14; Philadelphia, PA, USA; 2014. pp. 305–319.

[11] Moniz H. The Istanbul bft consensus algorithm. arXiv preprint 2020; 2002.03613.

[12] Xu H, Zhang L, Liu Y, Cao B. Raft based wireless blockchain networks in the presence of malicious jamming. IEEE Wireless Communications Letters 2020; 9 (6): 817–821. doi: 10.1109/LWC.2020.2971469

[13] Mihelj J, Zhang Y, Kos A, Sedlar U. Crowdsourced traffic event detection and source reputation assessment using smart contracts. Sensors 2019; 19 (15): 3267. doi: 10.3390/s19153267

[14] Reyna A, Martín C, Chen J, Soler E, Díaz M. On blockchain and its integration with iot. challenges and opportunities. Future Generation Computer Systems 2018; 88: 173–190. doi: 10.1016/j.future.2018.05.046

[15] She W, Liu Q, Tian Z, Chen JS, Wang B et al. Blockchain trust model for malicious node detection in wireless sensor networks. IEEE Access 2019; 7: 38947–38956. doi: 10.1109/ACCESS.2019.2902811

[16] Dabbagh M, Choo KKR, Beheshti A, Tahir M, Safa NS. A survey of empirical performance evaluation of permissioned blockchain platforms: challenges and opportunities. Computers & Security 2021; 100: 102078. doi: 10.1016/j.cose.2020.102078

[17] Yao H, Mai T, Wang J, Ji Z, Jiang C et al. Resource trading in blockchain-based industrial internet of things. IEEE Transactions on Industrial Informatics 2019; 15 (6): 3602–3609. doi: 10.1109/TII.2019.2902563

[18] Shaheen J. Apache kafka: real time implementation with kafka architecture review. International Journal of Advanced Science and Technology 2017; 109: 35–42.

[19] Collina M, Corazza GE, Vanelli-Coralli A. Introducing the qest broker: Scaling the iot by bridging mqtt and rest. In: 2012 IEEE 23rd International
Symposium on Personal, Indoor and Mobile Radio Communications (PIMRC); Sydney, NSW, Australia; 2012. pp. 36–41.

[20] Chien HY, Chen YJ, Qiu GH, Liao JF, Hung RW et al. A MQTT-API- compatible IoT security-enhanced platform. International Journal of Sensor Networks 2020; 32 (1): 54-68.

[21] Elazhary H. Internet of things (iot), mobile cloud, cloudlet, mobile iot, iot cloud, fog, mobile edge, and edge emerging computing paradigms: disambiguation and research directions. Journal of Network and Computer Applications 2019; 128: 105–140. doi: 10.1016/j.jnca.2018.10.021

[22] Son B, Her Y, Jung-Gyu Kim JG. A design and implementation of forest- fires surveillance system based on wireless sensor networks for South Korea mountains. International Journal of Computer Science and Network Security 2006; 6 (9B): 124–130.

[23] Datta S, Das AK, Kumar A, Khushboo, Sinha D. Authentication and privacy preservation in iot based forest fire detection by using blockchain – a review. In: 4th International Conference on Internet of Things and Connected Technologies; Jaipur, India; 2020. pp. 133–143.

[24] Paz S, Bernardino J. Web platform assessment tools: An experimental evaluation. In: Web Information Systems and Technologies; Porto, Portugal; 2017. pp. 45–63.

[25] Seneviratne C, Wijesekara PADSN, Leung H. Performance analysis of distributed estimation for data fusion using a statistical approach in smart grid noisy wireless sensor networks. Sensors 2020; 20 (2): 567. doi: 10.3390/s20020567

[26] Ezenwoke A, Daramola O, Adigun M. QoS-based ranking and selection of SaaS applications using heterogeneous similarity metrics. Journal of Cloud Computing 2018; 7: 15. doi: 10.1186/s13677-018-0117-4

[27] Schäffer M, Di Angelo M, Salzer G. Performance and scalability of private ethereum blockchains. In: Business Process Management: Blockchain and Central and Eastern Europe Forum; Vienna, Austria; 2019. pp. 103-118.

[28] Benefield R. Agile deployment: Lean service management and deployment strategies for the saas enterprise. In: 2009 42nd Hawaii International Conference on System Sciences; Waikoloa, HI, USA; 2009. pp. 1–5.

[29] Musungate BN, Candan B, Çabuk UC, Dalkılıç G. Sidechains: Highlights and challenges. In: Innovations in Intelligent Systems and Applications Conference (ASYU); Izmir, Turkey; 2019. pp. 1–5.