

Alma Mater Studiorum Università di Bologna
Archivio istituzionale della ricerca

Blockchain and Web of Things for Structural Health Monitoring Applications: A Proof of Concept

This is the final peer-reviewed author's accepted manuscript (postprint) of the following publication:

Published Version:

Gigli, L., Sciallo, L., Montori, F., Marzani, A., Di Felice, M. (2022). Blockchain and Web of Things for Structural Health Monitoring Applications: A Proof of Concept [10.1109/CCNC49033.2022.9700679].

Availability:

This version is available at: <https://hdl.handle.net/11585/890019> since: 2022-07-05

Published:

DOI: <http://doi.org/10.1109/CCNC49033.2022.9700679>

Terms of use:

Some rights reserved. The terms and conditions for the reuse of this version of the manuscript are specified in the publishing policy. For all terms of use and more information see the publisher's website.

This item was downloaded from IRIS Università di Bologna (<https://cris.unibo.it/>).
When citing, please refer to the published version.

(Article begins on next page)

This is the final peer-reviewed accepted manuscript of:

L. Gigli, L. Sciullo, F. Montori, A. Marzani and M. Di Felice, "Blockchain and Web of Things for Structural Health Monitoring Applications: A Proof of Concept," 2022 IEEE 19th Annual Consumer Communications & Networking Conference (CCNC), Las Vegas, NV, USA, 2022, pp. 699-702

The final published version is available online at
<https://dx.doi.org/10.1109/CCNC49033.2022.9700679>

Rights / License:

The terms and conditions for the reuse of this version of the manuscript are specified in the publishing policy. For all terms of use and more information see the publisher's website.

This item was downloaded from IRIS Università di Bologna (<https://cris.unibo.it/>)

When citing, please refer to the published version.

Blockchain and Web of Things for Structural Health Monitoring Applications: A Proof of Concept

Lorenzo Gigli*, Luca Sciallo*, Federico Montori*[‡], Alessandro Marzani^{†‡}, Marco Di Felice*[‡],

* Department of Computer Science and Engineering, University of Bologna, Italy

[†] Department of Civil, Chemical, Environmental, and Materials Engineering, University of Bologna, Italy

[‡] Advanced Research Center on Electronic Systems “Ercolo De Castro”, University of Bologna, Italy

Emails: {lorenzo.gigli, luca.sciallo, federico.montori, alessandro.marzani, marco.difelice3}@unibo.it

Abstract—Interoperable and secure data management techniques are fundamental for most of large-scale Structural Health Monitoring (SHM) systems. Indeed, given the relevance of SHM critical measurements, data integrity must be protected against tampering or falsifications. In this paper, we propose a four-layer SHM architecture that allows to build an effective data pipeline from sensors to consumer applications, passing through the cloud. The architecture is built on top of the MODRON platform and exploits the recent advances of the W3C Web of Things (WoT) standard for interoperability. We then discuss how third-party services can take benefit of the W3C WoT architecture to retrieve the SHM critical data and to publish them on the Ethereum Blockchain through an SHM-specific Smart Contract, for data protection and traceability purposes. We test the effectiveness of the Smart Contract implementation in terms of latency and costs under simulated workloads.

Index Terms—Internet of Things, W3C Web of Things, Structural Health Monitoring (SHM), Blockchain, Smart contracts

I. INTRODUCTION

Structural Health Monitoring (SHM) denotes a broad interdisciplinary research area focused on methods and technologies for the continuous assessment of the conditions of civil and industrial buildings and infrastructures. Current deployments of SHM systems can take advantage of the Internet of Things (IoT) solutions for enhanced scalability and cost effectiveness [1]. At the same time, while increasing attention has been dedicated to the sensing operations, some data-related and software-related issues are still barely explored and may affect the deployments of SHM systems on the large scale. In this paper, we address a major requirement of SHM systems, namely *data protection*, which takes into account the relevance of information gathered by SHM systems for the safety of workers and end-users. Hence, data about critical events must be preserved from possible tampering, yet made available to the stakeholders to enable further diagnoses.

In this paper, we make use of a generic platform for data acquisition and management in heterogeneous SHM scenarios, called MODRON [2], developed within the INAIL MAC4PRO project. MODRON natively addresses interoperability across heterogeneous IoT platforms and devices, by defining uniform interfaces on how the IoT components should interact with each other through the W3C Web of Things (WoT) standard

[3]. The standard introduces the concept of Web Thing (WT) as a physical (*e.g.* an IoT device) or virtual entity (*e.g.* a micro-service) whose capabilities are described through a standardized collection of metadata called the Thing Description (TD). We consider a scenario where SHM stakeholders (*e.g.* public authorities) can take advantage of the W3C WoT-oriented architecture to query the WTs and store their critical data over a Blockchain network. The latter has emerged as a breakthrough technology in many IoT use-cases for distributed data replication and securing, *e.g.* for traceability of IoT operations. Through the definition of a new Smart Contract for the SHM domain, we enable push operations of critical SHM data (*e.g.* alerts caused by anomalies or over/under threshold values) onto the Ethereum Blockchain, hence making them not alterable by the SHM data providers.

More in detail, to protect the integrity of SHM measurements, we describe the integration of our W3C WoT architecture with a controller service, built on top of MODRON platform, which retrieves critical SHM data and publishes them on the Ethereum blockchain. To this aim, we detail the implementation of a Smart Contract for SHM, including basic functionalities of measurements reading/writing. We then validate the operations of MODRON in real-world SHM use-cases from the MAC4PRO project, related to the real-time, continuous monitoring of civil structures. In addition, we evaluate the effectiveness of the implementation of the proposed Smart Contract in terms of latency and pricing under simulated workloads. The performance evaluation demonstrates that the proposed implementation of the Smart Contract is able to scale under increasing workloads, and to greatly reduce latency/costs metrics when compared to naive solutions.

II. PROPOSED WoT-BASED ARCHITECTURE FOR SHM

We consider the four-layer logical architecture of a generic SHM system, depicted in Figure 1. The *Sensing* Layer is in charge of measuring the physical quantities useful to assess the structural health of the target entity, hence including the wired/wireless sensor networks installed on the monitored structure. The *Edge* Layer is a software stratum providing a uniform representation of the heterogeneous sensing devices, in order to ease their remote management and data retrieval, while hiding most of the details related to their implementation. In our case, the representation is based on

the W3C WoT standard [3] as detailed in [2]. The software stratum is implemented by one or more Edge Processing Units (EPUs) located in the proximity of the monitored structure. The *Cloud Layer* addresses SHM data storage, aggregation, processing, visualization, and export via dedicated APIs. Due to the computational requirements, we assume that this layer is hosted on a remote private/public cloud, connected to the EPUs via an Internet connection. Finally, the *Service Layer* is constituted by third-party services/applications, external to the SHM platform but able to access its data in order to offer additional visualization/processing functionalities to the stakeholders. The main contribution of this paper involves the Service layer, where a third-party service, the SHM Controller, accesses sensor data from the Cloud and stores critical values onto a Smart Contract deployed on an Ethereum blockchain (detailed in the next section). The Edge and the Cloud layers are implemented within the MODRON platform [2], while some details of the Sensing layer technologies used in our test-beds are provided in [4].

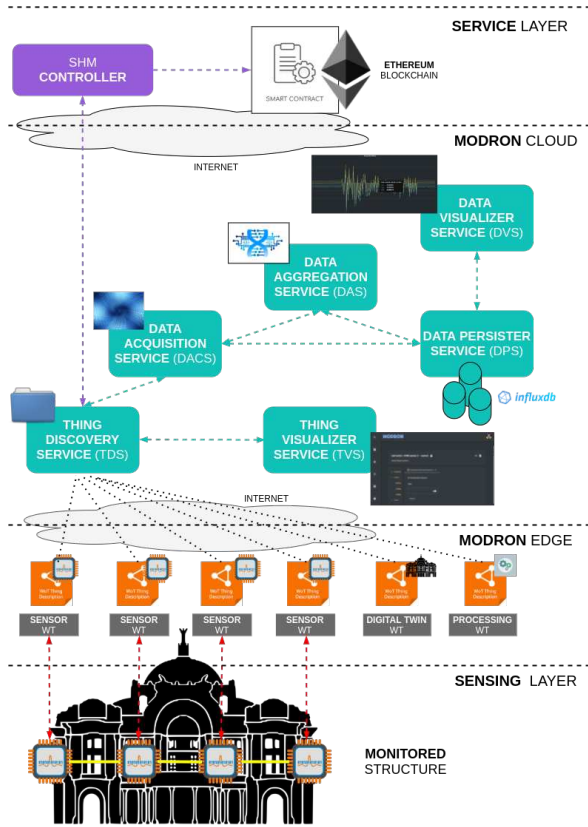


Fig. 1. The MODRON architecture and its integration with the Service Layer.

III. BLOCKCHAIN INTEGRATION

The SHM Controller is a service we developed that enables the integration with the Blockchain. It is external to the MODRON platform and designed to be used by a third-party, since it is designed to reduce the risk of data alteration in the pipeline from sensors to cloud; at the same time, it

exploits the W3C WoT-oriented architecture of MODRON and, more specifically, the presence of WTs to retrieve data from heterogeneous sensors. As shown in Figure 1, the SHM Controller retrieves the list of active Sensor WTs; then, it consumes the TDs and receives the measurements directly from the WTs. Due to costs and performance issues, it may not be realistic to store all the SHM data on the Blockchain, therefore the SHM Controller applies a filter over them, *i.e.* only the measurements outside a safety range are passed to the Smart Contract. In the following, we detail the API of the Smart Contract and its internal implementation.

A. Smart Contract: Overview and Data Structures

The sensitive data coming from the SHM Controller is stored by a Smart Contract deployed on the Ethereum blockchain called “SHMController Smart Contract” (SSC). In the following, we outline the structure and the main features of the SSC, in particular: (i) The data structures used as storage, and (ii) the methods for uploading new data points and for querying the knowledge base on top of certain parameters. It is necessary to highlight how producing an efficient Smart Contract requires to minimize the number of executed operations, as, the more complex and long a procedure is, the more resources will be consumed. The metric reflecting the amount of computational effort to run Smart Contract operations, and related fee required, is also called *gas* in Ethereum. Hence, minimizing gas consumption is the must-have property of a Smart Contract implementation. On a side note, this might not be directly true for “pure” and “view” methods (*i.e.* those not directly altering the data structures), which do not cause gas consumption per se, however, it becomes so in side cases, for instance when the Smart Contract gets queried by a non-view/pure method of another Smart Contract. Since the SSC stores data points that ideally identify critical or abnormal sensor measurements over a monitored structure, we would not expect continuous bursts of data, rather, few occasional measurements that denote important events. The listing below shows the structure of a single on-chain measurement:

```
struct Measurement {
    uint64 timestamp;
    string structureId;
    string sensorId;
    string sensorType;
    string data;
}
```

Here, the timestamp is a Unix time – exact to the ns – and the data field encodes the data value in a string. The latter happens because data might not always consist of only one number (*e.g.* accelerometers have three values, one for each axis) and values themselves are not processed on-chain, therefore expensive encoding and decoding operations are left to the clients. The actual data structures where data is saved are the following three “mappings” (key-value dictionaries):

```
mapping(uint256=>Measurement) private measurements;
mapping(uint256=>uint256[]) private dateMap;
mapping(string=>uint256[]) private structureMap;
```

The first data structure is a list that maps each measurement with a progressively generated identifier. The choice for a progressive number, rather than a randomly generated one (e.g. UUID), is important because it speeds up filtering operations. Every time a new measurement is loaded into the SSC, it will be pushed back in the measurements list and will be assigned the highest identifier, therefore, the list is chronologically sorted by construction. The second mapping `dateMap` maps a day with a list of measurement identifiers. Thus, every time a new measurement is loaded into the SSC, its identifier will also be appended at the end of the list for the current day, ensuring that each sublist is sorted by construction. The third mapping `structureMap` maps a structure id with a list of measurement identifiers. Such as above, each sublist is sorted by construction. For efficiency purposes, the maximum number of days and structures handled by a single contract is fixed (any excess should be handled by additional contracts).

B. Smart Contract: Methods

The SSC can be invoked by clients via the methods prototyped below. The design of such methods focuses on their computational efficiency, in order to prevent wastage of gas.

```
function insert(uint256 date, Measurement calldata
    measurement) external onlyOwner
function getByDates(uint256[] calldata dates)
    external view returns (Measurement[] memory)
function getByStructures(string[] calldata
    structureIds) external view returns (Measurement
    [] memory)
function getByDatesAndStructures(uint256[] calldata
    dates, string[] calldata structureIds) external
    view returns (Measurement[] %memory)
```

The `insert` method is used by a client to load a new measurement onto the SSC. This operation only costs $\mathcal{O}(1)$ because the new measurement is appended at the end of the measurement list, as well as at the end of the respective two lists in the mappings. The `getByDates` is a view method invoked to obtain all the measurements that were registered within the days given in input. Considering n as the number of total measurements stored in the SSC, then this operation is $\mathcal{O}(n)$ in the worst case, as it needs to retrieve the measurement identifiers for the requested days from the `dateMap` and extract the information of each of them from the measurement list. The `getByStructures` method has a similar behavior to the previous one, with the measurement identifiers extracted from the `structMap` instead; therefore, the cost of the operation is still $\mathcal{O}(n)$. The `getByDatesAndStructures` view method extracts measurements that belong both to the days and to the structures in input. For this reason, the method needs first to apply `getByDates` and `getByStructures`, then find the common identifiers in the two results. In order to do so efficiently, the results of `getByDates` and `getByStructures` need to be sorted by identifier. The first one is already sorted by construction, while the second one is composed by a set of sorted arrays. Sorting them onto a single array – the merge step of a *Mergesort* algorithm – is a polynomial operation, since the maximum number of structures is fixed. Finally, the algorithm compares the two

sorted arrays and outputs only the intersection, which is again a polynomial operation. In conclusion, this method scans the list of measurements a fixed number of times, therefore its complexity is still $\mathcal{O}(n)$.

IV. PERFORMANCE EVALUATION

In this Section we validate the above-defined SSC through extensive experiments, then we show a practical usage of it through the deployment of a mobile client. The SSC was developed in Solidity (version 0.8.2) by using Hardhat (version 2.3.3) as a development environment. We performed local testing of the SSC and calculated the metrics of execution time and consumed gas over the Hardhat network through a dedicated plugin. The SSC is fully upgradable thanks to the ERC 1967 Proxy pattern by OpenZeppelin. Finally, the SSC is deployed online on the Rinkeby Ethereum Test Network¹. All the tests were performed on an Intel Core i7 (6th Gen) 6700HQ / 2.6GHz with 16GB RAM and Linux Ubuntu 20.04.

A. Results

Using the above experimental setup we carried out several performance tests, which are reported here in Figure 2. In particular, we test the scalability of the three view methods – namely `getByDates`, `getByStructures` and `getByDatesAndStructures` – which are the only ones affected by the amount of data stored in the SSC. For each of them, as a comparison, we implemented a naïve version without the mappings. More in detail, the naïve `getByDates` cycles over all the measurements and, for each of them, it cycles over the dates passed in input and checks for matches. The naïve `getByStructures` has a similar behavior, while the naïve `getByDatesAndStructures` calls both methods and intersects the resulting arrays by checking the presence of every element of the first array in the second, resulting in a quadratic complexity. All the figures compare the SSC with its naïve counterpart described above. More in detail, the first two tests measure the scalability of the SSC for an increasing number of measurements, with dates fixed to 30 and structures fixed to 15. Each query requests 6 dates and/or 3 structures. Figure 2(a) shows the execution time of the function in milliseconds. It is worth noting that, even if in normal conditions view methods would not consume any actual gas, their execution is interrupted if they run out of (hypothetical) gas, just as if they were. This is the case of Naïve SSC, which takes a significantly higher amount of time compared to our implementation, to the point that the `getByDates` and `getByStructures` method do not terminate if the number of measurement is above ~ 1500 . The much more demanding `getByDatesAndStructures` can barely return with ~ 400 measurements. Conversely, our SSC can stand more than 3000 measurements with similar performance for all three methods. We can observe also how the time increases linearly with the size of the input for both SSC and Naïve SSC. In fact, even though the computational

¹<https://www.rinkeby.io/>

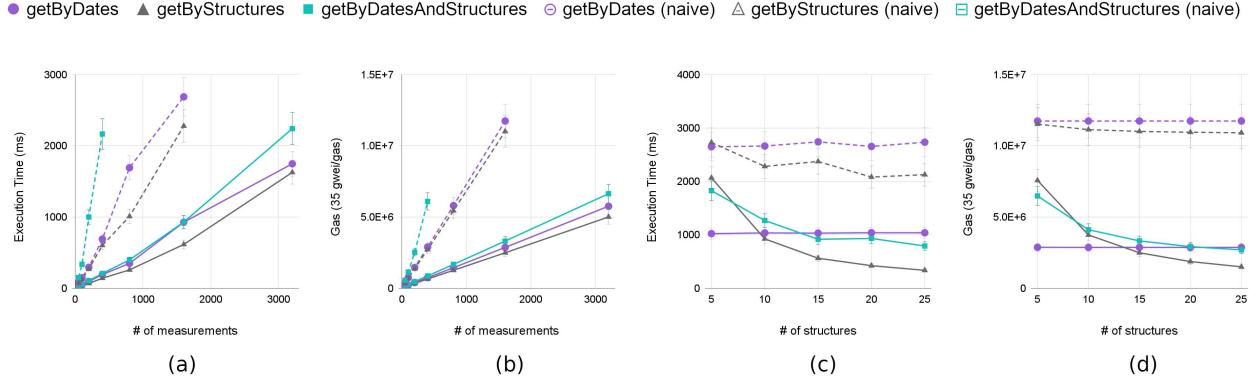


Fig. 2. Performance tests by increasing the number of measurements (structures fixed to 15) in (a) and (b); increasing the number of structures (measurements fixed to 1600) in (c) and (d).

complexity is linear in either case, yet the number of operations performed by the Naïve SSC is much higher. This is not true for `getByDatesAndStructures`, which, in its naïve version, features a quadratic complexity. Figure 2(b) shows a similar trend, however it measures the hypothetical gas. The price for the (hypothetical) unit of gas is fixed to 35gwei – a base unit of measure for gas transactions; 1gwei equals to 10^{-9} ETH – consumed by invoking one of the methods. A different trend can be observed in Figures 2(c) and 2(d), where the number of measurements is fixed to 1600 and the number of structures increases. In the SSC implementation we can observe how the time and the gas consumed by `getByStructures` and `getByDatesAndStructures` monotonically decreases, as, the more structures are deployed, the less measurements are hosted by a single structure. Logically, `getByDates` is unaffected. A similar trend can be observed for Naïve SSC, however the resource consumption is much higher; in fact, `getByDatesAndStructures` does not even run, as it runs out of gas instantaneously.

B. SSC Client Application

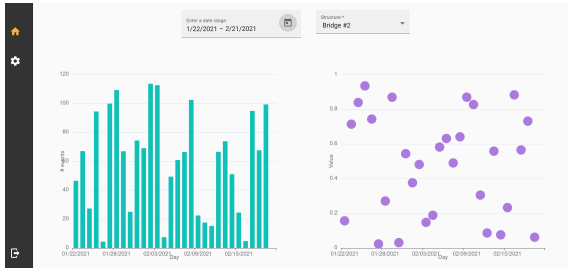


Fig. 3. The web dashboard of the SSC client application.

The client application is a Web dashboard (DApp), whose goal is to retrieve and visualize the SHM data stored in the Blockchain. More in detail, the application allows the user to specify the *network* type, the *SSC* address, and the *endpoint* of the chain from the settings page. Then, it automatically connects to the chain and gets ready for reading the anomalous

values generated by the structure. As shown in Figure 3, users can use a form for filtering the SHM data by *Structure ID* and *dates*. Filtered data points are displayed through a scatter plot and a bar plot. The client is an *Angular 10.1.2* application that relies on the *Infura* APIs to access the chain and on the *Ethers* library to access the Ethereum features.

V. CONCLUSION AND FUTURE WORKS

In this paper we addressed data management issues in IoT-based SHM applications, by investigating how the traceability and immutability of SHM measurements can be granted through Blockchain technologies; to this aim, we discussed how to deploy an SHM-oriented Smart Contract on top of our architecture, and we evaluated its performance in terms of gas and execution time. The joint adoption of WoT and Blockchain technologies in the SHM domain is quite new, and can open the way to extensive evaluations of the Blockchain integration over large-scale SHM scenarios.

ACKNOWLEDGEMENTS

This work has been funded by INAIL within the BRIC/2018, ID=11 framework, project MAC4PRO (“Smart maintenance of industrial plants and civil structures via innovative monitoring technologies and prognostic approaches”). The authors would like to thank Prof. Luca De Marchi and his group for the deployment of the testbeds.

REFERENCES

- [1] C. Arcadius Tokognon, B. Gao, G. Y. Tian, and Y. Yan, “Structural health monitoring framework based on internet of things: A survey,” *IEEE Internet of Things Journal*, vol. 4, no. 3, pp. 619–635, 2017.
- [2] C. Aguzzi, L. Gigli, L. Sciallo, A. Trotta, F. Zonzini, L. De Marchi, M. Di Felice, A. Marzani, and T. S. Cinotti, “Modron: A scalable and interoperable web of things platform for structural health monitoring,” in *2021 IEEE 18th Annual Consumer Communications & Networking Conference (CCNC)*. IEEE, 2021, pp. 1–7.
- [3] W3C Working Group. (2020) WoT Reference Architecture (Proposed Recommendation 9 April 2020). [Online]. Available: <http://www.w3.org/TR/wot-architecture/>
- [4] F. Zonzini, C. Aguzzi, L. Gigli, L. Sciallo, N. Testoni, L. De Marchi, M. Di Felice, T. S. Cinotti, C. Mennuti, and A. Marzani, “Structural health monitoring and prognostic of industrial plants and civil structures: A sensor to cloud architecture,” *IEEE Instrumentation & Measurement Magazine*, vol. 23, no. 9, pp. 21–27, 2020.