ITrade: A Blockchain-based, Self-Sovereign, and Scalable Marketplace for IoT Data Streams

Sina Rafati Niya, Danijel Dordevic, Burkhard Stiller
Communication Systems Group CSG, Department of Informatics IfI, University of Zürich UZH
Binzmühlestrasse 14, CH—8050 Zürich, Switzerland
Emails: [rafati|stiller@ifi.uzh.ch], danijel.dordevic@uzh.ch

***Abstract*—In recent years, the interest grew in the Internet-of-Things (IoT) and Blockchain (BC) integration for additional trust and decentralization. This opened potentials in various use cases, such as supply chain tracing, smart cities, and recently IoT data marketplaces. Therefore, this paper presents the design, implementation, and evaluation of the BC-based IoT data trading platform "ITrade". ITrade proposes a highly scalable microservice-based architecture based on clouds. ITrade enables end-to-end data streaming from IoT devices toward data buyers. The Smart Contract (SC)-oriented design of ITrade enables decentralized management of autonomous and distributed IoT data trading. ITrade evaluations attest its scalability as a reliable peer-to-peer data transmission platform.**

***Index Terms*—IoT, Data Streaming, Data Sovereignty, Blockchain, Smart Contract.**

I. Introduction

The world is reshaped by data connectivity resulting in the creation of new virtual and digital economies with the potential to generate almost 11.1 trillion USD per year by 2025 [13]. Now that the world's most valuable resource is data [24], new business opportunities arise from data monetization, *i.e.,* the process of generating economic benefits from data by trading data. Up to now, high-value personal data is usually handled by centralized databases and servers, belonging to large institutions, organizations, or companies. Such a centralization increases maintenance costs and risk of cyberattacks.

To ensure the highest possible control over the handling of personal data and digital assets, data traders need to be able to rely on data sovereignty [8], which becomes particularly relevant, where data is processed and/or stored [1], since cloud storage services have become increasingly popular as data persistence platforms for many businesses [12].

Blockchains (BC) form a transactional data storage system, which are maintained in a decentralized and distributed fashion. BC use cases have proven to be feasible not only in the Fintech area [7], [22], [16], but also, in IoT-oriented use cases, too [19], [20], including environmental monitoring [18], supply chain tracking [17], industrial cases [21], health care, smart cities, or smart agriculture.

In order to make IoT data more accessible, IoT data marketplaces have been proposed such as IDMOb [14], Datapace [11], Sash [25] and the proposed platform in [15]. These marketplaces provide technical solutions and business incentives for treating data as a tradeable virtual asset. However, this paper's current analysis based on [10] shows that these valuable marketplaces are not meeting user and use case demands with respect to privacy, scalability, and data sovereignty aspects. Therefore, this paper develops ***ITrade*** as a secure and scalable IoT data marketplace based on BCs. *ITrade* implements the decentralized management of data trading via Smart Contracts (SC). As a data streaming platform, it brings together individuals, companies, and organizations from private and public sectors, who are interested in IoT-collected data, for instance *(a)* as within studies performed for health-related use cases, which are most needed in the case of COVID-19 pandemic, and *(b)* individuals, who are interested in selling data collected by their devices in general. IoT owners can initiate data streaming via *ITrade* and benefit from its highly scalable architecture.

The remainder of this paper is organized as follows. Section II discusses deficiencies of current proposals in the IoT data trading field. Section III presents the design of *ITrade*, while Section IV discusses implementation details and the architecture of *ITrade*. Evaluation results of *ITrade* are presented in Section V and, Section VI covers a brief summary.

II. Requirement Analysis

The field study in this work revealed a gap between the state of the art and user expectations or use case demands. The main drawbacks of related work are as follows [10]:

User-friendliness: Most of the solutions proposed require manual steps to start trading data, making the implementation very labor-intensive. In addition, for the solutions developed there is a lack of clear explanations on implementation details.

Performance and Cost: Most of the related work studied commits each transaction (TX) on a BC. Hence, there is a hard cap on the number of TXs stored per second (tps) due to scalability limitations of BCs. Therefore, traders incur high storage and TX costs.

Deployment: Related work provides no or very little information about steps necessary to deploy the respective platform. These solutions do not follow a cloud-native approach [5]. One requirement of the solutions available is to be manually deployed, which makes it challenging to reproduce steps for testing, staging, quality analysis, and production.

Scalability: Storing large amounts of data via TXs on a BC in systems available results in inhibiting scalability. In

978-3-903176-32-4 © 2021 IFIP

addition, underlying data storage systems used by the solutions analyzed tend to lose performance and do not scale well, when the amount of data being handled exceeds 1 TB.

Data Sovereignty: There are three aspects that have to be considered to provide data sovereignty. *(a)* Storing the data in the same geographical location (geolocation) where the data is being generated. In case of employing cloud services, the cloud servers have to stay in the same location, and the Service Level Agreement (SLA) of the cloud provider guarantees that the data does not leave the data center. *(b)* Serving the traffic based on the geolocation of the users. *(c)* Preventing the users from accessing the data marketplaces from restricted locations. In this regard, there is not enough information about how related solutions are providing data sovereignty.

III. ITrade Design

To address the deficiencies of existing data marketplaces, *ITrade* is designed with dynamic elements and scalable modules, which enable sophisticated processes for secure device and user registrations, data streaming, and end to end transaction. The following introduces the entities and processes designed in *ITrade*.

A. ITrade Entities

ITrade contains four main entities, while each entity represents a SC deployed on the Ethereum BC. Thus, these entities are uniquely identified by their SC addresses.

The **Data Stream Principal (DSP)** represents a generic entity within the system that can act both as a Data Stream Seller (DSS) and a Data stream Buyer (DSB). Each DSP has an Owner Address, *i.e.,* its Ethereum's account address, its Name, a URL, and an RSA public key. The RSA key is used for exchanging the symmetric key used for encryption/decryption of a sensor data.

A **Data Stream Seller (DSS)** registers $0 - n$ sensors to the marketplace and sells their data. In addition to the DSP attributes, a DSS has a list of sensors registered by him/her.

A **Data Stream Buyer (DSB)** subscribes to data streams offered by DSSes. DSBs are individuals or organizations are interested in a particular piece of data generated by DSSes.

1) Sensors: Sensors are owned by DSSes. A sensor collects raw data, while being attached to a device with an active Internet connection (*i.e.,* IoT devices) to transmit the data into *ITrade*. As a novel approach and in contrast to the available data marketplaces, in *ITrade* the data being pushed to from IoT devices has to be encrypted on the DSS side beforehand. This prevents *ITrade* administrator and any middleman from being able to read data or misuse such data. Each sensor has the following attributes.

- The **Type** defines the data type being collected by sensors, such as a temperature, humidity, air, or water pollution index.
- A **Status** defines for every sensor upon its registration the state it is in. The DSS has to activate it before being able to send data from it. The sensor can also be deactivated or blocked by its owner.
- The **Geolocation** is represented by longitude and latitude of a sensor.
- The **Price per Data Entry** defines the price that a DSB has to pay, when subscribing to the sensor's data stream. Buyers have to enter the number of entries they want to buy.
- An **AES Private Key** is generated by DSS for each of his/her sensors. These private keys have to be securely stored by DSS locally.

2) Data Marketplace (MP): An MP is the central entity of the *ITrade* design. It embeds multiple micro-services for enabling ingestion, streaming, and persisting data. An MP shows the following attributes:

- The **MP Owner Address** is the address of *ITrade* owner and administrator.
- A **DSP Registration Price** defines that price each DSP has to pay when registering to the marketplace.
- The **Sensor Registration Price** defines a flat fee for registering sensors.
- The list of **Registered DSPs** names all DSPs registered.
- The **DSP Commission Rate** for each DSP defines a certain percentage of its price, which is transferred to the MP's Ethereum account as a streaming fee.

3) Data Stream Subscription (DSSub): A DSSub represents a subscription to a data stream. Each time a DSB subscribes to a data stream, a new DSSub SC is created. A DSSub operates on the following attributes.

- A **DSB ID** represents the identifier (ID) of the DSB.
- A **Sensor ID** defines the ID of the sensor to which the DSB has been subscribed to.
- The **Start Timestamp** defines the time from which the DSB has been subscribed to a sensor's data stream.
- The **Number of Data Entries** determines those data stream entries that the DSB subscribed to.

B. User Interactions and Processes in ITrade

1) DSP Registration: In interacting with *ITrade*, the first phase is DSP registration phase where a DSP registers to the MP by providing basic information about him/her-self together with a public RSA key. The RSA key-pair will be automatically generated by *ITrade*'s Web client. The DSP deploys a SC on the BC. The deployed SC represents a DSP instance. The DSP also has to pay the registration fee that is predefined by the *ITrade*'s admin. Once the DSP has been successfully registered, he/she can act both as a DSS or a DSB. Upon a successful registration, the DSP has to securely store the newly generated RSA private key which will be used for streaming data process in later steps.

2) Sensor Registration: For a sensor registration the DSS client generates an AES key used for encrypting sensor data. The key has to be securely stored by the DSS. The DSS initiates the sensor registration process by registering the sensor on the BC. This step incurs the sensor registration fee that is automatically transferred to *ITrade* admin's Ethereum account. The registration TX ID is then returned to the DSS.

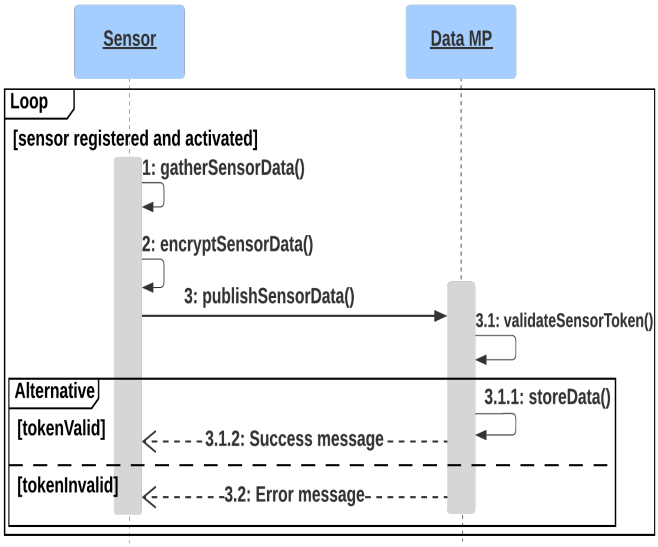

Fig. 1. Publishing Sensor Data in *ITrade*

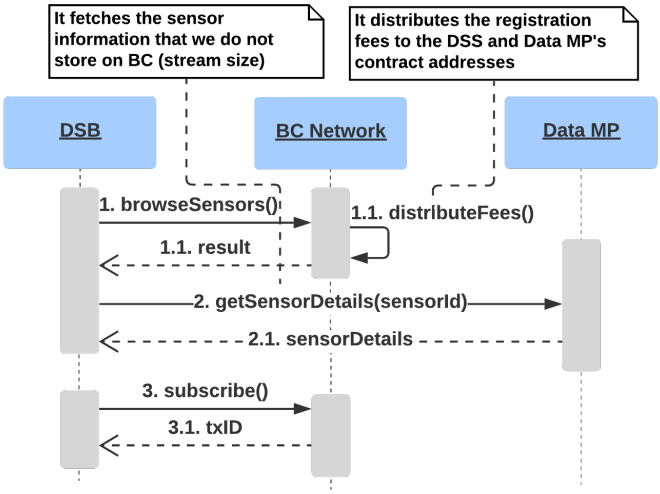

Fig. 2. Subscribing to a Data Stream Process Flow in *ITrade*

The next step is to activate the sensor on *ITrade*. The DSS does so by initiating a call that includes the TX ID obtained in the previous step. *ITrade* validates the sensor by validating the TX against the BC. Next, the result is returned to the Data MP and *ITrade* generates a sensor token that will be used for the DSS authentication when publishing that sensor's data streams. In case the TX is valid, *ITrade* will configure the sensor so that the data can be ingested to and streamed from the platform. Finally, the sensor token is returned to the DSS.

3) Publishing Sensor Data: Once a sensor is successfully registered and activated by DSS, it can start publishing data via Kafka Service API to the marketplace. DSS is the one who decides on how frequent shall the data collection happen, and which type of data shall be collected as he/she has the control over his/her IoT sensors. A sensor publishes the data to *ITrade*, where each iteration involves those steps as of Figure 1.

1. The sensor captures data from its physical environment.
2. Raw data is encrypted with the sensor's AES key.
3. The device calls *ITrade* via an API for which the payload corresponds to the encrypted sensor data and a sensor token that indicates sensors validity.

3.1. *ITrade* receives the call and validates the sensor token to detect possible malicious actors.

3.1.1. If the token is valid, *ITrade* will store the data and make it available for streaming.

3.1.2. In turn, a success message will be returned to the sensor device.

3.2. If the sensor token is invalid, an error message will be returned.

4) Subscribing to a Sensor's Data Stream: DSBs can search for available data streams at *ITrade*. If DSBs want to select a certain data stream, they have to subscribe to it, which is possible only by paying for each data entry. Consequently, a subscription amount will depend on the number of data entries DSB wants to buy. Payments at purchase are managed and enforced by Smart Contracts [10] (*cf.* Figure 2).

Once a DSB subscribes to a data stream, it can start obtaining a key from *ITrade* that will be him/her to decrypt the data stream messages. The key exchange process between DSS and DSB is as follows (*cf.* Figure 3):

1. Once a DSB subscribes to a data stream, it informs *ITrade* about that by sending the TX ID and the corresponding sensor's public key.
2. *ITrade* verifies that TX in the BC and requests the DSS to encrypt the data stream decryption key with the DSB's public key, which is sent to the MP.
3. The DSB can now fetch the encrypted key, decrypts it with its private key, and starts streaming the data. Thus, the data streamed are decrypted on the DSB side.

While Figure 2 visualises this process, the additional complexity introduced is highly beneficial, since it prevents *ITrade* from manipulating data it stores.

5) Streaming Data: A DSB can start streaming the data from the chosen sensor as shown in Figure 3:

1. A DSB makes a request against the Data MP API in order to obtain the data stream's decryption key. The request contains the data stream subscription purchase TXID.

1.1. The Data MP validates the TX with the Smart Contract.
1.2. All result is returned from the BC .
1.3. If the TX is valid, the Data MP will require the decryption key from the DSS.
1.4. The DSS will encrypt the data stream decryption key with the DSB's public key.
1.5. The key is returned to Data MP.
1.6. The Data MP keeps the key in the cache for subsequent requests.
1.7. The encrypted decryption key is returned to the DSB.
1.8. If the TX from step 1.1 is not valid, an error message is returned to the DSB.

2. The DSB asks the Data MP for those data received.

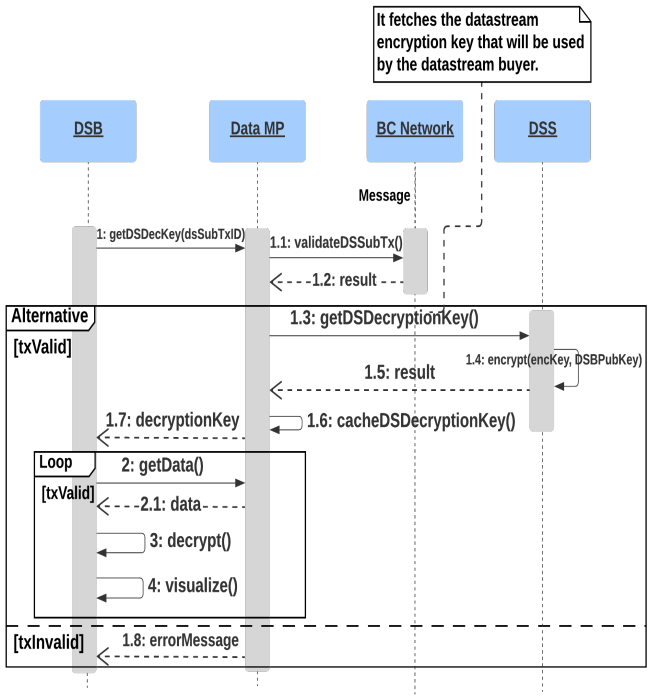

Fig. 3. Data Streaming Process Flow in *ITrade*

2.1. The DSB receives the result.

3. The data is decrypted with the key obtained in step 1.7.
4. The data subscribed to is streamed to the DSB.

Even though the data streaming process is already personalized within the registration processes, the use of static keys per stream follows the standard approach of using new ones for each new session. Thus, data buyer cannot reuse that key for future purchases.

IV. Implementation

ITrade is a cross-technology platform, established on the cloud, and integrated with orchestration tools and load-balancers. While the source code of *ITrade* 's implementation is made available at [9], the following overviews technologies and tools used, the architecture, and its implementation details.

A. Blockchain (BC) and Smart Contracts (SC)

ITrade is using the Ethereum BC, since it is supported by a large community and it obtains a large set of libraries as well as a built-in cryptocurrency enabling a seamless implementation of the payment system. In order to join the Data MP, each user has to have an Ethereum account. Each action (*e.g.,* sign in/up, sensor registration, and subscribing to data streams) is captured by means of deploying the corresponding SCs.

1) Data Marketplace: It defines the method for registering Data Stream Principals (DSP). A new DSP has to pay the registration fee to the DP owner, hence the *registerDataStreamPrincipal* method is of type *payable*. Upon successful call of this method, a new instance of the DSP SC will be deployed (*cf.* Listing 1).

```
contract DataMarketplace {
    function registerDataStreamPrincipal(
        string _dataStreamPrincipalName,
        string _dataStreamPrincipalURL,
        string _dataStreamPrincipalEmail,
        string _rsaPublicKey
    ) public payable;}
```

Listing 1. Data Marketplace Smart Contract

2) Data Stream Principal (DSP): DSP SC has the method **registerSensor()** that is called when a new sensor gets registered by a DSS (*cf.* Listing 2). The DSP has to provide info on the sensor type, geolocation data, and the price that the DSBs have to pay for each data entry. This method creates a new instance of the Sensor SC.

```
contract DatastreamPrincipal {
  function registerSensor(
    IoTDataMPLibrary.SensoryType _sensorType,
    string _latitude,
    string _longitude,
    uint _pricePerDataUnit
    ) public payable;}
```

Listing 2. Datastream Principal Smart Contract

3) Sensor: Sensor SC (*cf.* Listing 3) represents a sensor entity and it defines the method **subscribe()** that expects the following parameters. *(a)* the DSB contract address, *(b)* the start timestamp *i.e.,* the time since when the DSB is subscribed, and *(b)* the number of the data entries that the DSB is subscribed for. This method deploys an instance of the data stream subscription SC.

```
contract Sensor {
  function subscribe(
    address _dataStreamBayerContractAddress,
    string _startTimestamp,
    uint128 _dataEntries
    ) public payable;}
```

Listing 3. Sensor Smart Contract

4) Data Stream Subscription: It represents a subscription to a specific sensor and it has a method that returns true or false if the subscription is active or not (*cf.* Listing 4).

```
contract DatastreamSubscription {
  function isDatastreamSubscriptionValid()
    public view returns (bool);}
```

Listing 4. Datastream Subscription Smart Contract

Complete SC codes are available at [9].

B. Streaming System

There are several aspects that should be considered before choosing a streaming system such as *(a)* **Message consumption model:** with two options: *pull-based* mechanisms allow the consumers to manage their message flow *i.e.,* users pull only the messages they need. In contrast, *Push-based* mechanisms put too many responsibilities on a streaming system

since the system would need to manage the message consumption for each consumer which is not a scalable approach. Thus, a pull-based approach is preferable in a data marketplace. *(b)* **Number of components** and, *(c)* **Storage architecture**.

Among the possible streaming protocols such as Pulsar and Kafka, *ITrade* implementation is based on Kafka. This decision is due to the fact that Pulsar uses an index-based storage system that forms a tree structure. That enables fast access to the messages but introduces the write latency. Both Kafka and Pulsar retain the messages indefinitely meaning both can be used as storage systems. Kafka uses fewer components than Pulsar. That makes Pulsar more difficult to deploy and manage. Kafka uses a commit log as a storage layer. New messages are appended at the end of the log. Reads are sequential starting from the offset and moving towards the end of the log [2].

In *ITrade* each sensor has its own Kafka topic where it can publish the data and the subscribers can stream from. The sensor's SC address is used for the topic name since the addresses are already globally unique.

C. ITrade Architecture

ITrade services are orchestrated and deployed on a Kubernetes cluster. It can be seen (*cf.* Figure 4) that each component inside the cluster is deployed as a Kubernetes service. This approach provides an internal load balancer in front of a group of running containers and enables scaling of each group of components without the users noticing it.

Figure 4 represents the components in *ITrade* and their correlation. An internal Kafka cluster provides the streaming and storage features. The whole cluster is deployed within a private virtual network. The only public-facing component is the load balancer that has the TLS certificates attached to it for a secure connection. Both DSSes and DSBs access the corresponding services via the load balancer.

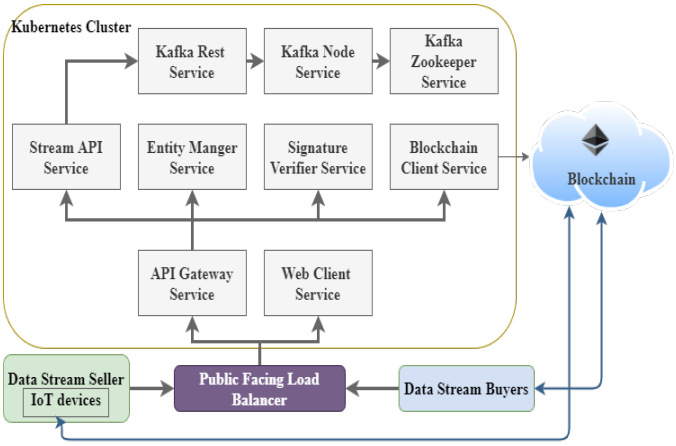

Fig. 4. Component Overview of *ITrade* Architecture

1) API Gateway: API Gateway is the *front door* for *ITrade*. It exposes the HTTP GraphQL endpoint that the clients (*ITrade* users) can call. API Gateway is implemented in Java programming language with help of the Spring Boot framework. GraphQL is a query language that enables fetching the data needed on the client-side. It also enables the fetching of multiple resources with a single API call, while REST APIs would require loading the results from multiple URLs.

2) Entity Manager: Since the goal in *ITrade* implementation is to have a high-performance system while still using a BC, the Entity Manager is implemented as a caching service to achieve this goal. Therefore, each call made to the BC is cached for a certain amount of time in order to improve the performance of the system. The caching is backed up by a key-value in-memory database.

When a DSB subscribes to a data stream, he/she will provide the transaction ID as a proof that he/she has been subscribed. The API Gateway will first call the BC client to validate the transaction. The BC client will then return the result to the API gateway which will save the transaction information to the Entity Manager in case the transaction is valid, otherwise, it will reject the client call. When the DSB starts streaming the data, he/she has to include the JSON Web Token (JWT) in the authorization header, that token gets checked by the Entity Manager service. A similar process applies when publishing the data from the IoT devices. Each activated sensor has a JWT token assigned to it which has to be included in each call thus the clients can be authenticated.

The signature verifier component provides a passwordless authentication for users. *ITrade* utilize a cryptographically secure authentication flow with help of the Web3.js library [4]. *ITrade* relies on the property that it is cryptographically easy to prove the ownership of an account (*i.e.,* Ethereum account) by signing a piece of data with the user account's private key. *ITrade*'s specific implementation uses a message-signing-based authentication mechanism where the users are identified by their Ethereum account addresses.

V. Evaluations

To evaluate and monitor the components involved in the microservice-based architecture of *ITrade*, the Istio [6] platform is employed. Istio embeds a proxy (*i.e.,* Envoy) in front of each service to capture each request, by which the latency, request/response size, and success rate are measured. The Envoy proxy uses 0.5 Virtual CPU (vCPU) and 50 MB memory per 1000 requests-per-second going through it. Envoy adds 3.12 ms to the 90th percentile latency. Additionally, this test environment runs the control plane component of Istio with 2 vCPU and 4 GB memory.

In order to measure the performance and user experience of *ITrade*, the "*Percentile Latency*" is used as a main metric. The *Percentile Latency* gives the maximum latency for the fastest percentage of all requests. For instance, **P50 Latency** gives the maximum latency for the fastest 50% of all requests.

In the evaluation of *ITrade* "*Distributed Tracing*" is employed for monitoring and profiling. *Distributed Tracing* promotes the idea of distributed context propagation, which means that each request has associated metadata to be followed across multiple microservices [3]. Considered parameters of this approach include *(a)* Trace: It is the sequence of calls through

the system that are needed to resolve a request. Although only one call against an entry point in the distributed system may be able to resolve the request right away, oftentimes multiple subcalls are needed to work together in order to resolve the request. *(b)* Span: When a trace includes multiple subcalls, each subcall represents a span. Each subcall is timed and accepts key-value tags as well as fine-grained, timestamped, structured logs attached to a particular span instance. *(c)* Span context: It is a metadata attached to each span used for linking spans to their trace. Such a distributed tracing mechanism offers deep insights into the HTTP requests. Distributed Tracing helps to inspect requests against tail latency [23], the overall latency, and to detect the particular services that are the biggest contributors to those delays.

In an evaluation of *ITrade*, 100 IoT devices transmit the sensor data at about 2000 messages per second. 100 clients (representing DSBs) simultaneously stream (receive) the data at about the same rate. They all make requests per second via the public-facing load balancer. The total number of requests steadily increases up to 5000 requests per second. The test ran for a almost an hour. In this test *ITrade* was deployed in Amazon Web Service (AWS) servers and it was connected to an Ethereum test network.

The P99 latency was as high as 2.3 s for about 60% of the testing time. The P90 latency was much lower, ranging from 50 ms to more than 1 s at the very end of the testing. The P50 latency maintained a very low value *i.e.,* 23 ms. This means with *ITrade*, the maximum delay experienced for the 50% of streaming is at most only 23 ms.

P50 latency: maintained low values of 23.59 ms for the API Gateway which is satisfactory. The biggest contributor to this latency was the downstream Kafka REST service with 12.63 ms delays.

P90 latency: sees a considerable amounts for latency. Here is where the *ITrade* architecture started reaching the limits of the Kafka REST component since the latency for the API Gateway was as high as 480.21 ms and about 87% of that latency can be attributed to Kafka REST component (420.51 ms). The most significant latency experienced was for the Stream API service which was expected since it is an upstream service of the Kafka REST.

P99 latency: showed a latency for the API Gateway equal to $\sim$1.7 s. 97% of that latency can be attributed to the Kafka REST downstream service. Kafka REST components had a few packets dropped so the overall success rate was 99.99%. Other services did not experience the same issue, meaning their success rate was 100%.

After running the several scenarios, it is concluded that scalability of *ITrade* reached certain limitations due to the external Kafka REST component due to its stateful consumer model, hence applying the K8s HPA is not applicable in this case. Changing this approach would require significant engineering effort meaning that the only way to overcome this issue at the time of writing is to make sure that Kafka REST workload on network and computing is scheduled and optimized upfront.

Moreover, it can be concluded that the employed components in *ITrade* architecture are reliable and able to scale horizontally. For instance, building a Stream API service that abstracts enables to switch to another streaming system in the future without changing the rest of the system. Should that be addressed, higher performances might be achieved.

Since *ITrade* is deployed on a cloud environment, data sovereignty is of particular importance. *ITrade* is compliant with European data sovereignty regulations. It is deployed in Europe and ensures data sovereignty for Europe only. *ITrade* is hosted on AWS in Europe which guarantees the data does not leave the data center. Traffic is being routed to the right location by utilizing the geolocation routing policy. This means that the Domain Name Service (DNS) server will resolve the DNS queries to the IP addresses according to the geographic location of the users. Restricted access to *ITrade* from restricted locations is ensured by enabling the Web Application Firewall (WAF) service provided by AWS. *ITrade* utilizes the WAF service by filtering the traffic by the source IP addresses. Meaning the traffic that originates outside Europe will be blocked. Additionally, data sovereignty is ensured in *ITrade* by encrypting/decrypting the data on the user's side. This means that even if a malicious actor gets possession of the data, such encrypted data is useless without the decryption keys.

VI. Summary

This paper presented the *ITrade* platform, a modular and secure IoT data marketplace following a microservice-based architecture. The goal of *ITrade* was, on one hand, to make it easier for individuals and companies to benefit from their self-generated data in financial terms, and, on the other hand, to help data seekers such as in healthcare, academic, or social studies, to access a large number of data providers easily and without privacy violations.

Evaluations on the scalability of *ITrade* have shown that a viable Blockchain-integrated approach was reached. The reliability of data streaming with *ITrade* is above 99%, even when massive number of transactions being streamed. This is due to the orchestration and streaming technologies implemented by Kubernetes and Kafka, respectively, all being integrated into the processes designed in *ITrade* . The decentralization in *ITrade* is fulfilled via several Smart Contracts, while user privacy and data sovereignty is fully guaranteed.

As part of future work, it is envisioned to develop and deploy mechanisms for secure transmission of data and preventing data buyers from reselling the data once acquired.

VII. Acknowledgements

This paper was partially supported by (a) the University of Zürich UZH, Switzerland, and (b) the European Union Horizon 2020 Research and Innovation Program under grant agreement No. 830927, namely the Concordia project.

References

[1] "Data Sovereignty and The Cloud," https://www.itgovernance.co.uk/data-sovereignty-and-the-cloud, last visit: October 1, 2020.

[2] "Kafka vs Pulsar - Performance, Features, and Architecture Compared," https://www.confluent.io/kafka-vs-pulsar/, last visit: October 1, 2020.

[3] "Opentracing overview," https://opentracing.io/docs/overview/, last visit: October 1, 2020.

[4] "web3.js - Ethereum JavaScript API," https://web3js.readthedocs.io/en/v1.3.0/, last visit: October 1, 2020.

[5] "CNCF Cloud Native Definition v1.0," https://github.com/cncf/toc/blob/master/DEFINITION.md, June 2018, last visit: October 18, 2020.

[6] I. Authors, "Istio / Performance and Scalability," https://istio.io/latest/docs/ops/deployment/performance-and-scalability/#performance-summary-for-istio-hahahugoshortcode-s0-hbhb, September 2020, last visit: October 1, 2020.

[7] T. Bocek, S. Rafati, B. Rodrigues, and B. Stiller, "CoinBlesk - A Real-time, Bitcoin-based Payment Approach and App," *ERCIM News: Blockchain Engineering*, Vol. 1, No. 110, pp. 14–15, July 2017. [Online]. Available: https://ercim-news.ercim.eu/en110/special/coinblesk-a-real-time-bitcoin-based-payment-approach-and-app

[8] S. Couture and S. Toupin, "What Does The Notion of 'Sovereignty' Mean When Referring to The Digital?" *New Media and Society*, Vol. 21, No. 10, pp. 2305–2322, 2019. [Online]. Available: https://doi.org/10.1177/1461444819865984

[9] D. Dordevic, "BIoT Data Market Place," https://github.com/IoT-Data-Marketplace, last visit: October 1, 2020.

[10] ——, "Data Sovereignty Provision in Cloud-and-Blockchain-Integrated IoT Data Trading," Master's thesis, Zürich, Switzerland, September 2020. [Online]. Available: https://owncloud.csg.uzh.ch/index.php/s/AFyTkCf2Dfrd99A

[11] D. Draskovic and G. Saleh, "Datapace - Decentralized Data Marketplace Based on Blockchain," in *Datapace*, December 2017.

[12] Gartner, "Gartner Forecasts Worldwide Public Cloud Revenue to Grow 17% in 2020," shorturl.at/ltE79, November 2019, last visit: October 1, 2020.

[13] J. Manyika, M. Chui, P. Bisson, J. Woetzel, R. Dobbs, J. Bughin, and D. Aharon, "The Internet of Things Mapping the Value Beyond The Hype," https://owncloud.csg.uzh.ch/index.php/s/abaGWX2PzPLnc7W, June 2015, last visit: October 1, 2020.

[14] K. R. Ozyilmaz, M. Dogan, and A. Yurdakul, "Idmob: Iot data marketplace on blockchain," in *Crypto Valley Conference on Blockchain Technology (CVCBT)*, 2018, pp. 11–19.

[15] R. Radhakrishnan, G. S. Ramachandran, and B. Krishnamachari, "SDPP: Streaming Data Payment Protocol for Data Economy," in *IEEE International Conference on Blockchain and Cryptocurrency (ICBC)*, 2019, pp. 17–18.

[16] S. Rafati Niya, S. Allemann, A. Gabay, and B. Stiller, "TradeMap: A FINMA-compliant Anonymous Management of an End-2-end Trading Market Place," in *15th International Conference on Network and Service Management (CNSM)*. Halifax, Canada: IEEE, October 2019, pp. 1–4. [Online]. Available: https://ieeexplore.ieee.org/document/9012706

[17] S. Rafati Niya, D. Dordevic, A. G. Nabi, T. Mann, and B. Stiller, "A Platform-independent, Generic-purpose, and Blockchain-based Supply Chain Tracking," in *IEEE International Conference on Blockchain and Cryptocurrency (ICBC 2019)*. Seoul, South Korea: IEEE, May 2019, pp. 11–12. [Online]. Available: https://ieeexplore.ieee.org/document/8751415

[18] S. Rafati Niya, S. S. Jha, T. Bocek, and B. Stiller, "Design and Implementation of an Automated and Decentralized Pollution Monitoring System with Blockchains, Smart Contracts, and LoRaWAN," in *IEEE/IFIP Network Operations and Management Symposium (NOMS 2018)*, Taipei, Taiwan, April 2018, pp. 1–4. [Online]. Available: https://ieeexplore.ieee.org/document/8406329

[19] S. Rafati Niya, F. Maddaloni, T. Bocek, and B. Stiller, "Toward Scalable Blockchains with Transaction Aggregation," in *5th Annual ACM Symposium on Applied Computing (SAC 20)*, 2020, p. 308–315. [Online]. Available: https://doi.org/10.1145/3341105.3373899

[20] S. Rafati Niya, E. Schiller, I. Cepilov, F. Maddaloni, T. Surbeck, K. Aydinli, and T. Bocek, "Adaptation of Proof-of-Stake-based Blockchains for IoT Data Streams," in *IEEE International Conference on Blockchain and Cryptocurrency (ICBC)*. Seoul, South Korea: IEEE, May 2019, pp. 15–16. [Online]. Available: https://ieeexplore.ieee.org/document/8751260

[21] S. Rafati Niya, E. Schiller, I. Cepilov, and B. Stiller, "BIIT: Standardization of Blockchain-based I2oT Systems in the I4 Era," in *Management in the Age of Softwarization and Artificial Intelligence*. Budapest, Hungary: IEEE, April 2020, pp. 1–9. [Online]. Available: https://ieeexplore.ieee.org/document/9110379

[22] S. Rafati Niya, F. Schüpfer, T. Bocek, and B. Stiller, "A Peer-to-peer Purchase and Rental Smart Contract-based Application (PuRSCA)," *it-Information Technology*, Vol. 60, No. 5, pp. 307–320, October 2018. [Online]. Available: https://www.degruyter.com/view/journals/itit/60/5-6/article-p307.xml

[23] section.io, "Preventing Long Tail Latency," https://www.section.io/blog/preventing-long-tail-latency/, November 2018, last visit: October 1, 2020.

[24] The Economist, "Regulating the Internet Giants - The world's most Valuable Resource Is No Longer Oil, But Data," https://www.economist.com/leaders/2017/05/06/the-worlds-most-valuable-resource-is-no-longer-oil-but-data, June 2017, last visit: October 1, 2020.

[25] H. T. T. Truong, M. Almeida, G. Karame, and C. Soriente, "Towards Secure and Decentralized Sharing of IoT Data," in *IEEE International Conference on Blockchain (Blockchain)*, 2019, pp. 176–183.