On Supporting Contract-aware IoT Dataspace Services

Florin-Bogdan Balint
Distributed Systems Group, TU Wien
Vienna, Austria
florin.balint@student.tuwien.ac.at

Hong-Linh Truong
Distributed Systems Group, TU Wien
Vienna, Austria
truong@dsg.tuwien.ac.at

***Abstract*—Advances in the Internet of Things (IoT) enable a huge number of connected devices that produce large amounts of data. Such data is increasingly shared among various stakeholders to support advanced (predictive) analytics and precision decision making in different application domains like smart cities and industrial internet. Currently there are several platforms that facilitate sharing, buying and selling IoT data. However, these platforms do not support the establishment and monitoring of usage contracts for IoT data. In this paper we address this research issue by introducing a new extensible platform for enabling contract-aware IoT dataspace services, which supports data contract specification and IoT data flow monitoring based on established data contracts. We present a general architecture of contract monitoring services for IoT dataspaces and evaluate our platform through illustrative examples with real-world datasets and through performance analysis.**

***Keywords*-IoT, Thing-as-a-Service, Data Contract, Monitoring, Quality of Data, Quality of Service.**

I. Introduction

The number of IoT devices has increased drastically in the past few years, providing various types of IoT data. Let us consider the following scenario in which there is a network of hundreds or maybe even thousands of devices which provide different types of measurements (e.g., air temperature, atmosphere pressure, and air quality). For example, in the Array of Things project, different *Things* are installed to monitor Chicago's environment [1]. In this scenario different stakeholders might be interested in different types of data produced by these devices: local news agencies might be interested in temperature recordings, universities and research institutes might be interested in air quality data, and public health institutes might be interested in the measurement of contaminants in the air.

Nevertheless, all stakeholders have one thing in common: they need to make sure that they get and use the data based on certain constraints w.r.t. quality of data, data usage, price, to name just a few. These constraints can be specified in data contracts. For example, in [2], a conceptual view on sharing IoT data among different companies has been outlined to emphasize the importance of contracts. Therefore, we must ensure, that different data contracts can be negotiated and concluded for different data produced by different *Things*. However, assuming that 1 to n data contracts can be concluded, and each data contract is concluded for the data produced by 1 to m *Things*, a lot of different data contracts can exist. Furthermore, each of these data contracts can have different data contractual clauses (e.g., different subscription periods and different price). Thus, data contracts for IoT are very complex and it is not clear how such contracts would be established, monitored and enforced in such a conceptual view. Existing IoT data platforms have not supported such contracts adequately. The main challenge in these scenarios is to monitor (big) data flows for different data contracts according to their contractual clauses in real-time. Allowing the monitoring of the data flow for a data contract enables the involved parties to detect data contract violations at runtime. For example if a data contract is concluded for a certain type of quality of data (e.g., schema completeness) and the submitted data flow does not meet the expected values, then this could be detected almost in real-time or only with a short delay, while data is still being transferred and the data contract subscription period is still running.

Multiple IoT platforms and data marketplaces (e.g., tilepay [3], MARSA [4] and BDEX [5]) have been created to facilitate IoT data sharing, purchasing and selling. These platforms offer a variety of features like multiple pricing models and near real-time data transfer, however, they are limited regarding the data contract management (covering activities from individual data contract negotiation and until the delivery of data flows met contract terms). Our work is motivated by the absence of contract-aware platforms, which enable the monitoring of the data flow of individual data contracts in IoT dataspaces. As we discuss in the related work, there are many industrial systems that enable the development of IoT dataspace/hub, such as Microsoft IoT [6], Amazon IoT [7], Predix.io [8], etc., these systems lack tools and services to allow the incorporation of services required for contract establishment and monitoring. The main contributions of this paper are: (i) a new extensible framework for data contract management, which takes into account both technical and business aspects to allow the monitoring of individual IoT data contracts, and (ii) an extensible architecture, which supports the extension with custom microservices for data contract monitoring.

The remainder of this paper is organized as follows: Section II of this paper presents the background and related work. Section III describes the new proposed framework and architecture. In Section IV-A we describe our prototype implementation. In Sections IV-B and IV-C we describe our experiments and present our evaluation. Finally, in Section V we summarize the paper and outline future work.

II. Background and Related Work

A. Background and Approach

Dataspaces have been already introduced as an "abstraction of data management" and as a "data co-existence approach" [9]. We define an *IoT dataspace* as the dataspace, where all kind of data is available from multiple IoT devices or *Things*, which all belong to multiple providers. Like in the current state of the art in IoT data ingestion and management, we consider IoT dataspaces implemented using concepts of data hubs: streams of IoT data are aggregated in centralized clouds or distributed edge servers via queues and eventually data are stored or relayed to other data nodes (consumers or integrators). Examples are to utilize industrial data hubs like DeviceHub.net [10], IBM Internet of Things - Big Data & Analytics Hub [11] and Equinix Data Hub [12].

In some cases, people might be interested to purchase data produced by some devices during a specific time period (e.g., quality of seawater for lobster farms during the raining seasons). In these cases data, which is produced by *Things*, is traded. This can be also related to *Thing-as-a-Service*, because data produced by different *Things* is bought/sold for a limited period of time.

1) IoT Data Contracts: To trade data (produced by *Things*), we must use data contracts for sharing, purchasing and selling data. In Truong et al. [13] the main properties of data contracts were investigated. Several properties of data contracts have been presented such as data rights, quality of data, regulatory compliance, pricing model, and control and relationship. We adopt them for IoT data contracts in this paper.

2) Quality of Data: The quality of data (QoD) is one aspect that can be taken into account for data contracts, because IoT data has certain quality attributes. The challenge here is to measure the QoD, as part of data contract monitoring. Multiple dimensions have been associated with respect to data quality [14], [15], [16], [17], [18]. One of them is data completeness, which is generally defined as "the extent to which data are of sufficient breadth, depth, and scope for the task at hand" [16]. For our new proposed data contract model (see also Section III) we consider schema completeness measured in percent, because in some cases, where an IoT data transmission message consists of multiple attributes, if one attribute is missing, the whole message might become worthless (e.g., a GPS device broadcasting information, but due to some error the latitude is missing or NULL).

Another metric in our data contract model is data conformity, which "describes how well data adheres to standards and how well it's represented in an expected format" [19]. The expected data format can play an important role, especially when processing data (e.g., if the data is not in the expected format, then the data might not be processed).

The challenge to measure data completeness and conformity for a transmitted message lies in knowing the exact structure (or model) of the message (i.e., what attributes exist and what data types they have). In our work, conceptually, we access structure information from systems with registered *Things*, such as HINC [20] or AWS IoT Registry[1].

Our data contract also supports data currency which indicates "how promptly data is updated" [14]. For example, in a smart city, in highly polluted industrial areas, when purchasing air quality data, e.g., for the measurement of contaminants, the data values are expected to be current, if the values are delivered with a delay of e.g., a few hours, the data might be worthless.

3) Quality of Service (QoS): Besides the QoD, the QoS can also be bound to be part of a data contract. For example *Things* are expected to provide different measurements within a certain frequency. Furthermore, it might be in the interest of the involved parties of a data contract to find out if the expected broadcasting frequency does not equal the actual broadcasting frequency.

B. Related Work

In [21] quality data metrics for relational data stream management systems are discussed. Essentially, such metrics are important for data contracts. Other papers like [18] surveyed, presented and discussed data quality metrics and evaluation methods for sensors and IoT. Our work differs as we do not focus on methods for evaluating QoD but support QoD in the contract by incorporating existing methods for evaluating QoD. Challenges in QoD in smart cities are discussed in [22]. Our framework could be utilized to address some aspects in these challenges.

Several platforms have been developed for purchasing and selling data produced by IoT devices. MARSA is a dynamic cloud-based marketplace for near real-time human-sensing data [4]. This platform supports near real-time delivery of data from consumers to buyers [4] and multiple cost models. However, it does not support the individual negotiation of the data contractual terms and the monitoring of such contracts. If a user wants to provide and buy data at the same time, two different accounts are necessary since the platform strictly distinguishes between these two types of users. Another platform is BDEX [5], which supports the monitoring of the data exchange but no individual contract conclusion.

Industry systems, like Microsoft Azure IoT, Amazon IoT, and Google IoT, have "IoT data hubs" which include brokers and data ingest clients that store data into different types of databases but do not support IoT data contract management. IoT dataspaces can be seen as similar to IoT data hubs, because in an IoT dataspace large amounts of data are produced by multiple IoT devices. We implemented a scalable architecture based on messages brokers and microservices to support individual data contract conclusion. Through a provided API our components can be plugged in to existing IoT data hubs in order to support individual data contract conclusion and monitoring. As a proof-of-concept we developed a prototype which can run on its own and act as an IoT data hub.

[1]http://docs.aws.amazon.com/iot/latest/developerguide/iot-thing-management.html

III. Contract-aware Platform for IoT Data

A. Architecture

In our framework, we consider *Things* that publish data to IoT data hubs. Figure 1 illustrates our view on IoT dataspaces. In an IoT dataspace, multiple IoT devices – called *Things* – produce different types of data. An IoT provider can own several *Things* and might want to sell his/her data to different data consumers. Hence in an IoT dataspace, with the help of our system (discussed later), data from different providers will be handled as *Data Packages* according to specific data contracts and will be delivered to the consumers.

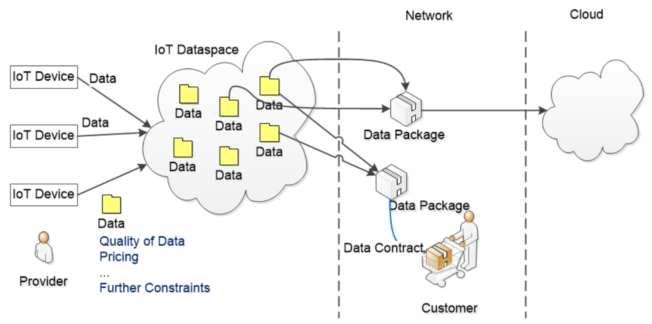


Fig. 1: IoT Dataspace Concept

To support data contract monitoring and enforcement, we developed a microservice-based architecture, shown in Figure 2. All microservices can be scaled and run on their own in different types of virtual environments and they are managed through a *ServiceHandler*. The *ServiceHandler* provides a plug-in API for integrating multiple microservices of different types, including custom made, which makes our architecture easily extensible by custom services. We developed two types of microservices: *Monitoring* and *DataContract* microservice. With the microservice architecture, instances of services in our framework can be easily deployed into virtual machines and containers to deal with the scale of IoT dataspaces. For the sake of simplicity, we are assuming that all IoT data stream messages are published to message brokers/queues from where they can be consumed. Different *Things* might publish the data to separate queues or to the same queue. We further assume that multiple queues can be grouped together to form data nodes.

The *Monitoring* microservice is responsible for continuous consumption of all data stream messages from data nodes. One *Monitoring* microservice instance can be configured to consume all messages from one data node. After messages are consumed, they are automatically redistributed to all relevant subscribers (in accordance to established data contracts). While redistributing messages, QoD and QoS metrics can be computed and verified by the *Monitoring* microservice. The *DataContract* microservice is responsible for negotiating and concluding data contracts. All operations regarding data contracts are run through this service. Finally, the *ServiceHandler* provides an API for accessing the functionality of the current microservices. This makes the whole framework capable of being integrated with other platforms or data hubs.

B. Data Contract Negotiation

Figure 3 presents interactions among different stakeholders and services in order to negotiate and establish a contract. Possible customers can be either persons or software. Providers want to register *Things* in order to be able to sell data. Both providers and customers are interested in negotiating data contracts within a dataspace. Furthermore, once a data contract is concluded, it can be in the interest of both parties to monitor the data flow of the purchased data.

Data contracts can be used to specify contractual terms when purchasing or selling data. Many possible pricing models can exist (e.g., pay per transaction and per data size), however, for the proof of concept we choose a pay per subscription period pricing model (e.g., 10 EUR for the data produced by a *Thing* for one month).

As described in Truong et al. [13] data contracts consist of data contract terms and values. Based on the proposed data contractual clauses in [13], we modeled data contracts to contain a few representative terms as data contractual clauses (see Table I). As proposed in Truong et al. [13] we use term values as single values, a set or a range of values.

Category	**Term(s)**
Data Rights	Derivation, Collection, Reproduction, Commercial Usage
Quality of Data	Completeness, Conformity, Average Message Age, Average Message Currency
Quality of Service	Availability
Pricing Model	Price, Subscription Period
Purchasing Policy	Contract Termination, Shipping, Refund
Control and Relationship	Warranty, Indemnity, Liability, Jurisdiction

TABLE I: Data Contractual Terms

An example of a data contract is provided in Listing 1. In our contract, the contract clauses, e.g., in Table I, are present together with a list of all *Things* IDs, for which the data contract was concluded. Such IDs can be obtained through meta-data of registered *Things*. Furthermore, the QoD and QoS clauses bound to the *Things* of a data contract are associated with QoD/QoS metrics in a separate document.

Listing 1: Data Contract Example

```
{
 ”_id”: ”57f57a770a975a2cc7f52cc5”,
 ”metaInfo”: {
  ”contractId”: ”...”,
  ”creationDate”: ”2016-10-06 00:11:03”,
  ”active”: false,
  ”party1Accepted”: true,
  ”party2Accepted”: true,
  ”revision”: 4,
  ”party1Id”: ”...”,
  ”party2Id”: ”...”
},
 ”dataRights”: {
  ”derivation”: true,
```

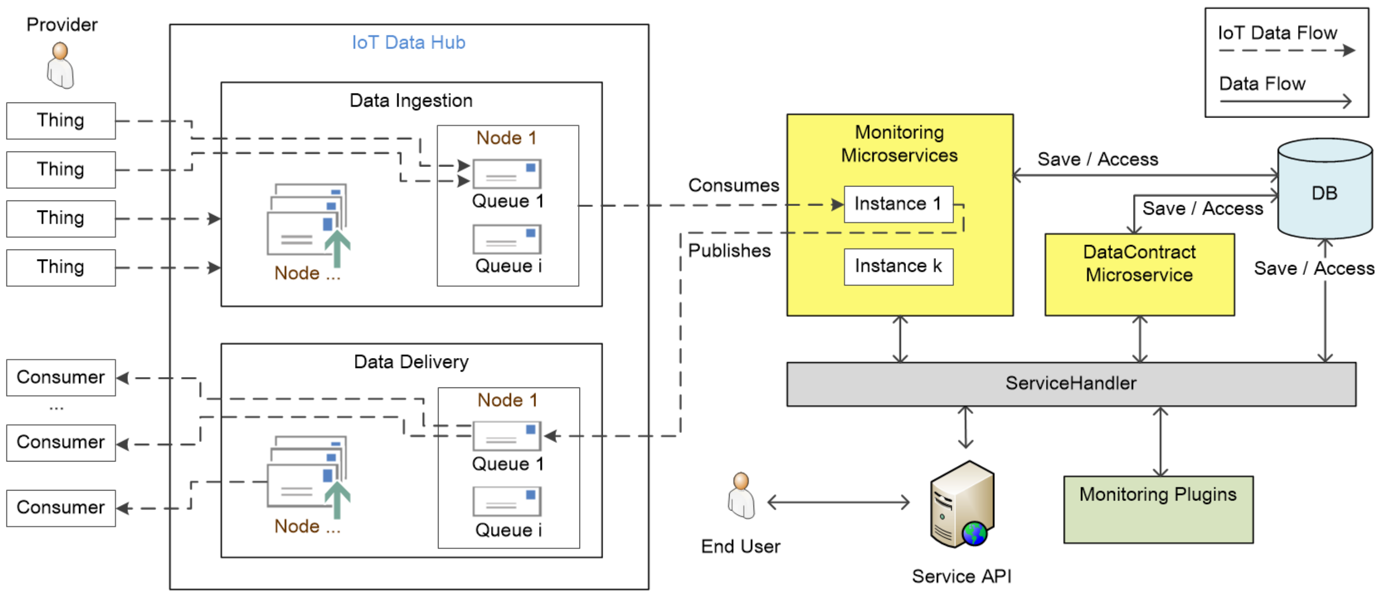


Fig. 2: Architecture of Contract-aware Dataspace Service Platform

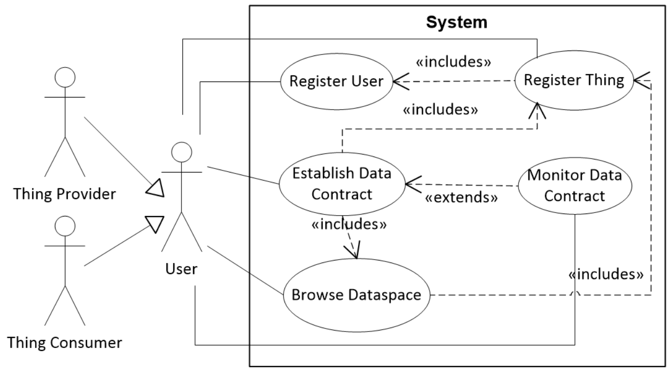


Fig. 3: Interactions in contract establishment

```
  ”collection”: true,
  ”reproduction”: false,
  ”commercialUsage”: false
},
”pricingModel”: {
  ”price”: 3,
  ”currency”: ”EUR”,
  ”transaction”: false,
  ”numberOfTransactions”: null,
  ”subscription”: {
    ”startDate”: ”2016−10−07 00:00:00”,
    ”endDate”: ”2016−10−08 00:00:00” }
},
”controlAndRelationship”: {
  ”warranty”: ”None”,
  ”indemnity”: ”None”,
  ”liability”: ”None”,
  ”jurisdiction”: ”AustriaVienna”
},
”purchasingPolicy”: {
  ”contractTermination”: ”Automatic”,
  ”shipping”: ”Automatic”,
  ”refund”: ”None”
},
 ”thingIds”: [{”thingId”: ”...”,}, ...]
}
```

From these clauses the following terms are negotiable: data rights, pricing model, purchasing policy, control and relationship. Based on these properties individual data contracts can be negotiated and concluded. Every data consumer can propose a data contract for the purchase of data produced by n Things, where the values of the negotiable terms can be freely set. The *Thing* provider receives the proposal and can either accept it immediately or send a counter proposal with modified terms values. This process goes on until both parties accept all terms of the data contract, which represents the point at which the data contract is established.

Once a data contract is established, all relevant data (from all relevant *Things*) is published to all relevant subscribers for the data contract subscription period. All data contracts are stored in the database and can be accessed through an API through the data contract microservice. This way multiple services can access data contracts, even custom made.

C. Data Contract Monitoring

This service is responsible for redistributing all incoming data from all data nodes according to all concluded data contracts and, if enabled, the monitoring of the QoD and the QoS of the data flow. There are two main challenges here: (1) data redistribution, because m data contracts can be concluded for n Things and (2) data monitoring, because n Things can produce q different types of data. In order to deal with these challenges we design *contract-aware operators* for IoT datahubs. We define an operator to be an operation which is always applied to the data stream. For the previously mentioned two challenges we develop two operator types: data operators op_D and monitoring operators op_M. Data operators

– op_D: apply all necessary operations on the data stream to redistribute the data correctly to all subscribers according to all data contracts. Depending on the situation, multiple data operators can be applied for a single contract; thus data operators can be combined into data pipelines. For example we apply a filtering operation op_{D_F} for each *Thing* that is to be monitored to filter out all relevant data contracts. Monitoring operators – op_M: define monitoring operations for streaming data. Monitoring operators can be implemented by plug-ins configured along with data operators. For the data quality assurance we have two categories: $op_M(QoD)$ for the QoD monitoring and $op_M(QoS)$ for the QoS monitoring.

We developed a scalable *Monitoring* microservice as a possible solution for the previously mentioned challenges. To avoid a possible monitoring overhead, the monitoring can be limited to a certain time period (i.e., sample period), however the time period is not explicitly defined and can theoretically range from a few milliseconds to the complete data contract subscription period. During the data contract subscription period, multiple monitoring operations can be performed.

Each instance of the *Monitoring* microservice receives a list (from the *ServiceHandler*) of the message brokers/queues and *Things* it is responsible for monitoring and redistributing the messages. Afterwards a filtering operation is applied, to filter out all relevant data contracts that need data redistribution and might need data monitoring. As discussed previously, one challenge is to always have a valid list of data contracts and of subscribers, because a lot of data can be produced in a short period of time (e.g., hundreds of sensors broadcasting data every 100 ms) and applying a filtering operation, or going to the database for each message, would require too much time and could lead to high data currency values. As a possible solution we implemented a component which retrieves all relevant data contracts and *Things* from the database and caches them into a map. This map is refreshed periodically (e.g., every 60 seconds) or as soon as data contracts expire or new ones are concluded. As the received messages are consumed, they are first redirected to all relevant subscribers and afterwards, if the current time is within the sample monitoring time QoD and QoS metrics can be computed. Our proposed solution for the data monitoring is illustrated in algorithm 1 ($op_M(QoD)$ is the operation for the meta-model traversal and QoD computation and $op_M(QoS)$ is the operation for the QoS computation).

The structure of the messages, which are produced by a *Thing*, is described using a meta-model. For the sake of simplicity and as a proof of concept, we provided a simple simple service and GUI to enable the specification and management of the meta-model of a *Thing* during the registration of *Things*, so that the *Monitoring* microservice, can deal with any JSON data, as long as the meta-model is known. However, as a possible extension for a custom component, it could be an option to retrieve the meta-model or exchange *Things* information with other platforms, e.g., we are currently integrating HINC [20].

Our meta-model specification consists of a list of attributes.

Algorithm 1 Algorithm for monitoring data contracts

```
Input: thingsList, nrOfConsumers, allDataContracts
Output: void
    LOOP thingsList
 1: for $thing$ in thingsList do
 2:     $dataContractList = op_{D_F}(thing)$
        LOOP nrOfConsumers
 3:     for $consumer$ in nrOfConsumers do
 4:         start $consumer$ and consume $message$ from $thing.queue$
 5:         $messageReceivalTime := currentTime;$
 6:         deliver $message$ to subscriber
 7:         if ($currentTime$ in $sampleMonitoringTime$) then
 8:             $messageDeliveryTime := currentTime;$
 9:             $op_M(QoD)(thing.metaModel, message);$
10:             $op_M(QoS)(thing, messageReceivalTime);$
11:         end if
12:     end for
13: end for
14: return
```

Each attribute has a certain type, which might be primitive (e.g., integer, boolean) or not (e.g., attribute containing an object) and a certain value (e.g., 1 for integer, or another attribute as the instance of an object). Having all this information, the quality of data of a message can be computed. Algorithm 2 shows how the *completeness* and the *conformity* of a message can be computed based on the meta-model of a message.

Algorithm 2 Algorithm for traversing the meta-model and computing QoD - op_{m1}

```
Input: $metaModel, message$
Output: $completeness, conformity$
    $completeness$ := true; $conformity$ := true;
    {loop all metaModel attributes}
 1: for $attribute$ in $metaModel$ do
 2:     if ($message$ not contains $attribute.name$) then
 3:         $completeness$ := false;
 4:         continue;
 5:     else
 6:         $cAttr := message.attribute$
 7:         if ($cAttr.dataType$ != $attribute.dataType$) then
 8:             $conformity$:= false;
 9:         end if
10:     end if
11:     if ($attribute.dataType$ is not primitive Type) then
12:         $traverseMetaModel(attribute.metaModel, message.attribute);$
13:     end if
14: end for
15: return
```

If the expected frequency of the produced messages of a device is known, then the availability can be computed based on that frequency and based on the message receival time. One possible solution for the computation of the availability is illustrated in Algorithm 3 (where f_e = expected frequency, t_{fmr} = first message receival time, t_{mr} = message receival time, m_e = number of expected messages, m_t = number of total received messages).

Algorithm 3 QoS computing algorithm - op_{m2}

```
Input: $thing$, $t_{mr}$
Output: $thing.monitoredqos$
    availability := 0;
 1: $f_e$ := $thing.f_e$;
 2: if ($thing.monitoredqos$ != $NULL$) then
 3:     $t_{fmr}$ := $thing.monitoredqos.t_{fmr}$;
 4:     $\Delta t$ := $t_{mr}$ - $t_{fmr}$; $m_e$ := $\Delta t$ / $f_e$; $m_e$++;
 5:     $thing.monitoredqos.m_e$ := $m_e$;
 6:     $thing.monitoredqos.m_t$++;
 7:     $thing.monitoredqos.availability$ := $m_t$ / $m_e$;
 8: else
 9:     $thing.monitoredqos.t_{fmr}$ := $t_{mr}$;
10:     $thing.monitoredqos.m_e$ := 1;
11:     $thing.monitoredqos.m_t$ := 1;
12:     $thing.monitoredqos.availability$ := 1;
13: end if
14: return
```

D. Contract Monitoring Trail

As presented in the previous subsection, the data flow, produced by *Things*, can be monitored for a certain period of time (sampling). This approach (monitoring for a certain time period) has been chosen in order to be able to supply quality metrics while data contracts are still running and the subscription period is not over. During the subscription period of a data contract, multiple monitoring periods can exist, which leaves a contract monitoring trail. These monitored values are grouped together and stored in the database. The contract monitoring trail can be accessed any time through the API of the *Service Handler*, even while computations are still running.

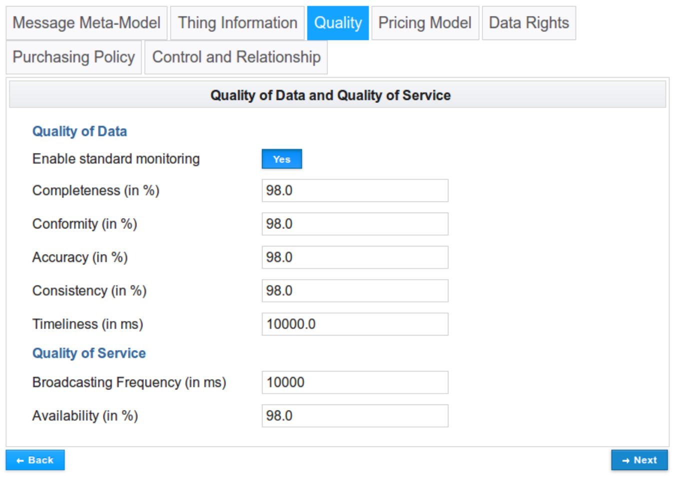
IV. Experiments

A. Prototype

We have implemented a prototype as a proof-of-concept for the previously proposed architecture which can run on its own and act as an IoT data hub[2]. We use MongoDB for storing contracts as documents. We developed the ServiceHandler and the microservices using Apache Camel. We choose Apache Camel due to its support for multiple enterprise integration patterns and the possibility of using and integrating different technologies. All microservices and the ServiceHandler run on an integrated Jetty server. The microservices and the ServiceHandler communicate with each other via REST services. As a queue we choose Apache ActiveMQ.

[2]the prototype is available at: https://github.com/e0725439/idac

B. Illustrative Examples



Quality of Data and Quality of Service

Quality of Data

Enable standard monitoring	Yes
Completeness (in %)	98.0
Conformity (in %)	98.0
Accuracy (in %)	98.0
Consistency (in %)	98.0
Timeliness (in ms)	10000.0

Quality of Service

Broadcasting Frequency (in ms)	10000
Availability (in %)	98.0

Fig. 4: Example of defining data contracts

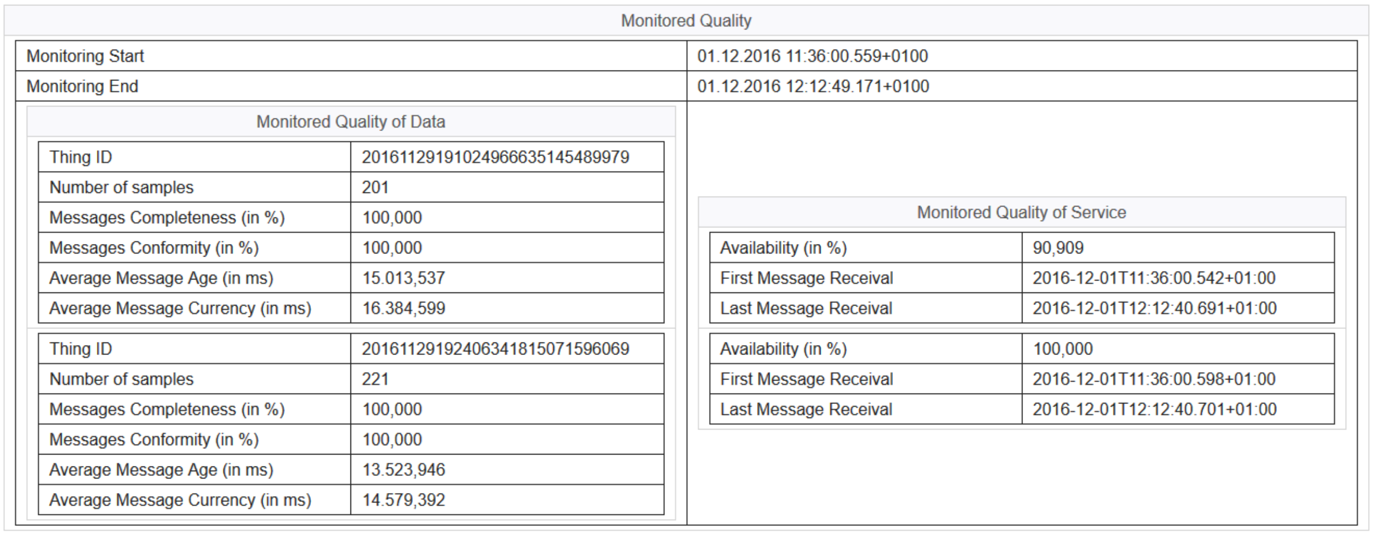
We emulated data transmissions from one dataspace consisting of 3 *Things* of three different types: (1) temperature sensor DS18B20, (2) water quality sampling data downloaded from [23], and (3) mobile device measurements obtained with [24]. In order to simulate continuous data transmissions, we sent the data to a message broker every 10 seconds.

We emulated several contracts for *Things*; each includes basic information: device ID, the meta-model of the produced messages, the broadcasting frequency, quality of data attributes (e.g., completeness and conformity) and quality of service attributes (e.g., availability). We applied two negotiation steps in each case: one data contract proposal, one counter proposal and one acceptance.

Figure 4, presents an example of defining contracts with different terms for different metrics; these terms are associated with *Things* via meta-information about *Things*. Figure 5 presents the dashboard through which one can observe contract monitoring. In this example a data contract was monitored, which consisted of data produced by two temperature sensors. Both *Things* had the expected broadcasting frequency of 10 seconds, but the devices were configured to broadcast data every 10 resp. 11 seconds, which resulted in an availability of 100% resp. 90%. In practice for large-scale IoT dataspaces, violations detected through contract monitoring data will be communicated to the consumers through APIs and/or saved into trails that can be retrieved.

From the dashboard and API, contracts can be also rated and we also provide features to recommend possible contracts given a new data from Things (the contract recommendation is out of scope of this paper). This demonstrates the extensibility of our framework by custom components.

These illustrative examples show that our prototype can serve as a possible solution to overcome the described problem and its challenges from Section I. We successfully negotiated and established individual data contracts, consisting of different data contractual clauses for the purchase of data



Monitored Quality

Monitoring Start	01.12.2016 11:36:00.559+0100
Monitoring End	01.12.2016 12:12:49.171+0100

Monitored Quality of Data

Thing ID	20161129191024966635145489979
Number of samples	201
Messages Completeness (in %)	100,000
Messages Conformity (in %)	100,000
Average Message Age (in ms)	15.013,537
Average Message Currency (in ms)	16.384,599

Thing ID	20161129192406341815071596069
Number of samples	221
Messages Completeness (in %)	100,000
Messages Conformity (in %)	100,000
Average Message Age (in ms)	13.523,946
Average Message Currency (in ms)	14.579,392

Monitored Quality of Service

Availability (in %)	90,909
First Message Receival	2016-12-01T11:36:00.542+01:00
Last Message Receival	2016-12-01T12:12:40.691+01:00

Availability (in %)	100,000
First Message Receival	2016-12-01T11:36:00.598+01:00
Last Message Receival	2016-12-01T12:12:40.701+01:00

Fig. 5: Data contract monitoring dashboard

produced by *Things*. Furthermore, once the data contracts were established, we were able to successfully monitor the data stream for each concluded contract. Finally, we also extended our framework by a custom recommendation component.

C. Performance Analysis

In this section we are going to evaluate different performance aspects of our prototype to demonstrate its reliability and stability. The prototype was completely deployed and evaluated in two environments: (1) a virtual server with the following properties: CentOS 7.2, Kernel: 64-bit, Intel(R) Xeon(R) CPU E5-2620 @2.00GHz, 2 cores, 8 GB RAM and (2) physical machine with Win. 10 x64, CPU: Intel(R) Core(TM) i3 CPU M370 @2.4GHz, 8 GB RAM. All system settings were chosen to demonstrate the performance and the reliability of the prototype, even when running within limited environments.

The first set of performance tests were conducted to evaluate the average computation time for the QoD and QoS for 100 data contracts, each consisting of one *Thing*, for 100 respectively 1000 messages produced by the three types of *Things* mentioned in the previous section. These performance tests run in environment (2) and were executed by calling the computation component directly (i.e., no messages were read/sent from/to any message broker).

The experiments properties and results are as follows (where t is the *Thing* type - see Section IV-B, n is the number of *Things*, m the number of produced messages per *Thing*, f the broadcasting frequency in *ms* and t_{avg} is the average computation time in *ms* of the QoD and QoS metrics for one single message):

- $t = 1, n = 100, f = 100, m = 100, t_{avg} = 1.46$
- $t = 1, n = 100, f = 100, m = 1000, t_{avg} = 1.07$
- $t = 2, n = 100, f = 100, m = 100, t_{avg} = 1.38$
- $t = 2, n = 100, f = 100, m = 1000, t_{avg} = 1.01$
- $t = 3, n = 100, f = 100, m = 100, t_{avg} = 1.77$
- $t = 3, n = 100, f = 100, m = 1000, t_{avg} = 1.20s$

This results demonstrate that the computation of QoD and QoS metrics is not a time consuming task.

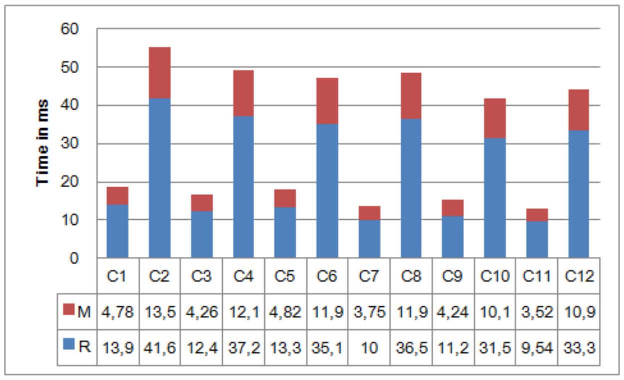
To further evaluate the performance of the monitoring component and the caused monitoring overhead in addition to simple message redistribution we conducted 12 further performance tests in environment (2) simulating real data transmissions and data contracts. For each of the previously mentioned *Thing* types 4 tests were conducted. In each of these 4 cases either 10 or 20 *Things* broadcasted data to either one or two message queues. In all cases each *Thing* was bound to a different data contract and for all *Things* a minimum number of 1000 messages were consumed. The results are shown in Figure 6 (where R stands for the time in milliseconds needed to read a message and redistribute it and M stands for the time used for the computation of the quality): the monitoring overhead ranges from approx. 32% to approx. 38%.

The fact that all messages are first redistributed and afterwards, if enabled, the quality is computed, and the fact that all operations are in the milliseconds range (all below 60) proves that data stream monitoring is negligible when dealing with no real-time data transmissions.

We conducted further performance tests in environment (1) to evaluate the behavior and performance of the *DataContract* microservice. All tests have been run using Apache JMeter, which enables multiple parallel calls to a single REST service and monitors automatically the following metrics: average, minimum and maximum response time, error percentage and throughput (i.e., number of successfully processed requests).

A negotiation process was simulated by each thread. This process consisted of 5 negotiation steps: 2 data contract offers, 2 counter-offers and 1 data contract establishment. All used data was generated by simulating and logging a few representative scenarios with the GUI, taking the respective data (JSON objects) and changing the IDs of the objects.

The results of the performance tests are shown in Table II.



	C1	C2	C3	C4	C5	C6	C7	C8	C9	C10	C11	C12
🟥 M	4,78	13,5	4,26	12,1	4,82	11,9	3,75	11,9	4,24	10,1	3,52	10,9
🟦 R	13,9	41,6	12,4	37,2	13,3	35,1	10	36,5	11,2	31,5	9,54	33,3

Fig. 6: Monitoring overhead

We run 4 sets of performance tests consisting of 10, 20, 40 and 80 parallel threads. In total 500, 1000, 2000 and 4000 requests were sent and processed successfully error free. As we can see, especially from the results of the last performance test, all recorded metrics indicate a stable system behavior, even when running within a limited environment with a high amount of requests. Although we have not executed our tests with many computing resources, the above results demonstrates the stability and performance of our prototype implementation even when running within a limited environment with different types of data contracts and different types of *Things*.

Test Nr.	**1**	**2**	**3**	**4**
Threads	10	20	40	80
Ramp-Up Period (s)	10	10	10	10
Loop Count	10	10	10	10
Average (ms)	59	52	62	106
Min (ms)	36	36	37	43
Max (ms)	223	122	393	815
Standard Deviation	25.06	11.80	19.99	57.30
Error (%)	0.00	0.00	0.00	0.00
Throughput	41.7	82.5	156.8	269.1

TABLE II: Performance of data contract negotiation

V. Conclusions and Future Work

In this paper we have introduced a contract-aware IoT framework, which supports data contract management and monitoring for IoT dataspaces. Our platform is designed using the microservice model to deal with different scales of IoT dataspaces. Although we have not integrated our framework with existing IoT data hubs, we tested our framework with emulated sensors using real-world datasets and presented the usefulness and the performance of our frameworks for continuous QoD and QoS monitoring in IoT contracts.

Our future work is to integrate our framework with real-world IoT hubs, like from Azure and Amazon. One important aspect is to integrate with *Things* service providers to obtain meta-data about *Things*, such as using HINC [20]. Furthermore, we plan to utilize existing stacks, such as Logstash, Streamsets, and Influx to implement *contract-aware operators*, enabling testing of our framework in a very large-scale IoT environment.

References

[1] Array of Things, "Array of Things," 2016, [Online]. Available: https://arrayofthings.github.io/. [Accessed: Sep. 28, 2016].
[2] O. Boris, J. Jan, S. Jochen, A. Sören, M. Nadja, W. Sven, and C. Jan, "Industrial Data Space," Fraunhofer-Gesellschaft zur Förderung der angewandten Forschung e.V., 2016, [Online]. Available: www.fraunhofer.de/content/dam/zv/de/Forschungsfelder/industrial-data-space/Industrial-Data-Space_whitepaper.pdf. [Accessed: Sep. 28, 2016].
[3] Tilepay, "tilepay," 2016, [Online]. Available: http://www.tilepay.org. [Accessed: Feb. 16, 2016].
[4] T. Cao, T. V. Pham, Q. H. Vu, H. L. Truong, D. Le, and S. Dustdar, "MARSA: A marketplace for realtime human sensing data," *ACM Trans. Internet Techn.*, vol. 16, no. 3, p. 16, 2016.
[5] BDEX, "Bdex, the marketplace for data," 2016, [Online]. Available: http://www.bigdataexchange.com. [Accessed: Mar. 21, 2016].
[6] Microsoft, *Microsoft Azure Marketplace*, 2016, [Online]. Available: https://azure.microsoft.com/en-us/marketplace/. [Accessed: May 21, 2016].
[7] I. Amazon Web Services, *Amazon Web Services*, 2016, [Online]. Available: https://aws.amazon.com/. [Accessed: Sep. 1, 2016].
[8] G. D. LLC, *Predix*, 2016, [Online]. Available: https://www.predix.io/. [Accessed: Oct. 11, 2016].
[9] M. Franklin, A. Halevy, and D. Maier, "From databases to dataspaces: A new abstraction for information management," *SIGMOD Rec.*, vol. 34, no. 4, pp. 27–33, Dec. 2005.
[10] I. S. S. z o. o., *DeviceHub.net*, IOT Solutions Sp. z o. o., 2016, [Online]. Available: https://devicehub.net/. [Accessed: Oct. 14, 2016].
[11] IBM, *Internet of Things - Big Data & Analytics Hub*, IBM, 2016, [Online]. Available: http://www.ibmbigdatahub.com/technology/internet-of-things. [Accessed: Oct. 14, 2016].
[12] Equinix, "Equinix data hub," Equinix, Inc, 2016, [Online]. Available: http://www.equinix.com/services/data-hub/. [Accessed: Oct. 14, 2016].
[13] H.-L. Truong, M. Comerio, F. D. Paoli, G. R. Gangadharan, and S. Dustdar, "Data contracts for cloud-based data marketplaces," *Int. J. Comput. Sci. Eng.*, vol. 7, no. 4, pp. 280–295, Oct. 2012.
[14] C. Batini and M. Scannapieco, *Data Quality: Concepts, Methodologies and Techniques (Data-Centric Systems and Applications)*. Springer, 2006.
[15] W. W. Eckerson, "Data quality and the bottom line," [Online]. Available: http://download.101com.com/pub/tdwi/Files/DQReport.pdf. [Accessed: Jul. 22, 2016].
[16] R. Y. Wang and D. M. Strong, "Beyond accuracy: What data quality means to data consumers," *Journal of Management Information Systems*, vol. 12, no. 4, pp. 5–33, 1996.
[17] K.-U. Sattler, *Data Quality Dimensions*, Boston, MA, 2009.
[18] A. Klein and W. Lehner, "Representing data quality in sensor data streaming environments," *J. Data and Information Quality*, vol. 1, no. 2, pp. 10:1–10:28, Sep. 2009. [Online]. Available: http://doi.acm.org/10.1145/1577840.1577845
[19] S. Systems, "Data quality glossary – conformity," May 2011, [Online]. Available: http://www.dqglossary.com/conformity.html. [Accessed: Jul. 24, 2016].
[20] D. Le, N. C. Narendra, and H. L. Truong, "HINC - harmonizing diverse resource information across iot, network functions, and clouds," in *4th IEEE International Conference on Future Internet of Things and Cloud, FiCloud 2016, Vienna, Austria, August 22-24, 2016*, M. Younas, I. Awan, and W. Seah, Eds. IEEE Computer Society, 2016, pp. 317–324.
[21] S. Geisler, C. Quix, S. Weber, and M. Jarke, "Ontology-based data quality management for data streams," *J. Data and Information Quality*, vol. 7, no. 4, pp. 18:1–18:34, Oct. 2016. [Online]. Available: http://doi.acm.org/10.1145/2968332
[22] P. Barnaghi, M. Bermudez-Edo, and R. Tönjes, "Challenges for quality of data in smart cities," *J. Data and Information Quality*, vol. 6, no. 2-3, pp. 6:1–6:4, Jun. 2015. [Online]. Available: http://doi.acm.org/10.1145/2747881
[23] City of Austin open data, "Water Quality Sampling Data," City of Austin, 2016, [Online]. Available: https://data.austintexas.gov/Environmental/Water-Quality-Sampling-Data/5tye-7ray. [Accessed: Oct. 14, 2016].
[24] OpenSignal, "OpenSignal," OpenSignal, 2016, [Online]. Available: https://opensignal.com/. [Accessed: Oct. 14, 2016].