Collecting data to FIWARE

Otto Hylli, Ville Heikkilä and Kari Systä

May 29, 2020

Contents

1 Introduction

The goal of the CityIoT[1] project is to define a vendor independent IoT platform for SmartCity applications. The platform should support multiple data sources and several applications using the data sources. Consequently, the data from different sources should be unified for efficient usage by applications. In the beginning of the project we analyzed technical options and selected FIWARE[2] as the technical framework, because it has similar goals towards vendor independence and has already gained interest in the SmartCity community.

In this report we summarize our experiences in collecting, unification and storing the data. The research goals of this work were to investigate applicability of FIWARE for multivendor cases in the context of Smart Cities, and to provide information for developers and researchers with similar challenges to our. The scope of this study has been limited to the data collection, preprocessing, unification and storing of the data. For example, access control and deployment of a FIWARE platform are out of the scope for this report.

In the CityIoT project we have created several pilots whose data was collected from several sources and stored in the FIWARE-based platform. This report collects experiences from these pilots and its goal is to give a general view what is it like to collect data to FIWARE: what are the advantages and disadvantages.

The research process was the following. In the beginning of the project we iteratively created the platform and conducted pilot cases with external stakeholders. The pilot cases were driven by concrete IoT pilots of cities of Tampere and Oulu. The role of our research team varied in the pilot cases: in some we implemented the whole data collection process and in some we just advised the company implementing the FIWARE integration. After implementing the 12 pilot cases we collected the experiences from the implementors. We then categorized the experiences into four broad categories. These categories cover the main aspects of collecting data: datamodels, data conversion, sending the data, and data storage and handling.

The rest of the document has been organized as follows. Section 2 gives an introduction to the FIWARE technology including the data modeling principles and the important FIWARE components. Section 3 describes the FIWARE platform instances used in the project and gives an overview of the pilot cases. The experiences, i.e., the main results are given in Section 4 where each aforementioned experience category has its own subsection. Finally, Section 5 gives our conclusions.

2 FIWARE technology

This section provides background information about FIWARE since basic knowledge about FIWARE is required in understanding the experiences reported later.

[1]`https://www.cityiot.fi`
[2]`https://www.fiware.org`

This will not be an in-depth overview of FIWARE. We will introduce the FIWARE data modeling principles and describe shortly the relevant FIWARE components used in CityIoT. The description is mainly limited to concepts which are relevant from the data collecting perspective.

2.1 Data models

The FIWARE data model is specified in the NGSI V2 specification. In FIWARE data is managed as context entities. Entities can represent various physical or logical objects such as devices, vehicles, weather observations and buildings. There is also a newer version of the data model specified in the NGSI-LD specification which is based on linked data. The new data model was not used in the pilots since work on the specification and components that implement the specification started during the CityIoT project and as of this writing none of the components is fully complete. However there is another CityIoT report that explores NGSI-LD.

Each NGSI-v2 entity has an id and type which together uniquely identifies the entity. The entity type then defines the attributes of an entity. An entity attribute has a name, a value and a type. The type can be a primitive type such as Text or Number. It can also represent more complex data structures such as a street address or a location defined with latitude and longitude coordinates. An attribute can also represent a relationship between entities in which case its value is the id of the related entity. FIWARE generates two build-in attributes for each entity: for the entity creation time (dateCreated) and last modification time (dateModified).

Attributes can also have related metadata which consists of metadata attributes that have a name, a value and a type. A metadata attribute can for example represent the accuracy of a measurement. A very common metadata attribute is a timestamp representing the time the attribute value was measured. There are also two build-in metadata attributes: dateModified and dateCreated, which work similarly to the corresponding entity level build-in attributes except they work on the attribute level. A conceptual schema of the NGSI-v2 data model is shown in figure 1.

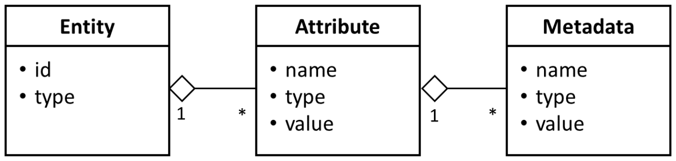

Figure 1: Conceptual schema for NGSIv2 data model.

A FIWARE data model for a certain domain is defined by specifying the required entity types and their attributes which includes attribute names and types. Attributes can also be marked as mandatory or optional. In FIWARE the entity data is represented as JSON and the data model is documented by defining a JSON schema for the entity types and writing a human readable markdown document describing the entities and their attributes. Existing JSON schema property definitions can be utilized as part of the specification. For example the aforementioned street address and location structures are defined elsewhere. The FIWARE community has defined data models for various domains such as weather, parking and street lighting.

Listing 1 shows an example entity of type *ThreePhaseAcMeasurement*.[3] This type is used to represent measurements from an electrical system that uses three phase alternating current. The source of the measurement could be for example a smart electricity meter. The example does not show the whole entity; instead it shows examples of different kinds of attributes.

The first lines define the compulsory entity id and type and the rest are entity type specific attributes. The first entity-specific attribute *dateEnergyMeteringStarted* is of the type *DateTime* and it represents the time for the beginning of energy measuring. The second attribute *refDevice* is a relationship to another entity of the type Device which represents the device making the measurements. The next attribute *name* has a name for the measurement. The second to last attribute *totalActiveEnergyImport* has the total amount of energy since the measuring started. It is of the type *Number*. It also has the time the measurement was made in the *timestamp* metadata attribute. The last example attribute *activePower* is of the type *StructuredValue* which indicates that it consists of multiple values. The values for *l1*, *l2* and *l3* contain the measured power for the three phases. In addition to the measurement time, the metadata provides additional information about the measurement. The *measurementType* and *MeasurementInterval* metadata attributes tell that the measurement is the average power measured during one second.

2.2 Components of FIWARE

The FIWARE project has developed multiple components for various purposes such as big data processing and access control. These components can be used to form a platform according to the platform provider's needs. Here we introduce the components relevant for this work: the Orion context broker, IoT agent for ultralight 2.0 and two alternatives for entity time series history components: STH comet and QuantumLeap.

2.2.1 Orion

The essential and core component of FIWARE is the Orion context broker. It is the basis of any FIWARE powered platform. It manages information about the current context, i.e., the state of the entities. Orion knows only the current state of each entity so when an entity is updated with new information the old information is no longer available from Orion.

[3]This data model was created in the CityIoT project and was then accepted as an official FIWARE data model.

Listing 1: Data model example showing an incomplete ThreePhaseAcMeasurement entity.

```
{
  "id": "ThreePhaseAcMeasurement:LV3_Ventilation",
  "type": "ThreePhaseAcMeasurement",
  "dateEnergyMeteringStarted": {
    "type": "DateTime",
    "value": "2018-07-07T15:05:59.408Z"
  },
  "refDevice": {
    "type": "Relationship",
    "value": ["Device:eQL-EDF3GL-2006201705"]
  },
  "name": {
    "type": "Text",
    "value": "ventilation"
  },
  "totalActiveEnergyImport": {
    "metadata": {
      "timestamp": {
        "type": "DateTime",
        "value": "2019-01-24T22:00:00.173Z"
      }
    },
    "type": "Number",
    "value": 150781.96448
  },
  "activePower": {
    "metadata": {
      "timestamp": {
        "type": "DateTime",
        "value": "2019-01-24T22:00:00.173Z"
      },
      "measurementType": {
        "value": "average"
      },
      "measurementInterval": {
        "value": 1
      }
    },
    "type": "StructuredValue",
    "value": {
      "L1": 11996.416016,
      "L2": 9461.501953,
      "L3": 10242.351562
    }
  }
}
```

Orion implements the NGSI v2 API specification which defines operations for creating, modifying, querying and deleting entities. NGSI is a RESTful API accessed with HTTP protocol. For example to create the entity shown in listing 1 a HTTP post request should be sent to Orion's entities API endpoint (/v2/entities) with the JSON in the listing as the payload.

Though not specified in the NGSI V2 specification Orion allows entities to be organized by separating them under different FIWARE *services*. The target service is specified with an HTTP-header (Fiware-Service). A single API request can then affect only entities of the given service. For example, a query for entities can only return entities of the given service. Also, two entities under different services can even share the same id and type with no problems. Services can be used together with FIWARE or third party authentication and authorization services to give an user access only to entities in specific services. Inside a service hierarchies of the data can be expressed with FIWARE *service paths* which also are not part of the NGSI v2 specification. This is a fundamental difference to typical RESTful APIs where the resource hierarchies are encoded in the URL-paths. Technically the service path is also given as HTTP headers (Fiware-ServicePath).

In addition to queries FIWARE supports publish-subscribe paradigm where clients can request HTTP notifications on any changes of specified entities. In the subscription request, the user can specify what kind of change should trigger the notification and what information is included in the notification. Many FIWARE components, for example STH Comet and QuantumLeap described below, integrate to Orion with this subscription mechanism. Orion itself can also receive subscriptions allowing the linking of different FIWARE platform instances.

2.2.2 IoT agent for Ultralight 2.0

Many IoT devices use various protocols to communicate and thus cannot be directly connected to Orion which requires the use of HTTP and the NGSI-v2 API. For this purpose FIWARE has the concept of an IoT agent which works as a bridge between the IoT device and Orion. The agent translates between the protocol used by the IoT device and the NGSI-v2 API and data model.

One such agent is the IoT agent for Ultralight 2.0 which translates for the Ultralight 2.0 protocol developed in the FIWARE community. It is a light weight text based protocol meant for resource constrained devices. Format for reporting measurements is a list of measurement name value pairs separated by the `|` character. For example `temp|20|hum|30` has the value 20 for temp (temperature) and 30 for hum (humidity). A timestamp can be added to the message which will be used as the entity attribute metadata timestamp. If it is not provided the current time is used. Ultralight 2.0 supports multiple protocols for transporting the messages: HTTP, MQTT and AMQP. For configuration of the agent including adding devices and mapping their measurements to entity attributes, the agent implements the common FIWARE IoT agent provision API.

2.2.3 STH Comet

The STH (short time historic Comet component allows entity history to be stored and queried. It stores the entity attribute values as raw time series data and also calculates aggregates from the data. The data is stored into a MongoDb database. The data can be queried via the STH Comet API. No tools for further analysis or visualization are available.

STH Comet uses the Orion subscription system to get entity data. In order to use STH Comet a suitable Orion subscription has to be created. A notable restriction of STH Comet is the lack of support for the newer FIWARE entity model defined in the NGSI v2 specification. STH Comet supports only the earlier version of the specification.

2.2.4 QuantumLeap

QuantumLeap is an alternative for STH Comet. It can also store entity history as time series data. It allows both the raw data to be queried and can also calculate aggregates from that data. Like STH Comet it also gets the entity data via Orion subscription notifications.

Unlike STH Comet, QuantumLeap is meant to support different backends for storing the data. The idea is also to use databases specifically intended for time series data. However, currently only one database CrateDB is fully supported. Measurements are saved to entity type specific database tables where each row has a timestamp and values for attributes. If the attribute does not have a value for that time its value is null.

Data can be queried via the QuantumLeap API and like STH Comet no tools for further analysis are available. The documentation of QuantumLeap however instructs how the open source, browser based Grafana data visualization tool can be used with the data QuantumLeap has stored. Grafana is not specific to FIWARE but it can be used with the data by directly accessing the backend database.

3 Use of FIWARE in CityIoT

This section explains how and for what FIWARE was used in the CityIoT project. First the FIWARE platform instantiations used in our data collection cases are described. Then the data collection cases, the experiences reported later are based on, are described.

3.1 Platform instances

Two FIWARE platforms were deployed during the CityIoT project: one at Tampere University and one at University of Oulu. The platforms also evolved during the project. Both platforms used one virtual machine and the FIWARE components and other related services such as databases were ran inside Docker containers. Both platforms used simple HTTP header-based API tokens for authentication and FIWARE services for separating the data of different cases. The most notable differences between the two platforms are the use of the ultralight IoT agent in Oulu and use of a different history component. The Tampere platform used QuantumLeap and Oulu platform used STH comet. The CityIoT Report on FIWARE Platform document describes our FIWARE deployment in more detail.

3.2 Data collection cases

During the CityIoT project various smart city-related IoT pilots were launched by city of Tampere and city of Oulu. In Oulu all of the pilots were integrated to University of Oulu's FIWARE platform by the companies implementing the pilots. In Tampere four pilots were integrated to Tampere University FIWARE platform by the researchers of the university. Overall 12 pilots were integrated to FIWARE. The integration included data collection from various domains such as street lighting, electricity monitoring and indoor condition monitoring. These various data collection cases are shortly introduced here.

3.2.1 Tampere cases

Street lights Electricity consumption and ambient illuminance data from the street light system of Tampere were collected once per day to an Azure SQL database by the city of Tampere. For accessing the data, an HTTP-based API implemented with tools offered by Azure was provided to the university researchers. This API was then used to transfer the data to FIWARE. The data includes voltage, current and illuminance measurements from 317 groups of street lights that overall contain about 40 000 street lights. This case and experiences from it are further described in the Transferring streetlight data to FIWARE - lessons learned report.

Smart street lights 400 smart street lights were installed to one region in Tampere. Each of these individual street lights report their electricity measurements, ambient illuminance measurements and the angle of the lamp pole. These measurements are reported to a FIWARE platform operated by the case implementer. Through the Orion subscription system the measurements are passed to the Tampere university FIWARE platform.

Electric bus Real time measurements from four electric bus and one hybrid bus were collected to a commercial IoT platform[4]. By using the API of the platform the measurements were transferred to FIWARE. The measurements included speed of the bus, battery power, battery charge, energy consumption and door status. Overall, 18 measurements were available though not all buses had measurements for all of them. Some of the measurements were updated every second and some every few seconds. This case and experiences from it are further described in the Converting and transferring data from another IoT platform to FIWARE: case Electric bus report.

Bus passenger analytics Bus passenger analytics data was collected from one bus by using a 3d character recognition sensor. This data was sent and analyzed in the service provider's system which offered a rest API for accessing the data. This API was used to transfer part of the data to FIWARE. The data included daily passenger amounts and bus stop statistics about passengers picked up and dropped off on the stop among others. Data was available from about three months though there were some gaps due to technical difficulties. This case and experiences from it are further described in the Storing Bus Passenger Analytics Data into FIWARE report.

[4] `https://iot-ticket.com`

3.2.2 Oulu

Lighting maintenance Internet connected electricity metering devices were installed to one building for detecting burned out lamps. The power consumed by lighting was monitored and decreases in that could then be used to detect that some lamps have broken down. Alerts about the number of broken lamps could then be sent to the building maintenance. The implementer integrated their system to FIWARE by sending the alerts and electricity measurements including power, voltage, current and frequency to FIWARE. The measurements were collected and sent every 10 minutes.

Water consumption Water consumption from a few city of Oulu buildings was monitored by reading old non-internet connected water meters with machine vision. This information was also used to detect water leaks. This system was integrated to FIWARE by sending the water consumption information and the water leak alerts to FIWARE.

Building visitor counting Sensors for counting the visitors were installed to one public city building in Oulu. Visitor count information was then also sent to FIWARE.

Building air pressure Hardware for monitoring the difference in air pressure between indoors and outdoors was installed to few buildings in Oulu. This information among other measured air quality information helps in building maintenance. These measurements were also sent to FIWARE.

Preventing vandalism A camera was installed outside a school for preventing loitering and vandalism. The camera image was analyzed automatically to detect the number of people. Based on the number of people and time of day alerts for unusual activity were sent to school administration and security personnel. These alerts were also sent to FIWARE.

Outdoor lighting The company implementing the pilot has developed hardware, that can be connected to different building automation systems, which unifies the available data and system control for use in the company's web based platform. This system was used in two buildings to monitor and control outdoor lighting. Two way communication with the system was implemented to FIWARE where the lighting could be monitored and controlled via FIWARE. The IoT agent was used in sending the data in this case.

Indoor conditions Air quality (e.g. carbon dioxide, volatile organic compounds) and noise level measurements were collected from one school building in order to make them visible and easily understandable for the users of the building. Subjective feedback about the conditions was also collected from the users. The condition measurements were also sent to FIWARE via the IoT agent.

Indoor conditions 2 Another company collected air quality and noise level measurements from a different school building. These measurements were also sent to FIWARE.

4 Experiences

This section describes the experiences in data collection encountered in the pilot cases. The experiences have been divided in to four broad categories: data models, data conversion, sending the data, and data processing and storage.

4.1 Data models

This section describes how the FIWARE entity based data models fit to our various use cases. It also describes how well we were able to utilize the existing data models: sometimes we could use a model as is, sometimes we had to adapt the models by adding attributes and sometimes we had to create new entity types.

Table 1 gives an overview of data model usage in the pilot cases. For each case it lists the entity types used, indicates if we had to add custom attributes to one or more of the existing entities or if we had to create new entity types ourselves. The new entity types are in bold and marked with the * character in the entity types list.

The table shows that the existing FIWARE data models were quite useful. Only in the *Building visitor counting* case no existing data model was used. Three other cases also required new entity types though they also utilized existing types. On the other hand only in 3 cases the existing data models did not need any modifications. In 6 cases some additional attributes were required. It has to be also noted, that in some cases the data produced by a pilot could not be considered fully valid according to the data model since there was not always data available for mandatory attributes of the entity.

Based on these experiences it can be said that existing FIWARE data models can be useful but they often do not fulfill all needs. Also the possibility exists that a FIWARE user has to design a new data model. Fortunately, the data model documentation offers some guidelines[5]. Also even though an existing data model could be adapted for the source data in some cases it might still be better to create a new data model. This might have been better in the *bus passenger analytics* case where we used the existing *UrbanMobility* data model. Making a new data model based on the source data might have been easier than understanding the existing model, which was not meant for this exact purpose, and attempting to modify it for the source data. The model was meant for

[5]`https://github.com/smart-data-models/data-models/blob/master/guidelines.md`

static public transport schedules. For example there was one entity type used to represent a trip on a bus line scheduled at a particular time but we used this entity to represent all trips to one direction on the line.

In this passenger analytics case another issue was the nature of the data which made it challenging to adapt for FIWARE. The data was partly higher level statistics that could be filtered in various ways. As an example lets take statistics related to a travel i.e. traveling a bus line to one direction. From the passenger analytics system we transferred the travel total statistics e.g. total number of passengers. In addition we stored the travel daily statistics e.g. number of passengers for each day. However by using the filtering options offered by the system for the total statistics, it is possible to get more fine grained information. For example, we could get the total number of passengers for last year's April weekends between 12 and 16. Replicating the same functionality with FIWARE data models and APIs would be complicated.

Table 1: The used FIWARE data models and required additions.

Case	**Entity types**	**Added attributes**	**Added entity types**
Street lights	Device WeatherObserved StreetlightGroup StreetlightControlCabinet	**X**	
Smart street lights	StreetLight StreetlightControlCabinet **SwitchingGroup** * **AmbientLightSensor** *	**X**	**X**
Electric bus	Vehicle	**X**	
Bus passenger analytics	GtfsRoute GtfsTrip GtfsShape GtfsStop GtfsStopTime	**X**	
lighting maintenance	**3PhaseAcMeasurement**[6] *		**X**
Water consumption	Device Alert AirQualityObserved		
building visitor counting	**visitorCounter** * **fillLevelCounter** *		**X**
building air pressure	AirQualityObserved PointOfInterest	**X**	

[6] A not yet official earlier version of our ThreePhaseAcMeasurement

Table 1: The used FIWARE data models and required additions.

Case	**Entity types**	**Added attributes**	**Added entity types**
preventing vandalism	Alert		
Outdoor lighting	StreetlightGroup StreetlightControlCabinet	**X**	
Indoor conditions	AirQualityObserved NoiseLevelObserved **Room** *		**X**
indoor conditions 2	AirQualityObserved NoiseLevelObserved		

When creating new data models or expanding existing ones it is important to both understand the source data and the FIWARE data modeling principles well. This comes especially apparent when there is no single person who understands both. In most of our cases the CityIoT researchers had the FIWARE knowledge and the case implementer had knowledge about the data. This requires then good communication and also understanding that this communication is important. For example in the *bus passenger analytics* case the company was not involved with the CityIoT project and communication was sometimes slow. Understanding the data happens also on different levels: some questions can be related to higher level concepts such as how the various entities relate to each other and some questions can be detailed questions about a data point: what does it mean and what values it can have. For example in the *indoor conditions 2* case it was unclear how the used sensor measured the noise level. Was the value it gave the noise level at the measurement time or was it an average from a longer period. This kind of information is important when the data is used.

Creating the FIWARE model can also offer opportunities for making the data more understandable. In the *electric bus* case the source data had numeric values for the status of the bus doors, and user of the data was assumed to know the encoding. For the data model we gave these attributes understandable textual values.

Using the FIWARE service paths as part of the data model caused lot of discussion during the project. They allow the formation of a hierarchy for the data and thus fit nicely to certain cases. For example in the *Indoor conditions* case there were multiple air quality sensors around the school building in different floors and rooms. Thus it was quite natural to model the floor / room hierarchy with service paths. For example /f/2/202 for room 202 in the 2nd floor or /f/1/116 for room 116 in the 1st floor. However, there are serious technical limitations that discourages the use of service paths. They cannot be queried from Orion which means that they have to be separately documented in somewhere so that, for example, the user of the data can utilize them in their

API queries. For example, the Oulu FIWARE platform uses the CKAN data registry to document the available data. To avoid the tedious manual data entry of documenting the data to CKAN, scripts were developed in the project which read the service path information directly from Orion's MongoDB database and via the CKAN API registers this information to the data registry. A human user can then just add some clarifying description for the automatically added data. Another issue with service paths is that orion also allows an entity with the same id and type to be created under different paths which can cause confusion later. Another limitation is related to STH Comet. Its way of storing the service path information causes strict limitations to the length of the path. The alternative for the service path hierarchy is to just use entity relationships or other attributes. This issue is further discussed in the CityIoT Building Data Model - Storing location information document.

When actually using the data models with FIWARE components it has to be noted that the components do not have any special support for data models. For example, Orion does not do any higher level validation for the data. It only cares that the data is valid NGSI v2 entities. Thus validating the data against a data model is the responsibility of the user.

4.2 Data conversion

This section explores issues related to converting the data from its original source into FIWARE entities required when using Orion directly and not through an agent. It explores what kind of conversions had to be made and how the construction of the entities and their attributes was done.

In all of the Tampere cases the data conversion had a similar basis. Data was fetched via a web API and the data was received in the JSON format. So, in all of these cases the data conversion involved building suitable entity representations from the fetched JSON.

In most cases the conversion was just mapping values from the source JSON to the FIWARE entity attributes. However, in some cases the value was also converted such as the door status values in the *electric bus* case described in the previous section. In the *street light* case we went even further and used and external service as a part of the data conversion. The source data did not contain understandable location coordinates. It only had street addresses which were then converted into location coordinates with the help of an external service.

Time information can sometimes cause conversion issues if the time zone used is not clearly specified in the format used by the data source. FIWARE uses the ISO 8601 time format and further recommends that UTC is used as the timezone. Thus for example Monday 9th of March 2020 at 10:42:30 Finnish normal time (UTC +2) should be represented as 2020-03-09T08:42:30Z. For example in the *street lights* case most of the timestamps did not contain time zone information. In the *electric bus* case unix timestamps were used so the conversion to the ISO format was easy with build-in methods of the programming language used in the conversion tool.

Some times values from multiple source attributes had to be combined under one attribute for example three phase voltage and current in the *street light* case or latitude and longitude in the *electric bus* case. In this kind of case it has to be kept in mind that subattributes of a StructuredValue attribute do not have their own metadata. Instead they all share the same metadata most notably the same timestamp. So, when combining separate measurements under one StructuredValue they all have to share the same timestamp.

Often a complete entity could not be constructed from the data returned by a single API request. Instead multiple requests had to be made to different API endpoints or the single API endpoint with different parameters. In the *street lights* case a single *Device* entity representing a door status sensor associated with a group of street lights required data from two API endpoints: one providing the electricity measurements and the location of the group and another providing the actual door status. In the *electric bus* case history for each measurement had to be fetched separately from the same endpoint by giving different measurement ids as a parameter.

For example if we want updates for one bus's location attribute for a 10 second period starting on 2nd March 2020 at 17:05:50 UTC. We have to make two HTTP requests to the data source: one to fetch latitudes for the time period and one to get longitude values for the period. Listing 2 line 1 - 48 shows the relevant parts of the request URIs and the response JSON. Note that the only difference in the paths on lines 1 and 25 are the measurement ids between datanode and processdata. The query parameters begin and end define the time period as unix timestamps in microseconds. Both responses contain 3 measurements each consisting of the coordinate value (v) and timestamp (ts) which is also a unix timestamp in microseconds. From this data we can get three latitude longitude value pairs where latitude and longitude share the same timestamp. However the data source does not guarantee that the timestamps match so that has to be checked during the data conversion. From the latitude longitude pairs we can then create the three location attribute updates with timestamp metadata shown starting on line 49.

As these experiences show, although the FIWARE data models are quite simple as is in principle the data conversion, it still requires some work. One company implementing the pilots thought the FIWARE data structures to be heavy from the perspective of integrating their system to FIWARE. This company and another company also felt that adapting their data to FIWARE required more effort than similar cases. The use of the ultralight IoT agent might have helped in these cases since the data format is more simple. However the agent still has to be configured for the conversion to NGSI-v2. The agent conversion has also some limitations. It cannot change the values to be more understandable or combine different values under a single StructuredValue attribute.

4.3 Sending the data

This section describes experiences related to sending the converted data to FIWARE. It discusses the various ways of adding and modifying the entity data

Listing 2: Data conversion example with two responses from a data source and three entity attribute updates created from them.

```
path: /datanodes/2074/processdata?begin=1583168750000000&end=1583168760000000
response: {
  "type": "processData",
  "limit": 10000,
  "order": "ascending",
  "begin": 1583168750000000,
  "end": 1583168760000000,
  "items": [ {
      "type": "num",
      "ts": 1583168750222297,
      "v": 61.502068
    },
    {
      "type": "num",
      "ts": 1583168755193949,
      "v": 61.502148
    },
    {
      "type": "num",
      "ts": 1583168756233061,
      "v": 61.502148
    }
  ]
}
path: /datanodes/2080/processdata?begin=1583168750000000&end=1583168760000000
response: {
  "type": "processData",
  "limit": 10000,
  "order": "ascending",
  "begin": 1583168750222297,
  "end": 1583168760222297,
  "items": [ {
      "type": "num",
      "ts": 1583168750222297,
      "v": 23.779667
    },
    {
      "type": "num",
      "ts": 1583168755193949,
      "v": 23.779671
    },
    {
      "type": "num",
      "ts": 1583168756233061,
      "v": 23.779671
    }
  ]
}
Attribute updates:
"location": {
    "type": "geo:json",
    "value": {
        "type": "Point",
        "coordinates": [ 23.779667, 61.502068 ]
    },
    "metadata": {
        "timestamp": {
            "type": "DateTime",
            "value": "2020-03-02T17:05:50.222297"
        }
    }
}
"location": {
    "type": "geo:json",
    "value": {
        "type": "Point",
        "coordinates": [ 23.779671, 61.502148 ]
    },
    "metadata": {
        "timestamp": {
            "type": "DateTime",
            "value": "2020-03-02T17:05:55.193949"
        }
    }
}
"location": {
    "type": "geo:json",
    "value": {
        "type": "Point",
        "coordinates": [ 23.779671, 61.502148 ]
    },
    "metadata": {
        "timestamp": {
            "type": "DateTime",
            "value": "2020-03-02T17:05:56.233061Z"
        }
    }
}
```

via the FIWARE APIs.

In FIWARE the data is not automatically copied from Orion to history component unless there is a suitable subscription. Thus, if some entity attributes should be stored in history component, a new subscription should be added. Entities can be created by sending their JSON representation to Orion. This first data contains at least an entity id and the type and usually some static attributes whose value will not change such as a name or location for something that does not move. After that dynamic attributes, whose value changes over time, can be updated as required. Orion offers multiple ways for updating the entity. There are API endpoints for updating a single entity attribute or for updating multiple attributes at once by sending an entity representation containing the attributes with new values. Orion also has a batch update endpoint that is used to create or update multiple entities with one request. It takes a list of entity updates as the request payload.

In many cases we found the batch update method very useful. When there is lots of updates it makes the update process more efficient. Though if Orion subscriptions are used to update the history component, some care is required when constructing the update list. If a single entity update has multiple attributes they should have the same timestamp in order to be stored correctly to the QuantumLeap component.The reason for this is explained in the next section. If there are multiple updates for the same attribute of a single entity in one update request, some of them might not be sent successfully to the history component. Another issue related to sending multiple updates of the same entity is to make sure that the newest update is sent last to Orion. Orion considers the information send most recently to be the most up to date version of the entity regardless the actual attribute timestamps.

The Orion subscription system has also other issues when it is used with a history component. Even if the entity updates are sent in separate requests to Orion all of them might not arrive to the history component. There are performance issues with the subscription system when too much data arrives too fast. This happened for example in the *electric bus* and *street lights* cases. To avoid this the sending of the data has to be slowed down, i.e., there must be a short gap between requests. This kind of data sending is challenging since you want to be as efficient as possible but you don't want to lose any data and there is no easy way to find the correct speed for sending the data. It has to be also noted that even though Orion can consider a subscription notification successfully sent, it does not guarantee that the data has been successfully stored to the history component. This can happen if an error occurs in the history component after it has acknowledged the notification request. Care has to be also taken when creating the subscriptions. Its possible to create the same subscription twice or to create different subscriptions whose notification conditions and notification contents overlap. The result then can be duplicate values in the history component.

Another possibility of avoiding the subscription issues is to simply not use them and to send the data directly to the history component. This was the case in the *electric bus* case where there were also other issues with the subscription system. In that case QuantumLeap was the history component used. The entity updates were directly sent to QuantumLeap's notify API endpoint where normally the Orion subscription notifications arrived. Though there was an issue where QuantumLeap did not accept notifications containing multiple updates even though the NGSI v2 specifies that a notification can contain multiple items. We were able to change the code to work with multiple updates. We also made a pull request about this change which was accepted.

A different way of adding data to Orion is to link it to another Orion via subscriptions. Orion has an API endpoint for processing subscription notifications. This allows data transfer between different FIWARE based platforms. This feature was used in the *smart street lights* case where the pilot implementer had connected the street lights to their own FIWARE platform from where data was then forwarded to the Tampere university FIWARE platform.

4.4 Data processing and storage

This section explores how the way FIWARE stores and handles the data affected the cases. It both affected how the data was sent and how it could be used.

The other reason for not using the subscription system in the *electric bus* case, as discussed in the previous section, were its issues in updating attributes that are updated independently from each other. Some measurements are updated about every second and some every few seconds. Creating a suitable subscription or subscriptions for this kind of data turned out to be impossible due to the way QuantumLeap processes and stores the data. A simple subscription, where all attributes are sent if any attribute changes, causes the original update times of attributes to be lost. This is because although the notification can have attributes with different timestamps, QuantumLeap chooses only the most recent time stamp of them and stores all attributes under that. This is also the reason for only including attributes that share the same timestamp into a single entity update described in the previous section. The consequences of this way of handling the data is illustrated in table 2, which shows the correct data, and table 3, which shows the data when using the simple subscription. However even if QuantumLeap would handle timestamps correctly we would then get duplicate values for attributes under the same timestamp. It has to be also noted that after the *electric bus* case collector was implemented, an option was added to Orion which sends only those values that have changed. This option probably would have solved this issue. This new option was introduced with Orion version 2.3.0 while we were using Orion version 2.2.0 at the time of implementation.

The other option of using subscriptions in the *electric bus* case was to create attribute specific subscriptions where if the attribute is changed its value is notified. But here also the way QuantumLeap processes and stores the data causes an issue as illustrated in table 4. Each attribute notification is processed and stored separately and the resulting data looks odd. It contains multiple database entries for the same timestamp, where each entry has a value for just only one attribute and shows that other attributes do not have any values.

Table 2: Data when send directly to Quantumleap.

Timestamp	airTemperature	chargeState	doorStatus	power
2019-05-27T09:00:01.000	10.90625	null	null	-34.8
2019-05-27T09:00:02.000	null	null	null	-54.2
2019-05-27T09:00:03.000	10.90625	null	null	-83.4
2019-05-27T09:00:04.000	null	null	null	-103.9
2019-05-27T09:00:05.000	null	77.0	closed	-152.0

Table 3: Data in Quantumleap when using a simple subscription.

Timestamp	airTemperature	chargeState	doorStatus	power
2019-05-27T09:00:01.000	10.90625	77.0	closed	-34.8
2019-05-27T09:00:02.000	10.90625	77.0	closed	-54.2
2019-05-27T09:00:03.000	10.90625	77.0	closed	-83.4
2019-05-27T09:00:04.000	10.90625	77.0	closed	-103.9
2019-05-27T09:00:05.000	10.90625	77.0	closed	-150.0

Because of this we also rounded the timestamps in to second precision from the original microsecond precision. Otherwise every measurement had a unique timestamp and thus its own entry in QuantumLeap as shown in table 5.

Table 4: Data in Quantumleap when using attribute subscriptions.

Timestamp	airTemperature	chargeState	doorStatus	power
2019-05-27T09:00:01.000	10.90625	null	null	null
2019-05-27T09:00:01.000	null	null	null	-34.8
2019-05-27T09:00:02.000	null	null	null	-54.2
2019-05-27T09:00:03.000	null	null	null	-83.4
2019-05-27T09:00:03.000	10.90625	null	null	null
2019-05-27T09:00:04.000	null	null	null	-103.9
2019-05-27T09:00:05.000	null	77.0	null	null
2019-05-27T09:00:05.000	null	null	closed	null
2019-05-27T09:00:05.000	null	null	null	-152.0

Table 5: Data in Quantumleap when timestamps are not rounded.

Timestamp	airTemperature	chargeState	doorStatus	energyConsumed
2019-05-27T08:59:59.562	null	77.0	null	null
2019-05-27T09:00:00.393	null	null	closed	null
2019-05-27T09:00:01.009	null	null	null	9.6
2019-05-27T09:00:01.421	10.90625	null	null	null
2019-05-27T09:00:03.422	10.90625	null	null	null
2019-05-27T09:00:04.661	null	77.0	null	null
2019-05-27T09:00:05.478	null	null	closed	null

There was also another issue related to processing of timestamps in QuantumLeap. As discussed in 4.1 the FIWARE data modeling documentation instructs that the meta data attribute indicating attribute measurement time should be named *timestamp*. However in our first QuantumLeap use case *street lights*, we noticed that QuantumLeap does not process the timestamp in the *timestamp* attribute. Instead it expected a meta data attribute named *TimeInstant* which is used by some FIWARE IoT agent components. We had to make a code modification to QuantumLeap to get it working with our data. We also made a pull request for this change which was accepted. This default QuantumLeap behaviour may indicate that there is not enough coordination in the FIWARE community for ensuring that all parts work seamlessly together.

How the FIWARE components handle attribute metadata affects how data is sent and used. Metadata is stored only to Orion and not to the history components. Thus, even though the user of the data would be interested only on the data history, they still have to check the metadata from Orion if they require it. This also does not take into account the possibility that the metadata might have changed at some point. However when sending updates for an attribute to Orion its metadata has to be always sent even if it has not changed. Otherwise the existing metadata is lost.

As already discussed in 4.1 Orion does not perform any data model validation. However the lack of validation goes further giving the data sender a lot of responsibility on correct NGSI v2 usage. Orion does not check that a value of an attribute matches the given type so attribute of type Number could have a character string as its value. This requires also more from the data user since they have to also check the data before using it. Orion does not require that the structure of a StructuredValue stays the same. This of course can give the data sender more flexibility but again can complicate the data user's work. There is also a limitation in Orion's processing of timestamps. It can handle time only on second precision. It does not even just round time to seconds instead it just truncates the timestamp to seconds.

When using STH Comet as the history component we observed some performance issues. They are due to STH Comet calculating some aggregate values such as averages from the measurements always when new data arrives. These aggregates are then stored for future use. This behaviour can be turned off but this configuration works on the level of the STH Comet instance so for example it cannot be set on a FIWARE service level. So, if there are varying requirements between different data sources for aggregate data multiple STH comet instances are required with different configurations. This differs from QuantumLeap which calculates aggregates only when they are requested. Of course then some performance issues may arise if there are too many requests for aggregated data.

If Grafana is used in visualizing data stored in QuantumLeap there may arise an issue in how to store static data. This was the case in the *street lights* and *bus passenger analytics* cases. Generally speaking it makes sense to store static data such as bus stop locations to only Orion. Dynamic attributes such as number of daily passengers for a bus stop is then stored to both Orion and

QuantumLeap. However accessing data from Orion with Grafana is not trivial. Since we wanted to use the static data in visualizations, we ended up also storing it to QuantumLeap although strictly speaking it is unnecessary duplication of data.

5 Conclusions

This document reports experiences in collecting data to FIWARE from 12 smart city pilots implemented in the CityIoT project. The pilots covered various smart city domains such as street lights, public transport, prevention of vandalism and indoor condition monitoring of buildings. Data was collected to one of two FIWARE platform instances that both used the Orion context broker and either the STH Comet or QuantumLeap measurement history components. We divided the reported experiences in to four broad categories: data models, data conversion, data sending, and data processing and storage.

In these cases we found the existing FIWARE data models quite useful. We were able to utilize them in most of the cases. However, in many cases we had to make some additions. Based on these experiences it can be said that a FIWARE user, who adds new data to FIWARE platform, may need to make some modifications to existing models or even to create a new data model. One important lesson from the pilot cases was the importance of understanding the source data when designing the data model. This often requires good communication between various stakeholders. Another important thing to note when using the FIWARE data models is to keep in mind that the FIWARE components do not actually care about conformance to data models so the responsibility of correct usage and validation falls to the users.

Based on our experiences, the conversion of the data from its original source to the FIWARE format is, in principle, quite straightforward. However, the conversion can still require surprising amount of work when the structure of the source data differs from the NGSI-v2 model. The FIWARE entity data might have to be combined from different source API endpoints. Issues may arise when data has to be combined into more complex attribute values and at the same time the timestamps of the values need to be synchronized.

We found the sending of the data to the Orion component to be simple since Orion offers many ways of sending data from the update of a single attribute to a batch update of multiple entities. However, some issues can arise when the data should also be sent to the history component. This is the case especially when the amount of data is big and it is updated often. The recommended way of using the Orion subscription system suffers from performance and other issues described in subsection 4.3. Thus, in some cases it can be easier to just ignore the subscription system and send the data directly to the history component.

Our experiences show that the way the FIWARE components process and store data can affect the data collection in various ways. For example, the way QuantumLeap processes timestamps may also lead to sending the data directly without using the Orion subscription system. Other notable issues were handling of metadata, lack of strict data validation and the performance issues of the STH comet component.

This report shows that real-life use of FIWARE in different cases can reveal issues that might not be apparent by just reading the FIWARE documentation or experimenting with the FIWARE tutorials. Although this document mainly concentrates on FIWARE issues and difficulties, our purpose is not to discourage the use of FIWARE especially since this document covers only one aspect of FIWARE and does not for example cover the positive aspects, especially the ability to handle many types of complex data. Rather, we hope that this document can give people considering using FIWARE extra information for making their decision and planning of the work. We also hope that this document may help people already using FIWARE to avoid or deal with the issues we have encountered.