Management of Sensor Data with Open Standards

Philipp Hertweck
Fraunhofer IOSB
philipp.hertweck@iosb.fraunhofer.de

Tobias Hellmund
Fraunhofer IOSB
tobias.hellmund@iosb.fraunhofer.de

Hylke van der Schaaf
Fraunhofer IOSB
hylke.vanderschaaf@iosb.fraunhofer.de

Jürgen Moßgraber
Fraunhofer IOSB
juergen.mossgraber@iosb.fraunhofer.de

Jan Blume
Fraunhofer IOSB
jan-wilhelm.blume@iosb.fraunhofer.de

ABSTRACT

In an emergency, getting up-to-date information about the current situation is crucial to orchestrate an efficient response. Due to its objectivity, preciseness and comparability, time-series data offer broad possibilities to manage emergency incidents. Since the Internet of Things (IoT) is rapidly growing with an estimated number of 30 billion sensors in 2020, it offers excellent potential to collect time-series data for improving situational awareness. The IoT brings several challenges: caused by a splintered sensor manufacturer landscape, data comes in various structures, incompatible protocols and unclear semantics. To tackle these challenges a well-defined interface, from where uniform data can be queried, is necessary. The Open Geospatial Consortium (OGC) has recognized this demand and developed the SensorThings API standard, an open, unified way to interconnect devices throughout the IoT, which is implemented by the **FR**aunhofer-**O**pensource-**S**ensor**T**hings-Server (FROST). This paper presents the standard, its implementation and the application to the domain of crisis management.

Keywords

Sensor data management, Time-series data, Data storage and retrieval, Open-source software, SensorThings API, OData, REST, MQTT, FROST

INTRODUCTION

The situational awareness of emergency managers relies on up-to-date and correct information. One possibility to retrieve most accurate and objective data is the utilization of sensors collecting time-series data, for example water levels of a river, wind speed or temperature. Monitoring a single sensor alone might not offer much benefit; however, the monitoring of many sensors targeted at a linked causality brings valuable insights. If these sensors are connected to a network, one speaks of the Internet of Things (IoT), which is rapidly growing: due to narrow production costs and huge potential in the field (Lee & Lee, July-August 2015), more than 30 billion connected IoT-devices are expected in 2020, growing to 75 billion devices in 2025 (Statista.com, 2019). One of the most promising prospects of IoT is the comparatively simple collection of vast amounts of data. Through a large number of devices monitoring different natural phenomena, an unlike larger amount of data can be collected. Still, since several different manufacturers produce these devices, the access to data is hindered through different data models and access protocols. In detail, data coming from IoT-devices is characterized through several traits. Firstly, the data comes from different sources and is heterogeneous, it is unstructured and without clear semantics. Without preprocessing, the data is not interoperable in the first place. Secondly, the large number of devices causes a mentionable amount of data that requires an annotation with metadata, which provides semantic information, to make them reusable. Lastly, the flexibility of IoT devices allows the collection of data at different points in time at different places offering spatial-temporal correlations (Li, Liu, Tian, Shen, & Mao, 2012).

The Open Geospatial Consortium (OGC) has recognized these challenges and developed the SensorThings API. It is a standard for the collection, storage and retrieval of time-series data. The data can be collected through IoT-devices, but it is not limited to these: the number of available sand packs during a flood event at a specific location or the number of manufactured items in a production line could be collected through other devices and subsequently be integrated, too. The standard defines a model for sensor data and its metadata, as well as an interface for both data storage and retrieval. The service can be queried with powerful filters, including geospatial search possibilities. The goal of the SensorThings API is to offer a unified way for the usage of sensor-data and to facilitate the development of data-driven applications. The Fraunhofer Institute of Optronics, System Technologies and Image Exploitation IOSB adopted this standard and has developed an open-source Java implementation of the SensorThings API. This implementation, called FROST, is fully compliant to the standard and certified by the OGC. FROST has been deployed in several projects where it forms the single point of managing time-series data.

This paper introduces the OGC SensorThings API in detail; explains the data model, the query options and interfaces. Subsequently, the FROST implementation and supplementary software is introduced. In a further section, it presents two EU-funded projects located in the domains of Climate Change and Crisis Management, in which FROST was successfully applied. Before the conclusion, a short passage highlights further developments and the relevance of this research.

RELATED WORK

The Sensor Observation Service (SOS) bases, like the SensorThings API on the OGC Observations and Measurements standard (O&M). It is, just like the SensorThings API a standard of the OGC and has been published in 2012 (Open Geospatial Consortium, 2012). It defines a web service interface, which allows querying observations, sensor metadata and representations of observed features. It defines means to register new sensors and remove obsolete sensors. The interface is realized by using the Simple Object Access Protocol (SOAP). The data itself is encoded in the Extensible Markup Language (XML). SOS offers at least three service operations: *GetCapabilities*, *GetObservation* and *DescribeSensor*, whereas the first will retrieve a description of SOS itself, describe further operations (if available) and state what kind of measurements are recorded. The second will retrieve specific observations and their metadata. The third gives explicit information about a sensor, e.g. measurement parameters etc.

The following abstract highlights the differences between SOS and the SensorThings API. The interested reader can find a full comparison in (Jazayeri, Liang, & Huang, 2015). In contrast to the *Representational State Transfer Principle* (REST), which is used by the SensorThings API, the SOAP interface used by SOS is in general more complex to use and has a bigger entry hurdle (zur Muehlen, Nickerson, & Swenson, 2005). The SensorThings API instead, using the *JavaScript Object Notation* (JSON), is more efficient regarding the amount of transmitted data, than the XML-encoded data transfer of the SOS (Mumbaikar & Padiya, 2013). However, SOS can have a JSON encoding extension, just as a SensorThings Service could have an XML encoding extension. As the SensorThings API has been developed by the OGC, it is backwards compatible to the SOS on a service level: all service functions defined by the SOS can be found in the SensorThings API. In contrast, through the implementation of ODATA filters, the SensorThings API offers more query capabilities on the service level, which cannot be answered by an SOS.

SENSORTHINGS API

To ease out the integration of different devices in data driven applications, the OGC developed a flexible, easy to use standard for the exchange of information in the context of the Internet of Things. With the SensorThings API standard, a framework, which interconnects sensing devices, data and applications over the web, is provided. The standard focuses on time-series data. Time-series are measurements of the same observed property, for instance the temperature, taken at different points in time. They can be measured in a more-or-less regular interval, e.g. every hour. The sensor can have either a fixed (fixed-point) or a moving (moving-point) location. An example for a fixed-point time-series are automatic water level gauges or more general the national weather stations available throughout Europe. These sensors are located in-situ and generate data at a fixed time interval. Sensors that can be hand-held or mounted on a vehicle or drone, and therefore have a different location for each measurement generate moving-point time-series. Drone-collected meteorological values are an example of this type.

DATA MODEL

The OGC SensorThings API data model not only covers plain sensor measurements, but also metadata like the unit of the measurement, a sensor description or a location. The metadata is connected to the original data stream.

The data elements are linked to each other enabling the user to find all necessary information of interest. The data model consists of eight entities, including properties and relations, which are shown in Figure 1. The model is described in following.

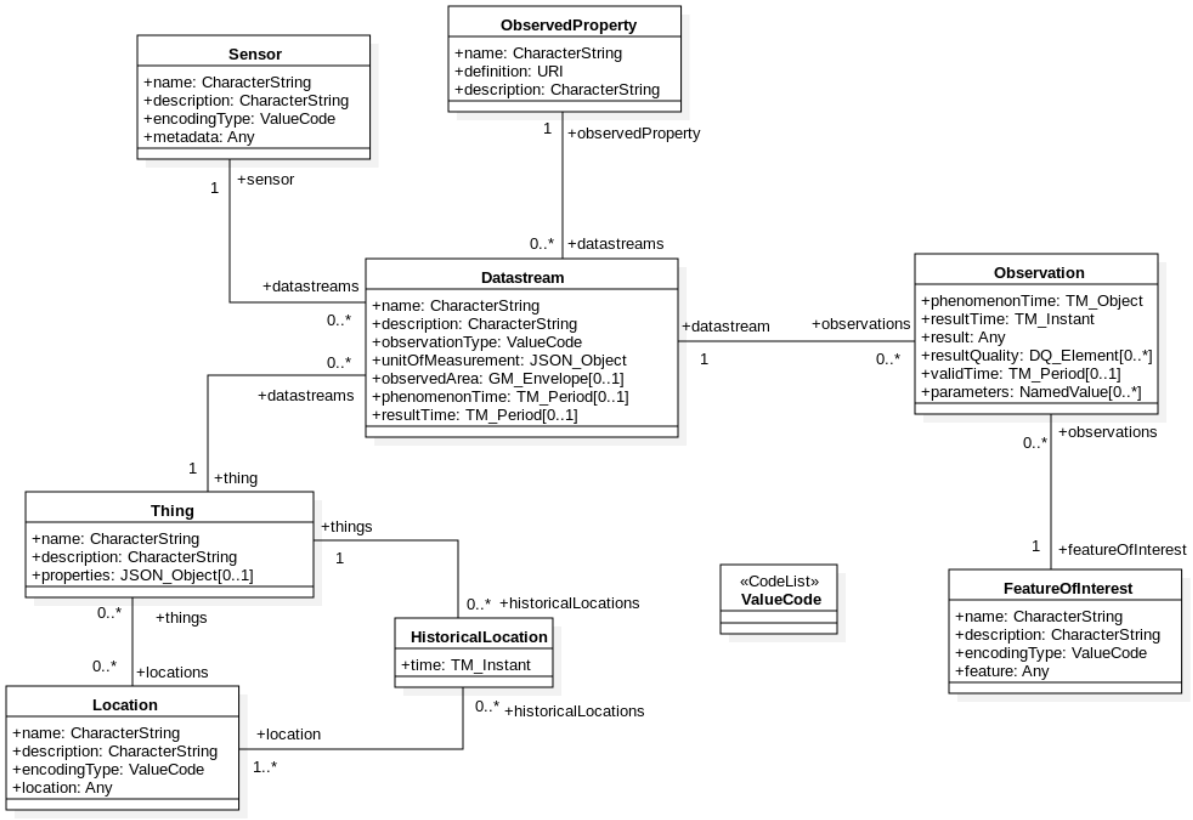

Figure 1: SensorThings API data model (Open Geospatial Consortium, 2016)

A *Thing* is a virtual or physical object. Depending on the use-case, the *Thing* can be the object being observed, like a river section, or the sensor platform, such as a satellite. *Location* describes the position of a *Thing*. A position can be described through geographic locations, encoded as points or areas. Symbolic locations, like a postal address, are possible, too. The *HistoricalLocation* is the link between a *Thing* and a *Location*, with the time indicating when the *Thing* was in a certain *Location*. A moving *Sensor* has several *HistoricalLocations*. A motionless *Thing* has none. This is defined through the SensorThings API, where the *HistoricalLocation* is created, when the *Thing* moves for the first time. A *Sensor* generates the data, which is described through the OGC SensorThings API Data Model. A *Sensor* can collect multiple *Datastreams*. A weather station for example can collect both temperature and humidity; in this example, the *Sensor* weather station would have two entities *Datastream*, one for each temperature and humidity. *Observation* contains the measurement made by a *Sensor*. An *ObservedProperty* is a characteristic of the *FeatureOfInterest* that is observed by a *Sensor*. For example, the water level in a river, or its temperature. A *Datastream* unites *Observations* of an *ObservedProperty*, which were made by a *Sensor* and are linked to a *Thing*. The *FeatureOfInterest* can be the geographic area or location for which an *Observation* was made. This can be the same as the *Location* of the *Thing*, which is often the case for in-situ sensing. In the case of remote sensing, the *FeatureOfInterest* can be different from the location of the *Thing*, dependent on what is chosen as *Thing*. The *FeatureOfInterest* is a geographical point or a polygon encompassing an area or volume, usually encoded in the format GeoJSON.

The relations between all these entities are described through the data model; this ensures finding all data entities that belong to another and only make sense in their own context.

REST

The OGC SensorThings API offers a RESTful interface for accessing the stored data. The REST programming paradigm is a well-known approach for realizing distributed systems. It is based on top of the Hypertext Transfer Protocol (HTTP), which forms the basis of the World Wide Web. REST is used for inter-machine communication and is widespread around web services. Alternatives are for example SOAP or Remote Procedure Calls.

The idea of REST was developed by Roy Thomas Fielding, published in his dissertation in (Fielding, 2000).

Fielding presents principles, which every REST-service must follow without suggesting how to implement them. The principles are described in the following. The first principle is the Client-Server Architecture, known from the World-Wide-Web. A server offers a service, which can be requested by a client. Through the usage of the widespread HTTP-protocol, a REST client implementation is available for nearly every programming language. The second characteristic of a RESTful service is its statelessness: every message sent to a REST service must contain all information needed to process this request. This brings two benefits: firstly, the service can be scaled according to the required usage. Secondly, it decreases complexity, since all information is summarized and no application state needs to be shared between two requests.

Other than its alternatives, REST demands unified interfaces. This contains for example the addressing scheme. Every entity in REST has to be uniquely identifiable, which usually is implemented through Uniform Resource Locators (URLs). The representation of entities is often achieved through JSON. Requests to the server are transmitted via HTTP. Thus, the HTTP-methods GET, POST, PUT, PATCH and DELETE are used to interact with the server. Hereby, GET is used to request information. To create a new instance of an entity the POST operation is used. By sending PUT (overriding a whole entity) or PATCH (overriding only provided attributes) messages, existing entities can be modified. DELETE is finally used to remove entities. For the SensorThings API, this means when entering the main URL of a SensorThings API server in a web browser, a GET request is issued to the server. The response of the server will contain a JSON file as shown in Figure 2.


```
value:
  0:
    name:   "Datastreams"
    url:    "https://beaware.server.de/SensorThingsService/v1.0/Datastreams"
  1:
    name:   "MultiDatastreams"
    url:    "https://beaware.server.de/SensorThingsService/v1.0/MultiDatastreams"
  2:
    name:   "FeaturesOfInterest"
    url:    "https://beaware.server.de/SensorThingsService/v1.0/FeaturesOfInterest"
  3:
    name:   "HistoricalLocations"
    url:    "https://beaware.server.de/SensorThingsService/v1.0/HistoricalLocations"
  4:
    name:   "Locations"
    url:    "https://beaware.server.de/SensorThingsService/v1.0/Locations"
  5:
    name:   "Observations"
    url:    "https://beaware.server.de/SensorThingsService/v1.0/Observations"
  6:
    name:   "ObservedProperties"
    url:    "https://beaware.server.de/SensorThingsService/v1.0/ObservedProperties"
  7:
    name:   "Sensors"
    url:    "https://beaware.server.de/SensorThingsService/v1.0/Sensors"
  8:
    name:   "Things"
    url:    "https://beaware.server.de/SensorThingsService/v1.0/Things"
```


Figure 2: FROST start page

As shown in Figure 2, the response contains all data model entities, as well as their URLs. Through these URLs, each individual on the server is addressable (e.g. */Sensors(10)*). Analogous to fetching data, an instance can be created, modified or deleted by sending a POST, PUT, PATCH or DELETE command to the appropriate URL. Since the database can contain a large amount of data, only a subset of all available data is returned for a GET command. This prevents both the server, as well as the client from an overload. Further data is sent upon request.

ODATA

The previous section showed how data can be retrieved and modified by using the REST interface. For many applications, it is not sufficient to retrieve all available data. Conditional upon the large variety of stored data, powerful and expressive filter mechanisms are required to receive the data of interest. In the SensorThings API, these are realized by applying the Open Data Protocol (OData), which is standardized by the Organization for the Advancement of Structured Information Standards (OASIS). It allows projections and filters similar to the Structured Query Language (SQL), which are specified as query strings in the URL. Projections can be done by naming the queried attributes in the *$select* parameter. For example, */Things?$select=@iot.id,description* will only select the id and the description of *Things*.

By passing the *$filter* parameter, it is possible to query for specific results. For example, */Observations?$filter=result gt 5* will return all observations that have a result value greater than 5. A wide range of filtering operators are supported; an exemplary list of supported functions is shown in Table 1.

Table 1. Exemplary list of functions that can be used with OData in the URL of requests

Comparison	Mathematical	Logical	String Functions	Geospatial	Date and Time
gt (greater than)	add (addition)	and	substringof(p0,p1)	geo.intersects(g1,g2)	now()
ge (greater equal)	sub (subtraction)	or	indexof(p0,p1)	geo.distance(g1,g2)	mindatetime()
eq (equal)	mul (multiplication)	not	trim(p0)	st_equals(g1,g2)	maxdatetime()
le (less equal)	div (division)		concat(p0,p1)	st_within(g1,g2)	date(t1)
lt (less than)	mod (modulo)			st_overlaps(g1,g2)	time(t1)
ne (not equal)					

MQTT

Message Queuing Telemetry Transport (MQTT) is an open-source protocol standardized by OASIS (Hunkeler, Truong, & Stanford-Clark, 2008). It follows the publisher-subscriber pattern. Thus, it is predestinated for collecting the data of IoT-sensors. By using this pattern, subscribers can register to a topic, which broadcasts information they are interested in. The information is initially provided by a publisher that sends a message to the corresponding topic, as soon as it gets available. A message broker (included in a SensorThings API compliant server) takes care of the subscriptions and forwards the messages to all the subscribers registered to the topic. A topic is a string that can have several hierarchical levels, separated by a slash. Through this, a client receives only the information published within these topics. The naming of the topics is analogue to the URLs of the entities in the SensorThings API implementation (e.g. */Datastreams*; see Figure 2 for others).

An implementation of the SensorThings API offers, next to the RESTful HTTP interface, an additional MQTT interface. The publisher-subscriber pattern is realized twice. On the one side, a SensorThings Server subscribes to the topics, which represent the entities. Sensors can use this interface to publish new observations, which are created on the server. On the other side, a sensor or - more often - a subsequent processing component may subscribe to changes of instances: this allows the implementation of features that need to directly react on changes (for example, by subscribing to the measurements of a water level sensor, a procedure can be triggered as soon as a threshold is reached).

Figure 3 shows a simplified schematic example: The thermocouple on the top right side posts its observations of 6.03°C to topic *v1.0/Datastreams(5)/Observations.* The actuator on the lower right side subscribes to the same topic and uses it as input for further processing: in this case, the gears would turn by 22.4° and could, for example, open or close a window. Clients can subscribe to topics (like the previously shown example) or to specific instances matching a filter (see previous chapter).

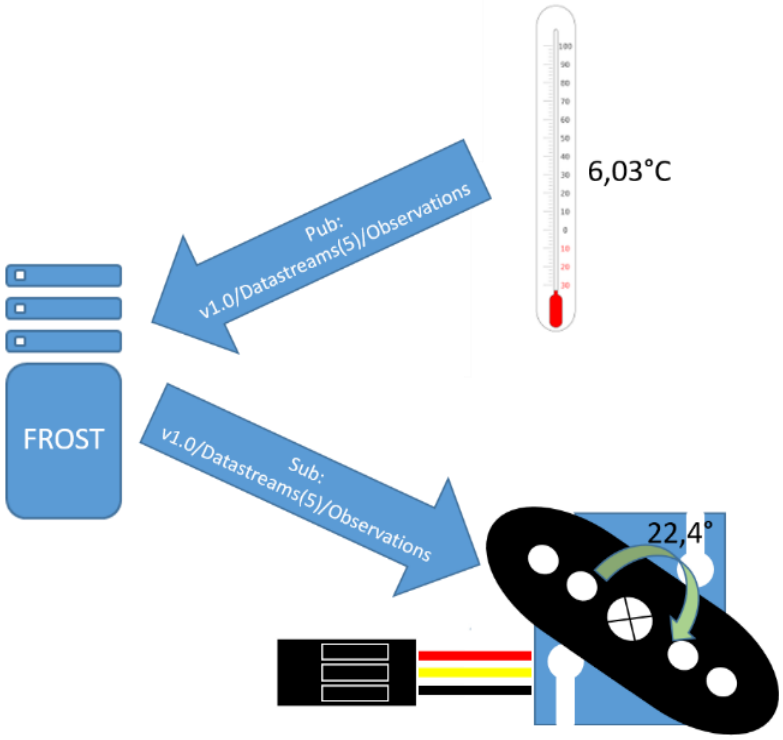

Figure 3: Using the MQTT Broker of FROST to interconnect multiple components

FROST

FROST is an open-source implementation of the SensorThings API standard, developed by Fraunhofer IOSB. The application was designed according to the SensorThings API standard and implements the named data model, REST-interface and MQTT extension and the specified ODATA querying mechanism. Therefore, the application is certified OGC compliant (Open Geospatial Consortium, 2016). The source code is publicly available on *https://github.com/FraunhoferIOSB/FROST-Server* under GNU Lesser General Public License v3.0. It is written in the Java programming language. Internally, FROST uses a PostgreSQL database for data storage, with the PostGIS extension to support geospatial queries. PostgreSQL is an object-relational database and PostGIS extends the database for spatial information. (For further information about both, please visit: *https://www.postgresql.org/* and *https://postgis.net/*).

To start an instance of FROST, different deployment possibilities exist. For example, the server is available as a pre-build WAR-file, which can be run on a web-server like Apache Tomcat or Wildfly. Both are freely available open source HTTP servers. The webpages *http://tomcat.apache.org/* and *http://wildfly.org/* will give the interested reader access to deeper information. Furthermore, FROST is available as a Docker image. This allows an easy start of an instance without the need of further dependencies, for testing and production-ready usage within a virtual environment. On *https://www.docker.com/* the software Docker is introduced. To scale out the deployment, a Helm (https://helm.sh/) chart for Kubernetes is available. Kubernetes (*https://kubernetes.io/*) is an open-source container orchestration platform, initially developed by Google, which offers the possibility to run the application in bigger cloud environments.

ARCHITECTURE

In this section, we will shortly describe the architecture of FROST. Especially when scaling the server in a cloud environment it is important to understand the four different FROST components. The different modules and their interaction are visualized in Figure 4.

The first component is the HTTP-frontend; it handles all incoming requests from the REST interface. In Figure 4 an example is shown, where this component is scaled to the number of two. Changes are directly written into the Database (second component) to have them persistent. Additionally a message is sent through an internal Message Bus (third component) to inform the MQTT components about the changed content. This fourth component evaluates if there are MQTT-subscriptions to which this information needs to be forwarded and does so if needed. In the example in Figure 4, this component is scaled to the amount of two.

There are two possibilities how the different components are bundled together. Using the WAR artifact or the "All-in-One"-Docker image, all components, except for the PostgreSQL-database are running inside a single instance of a Java Virtual Machine (JVM) as monolithic application. With an increasing number of requests, there might be the need to scale FROST. Since this horizontal scaling is not possible within a single JVM instance, there is the possibility to run all components separately. Even if there is only one instance of the database and the message bus needed, the MQTT and HTTP components can be scaled independently. Since all state is handled inside the database, those two components are stateless.

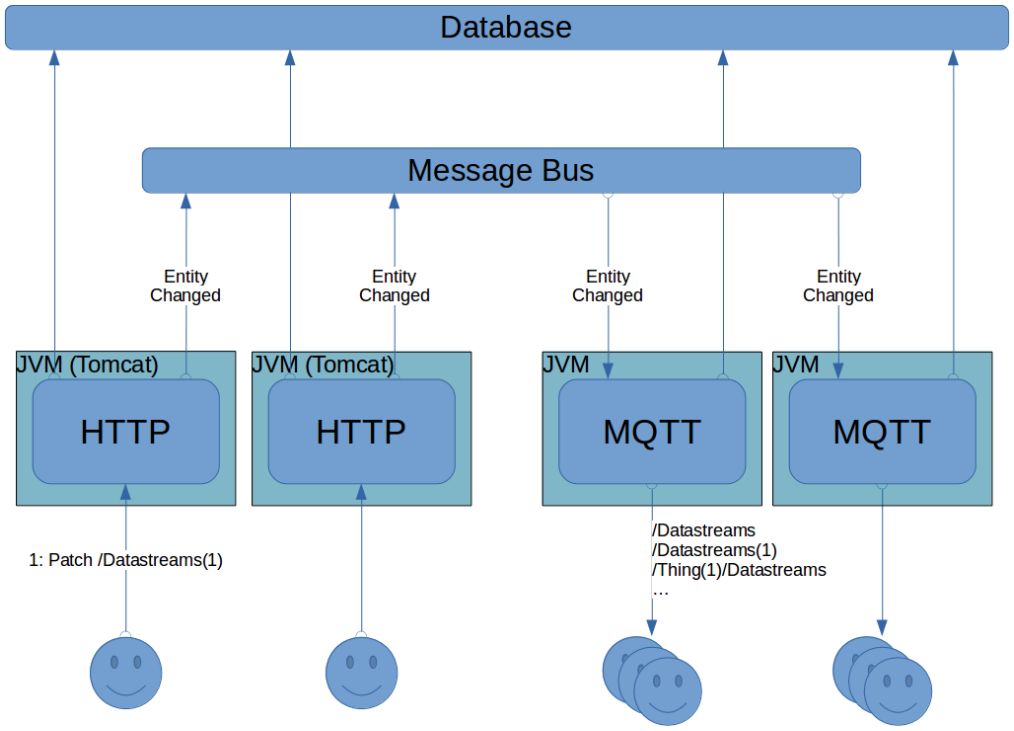

Figure 4: The FROST architecture

COMPATIBLE SOFTWARE TOOLS

Although the SensorThings API is comparatively new, the number of software tools around it is growing. In the following, we present a short list of freely available open-source software, which is compatible with FROST.

CHILLImport

The software is meant for application cases, where collected data is stored in a rudimentary file format and cannot be uploaded by an automated task. This for example is the case, for every sensor placed in a region without internet connection.

CHILLImport is a web-application whose main task is to offer a graphical interface to import data, whereas the software is flexible regarding the input file format and structure. The software converts the measured data into the SensorThings API data model and subsequently imports it into a FROST instance. To adapt different file formats and structures, the software can store and retrieve the configurations for the input file. The configurations can be reused, in case measured data with the same structure needs to be imported again. Configurations can be created by IT-affine users, whereas it is quite complex for an unexperienced user. During the import, an automatic duplicate check between the data to be imported and data stored within the server is conducted. This prevents redundant data within FROST. If an import fails, due to a corrupt import file, loss of connection or similar, CHILLImport prints out a file, in which the user can find all the data that was not imported.

The software is freely available under MIT license and can be found at GitHub:
https://github.com/Chillimport/ChillImport.

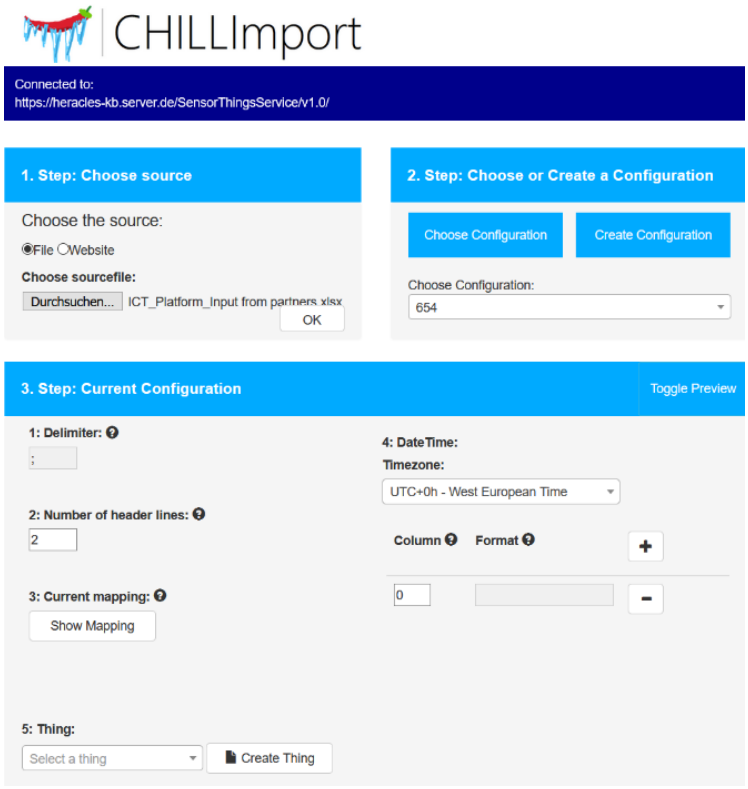

Figure 5: The CHILLImport user interface

FROST-Client

The FROST-Client is a Java library facilitating the interaction with servers and aims to simplify the development of SensorThings enabled client applications. It provides the possibility to connect to a server, which implements the SensorThings API, identified through a URL and create, read, update or delete data. These operations are available through Java classes and methods and therefore allow the easy integration of sensor data into an application.

It is freely provided under MIT license and can be found here: *https://github.com/FraunhoferIOSB/FROST-Client*.

The code snippet in Figure 6 shows a code snippet in the Java programming language, to connect to a FROST instance, posting a new *Thing*, updating it and finally deleting it by using the FROST-Client library.

```
URL serviceEndpoint = new URL("http://example.org/v1.0/");
FROSTService service = new FROSTService(serviceEndpoint);

Thing thing = ThingBuilder.builder()
    .name("Thingything")
    .description("I'm a thing!")
    .build();
service.create(thing);

// get Thing with numeric id 1234
thing = service.things().find(1234l);
// get Thing with String id ab12cd
thing = service.things().find("ab12cd");

thing.setDescription("Things change...");
service.update(thing);

service.delete(thing);
```

Figure 6: The FROST-Client

FROST-Manager

The FROST-Manager expands the FROST-Client through a graphical user interface managing data on any SensorThings Server. After configuring the URL of a FROST instance, it offers simple means for creating, retrieving, updating or deleting entities.

FROST-Dashboard

The FROST-Dashboard is one possibility to visualize data from the FROST-Server. It can display graphs showing sensor data over time or a visualization of alert levels in traffic lights, as shown in Figure 7. Further, it offers bar graphs, gauges or thermometers from which the user can configure the appropriate visualization.

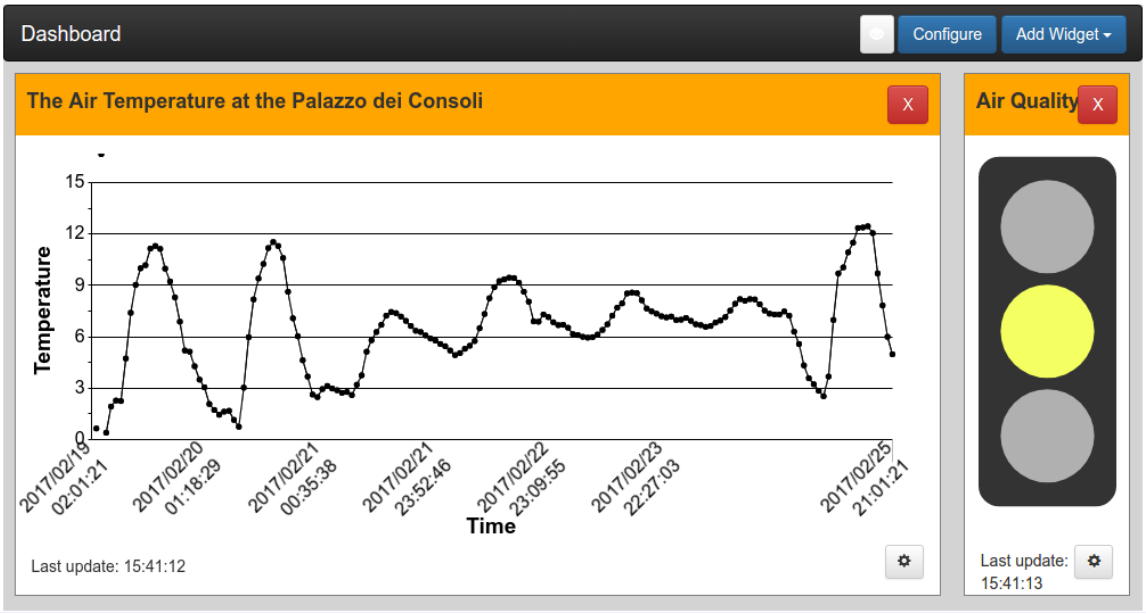

Figure 7: The FROST-Dashboard

Grafana-Dashboard Plugin

Grafana is a well-known application for displaying various kinds of time-series data. Especially for displaying metrics (e.g. of computer hardware or an application) it is very popular. It is also usable for displaying sensor data. A plugin is available to integrate data from a SensorThings API server, which can be found at the Grafana webpage: *https://grafana.com/plugins/linksmart-sensorthings-datasource.*

APPLICATION

This section presents two EU-funded projects, namely beAWARE and HERACLES, in which FROST is integrated as a central component within a bigger Decision Support System (DSS). By presenting both projects, we demonstrate the utilization of the SensorThings API in a practical way and prove the applicability in a hands-on context.

BEAWARE

The main goal of beAWARE is to develop a novel framework for crisis management that manages and supports authorities and end users before and during crises caused by extreme weather events. Up to now, the emergency management landscape consists of many splintered solutions for different problems. This leaves emergency management agencies with the task to integrate the best solutions for them into their system and to make sure that they properly work together. However, this represents a huge effort that many agencies are not able to make. The holistic approach to realize a crisis management framework provided by beAWARE shall help in this regard. The integrated solution of the DSS will include forecasts, early warnings, transmission and routing of the emergency data, aggregated analysis of multimodal input and management of the coordination between the first responders and the authorities.

FROST IN THE CONTEXT OF BEAWARE

Sensor data plays an important role in generating early warnings and gaining situational awareness. This could be for example meteorological data (temperature, humidity and rainfall), or the water level of a river section as well as weather forecasts. Depending on the use case, different sorts of sensors are available and their data can vary strongly regarding their type, frequency and format which makes the integration into a single platform difficult. FROST helps to solve this problem since it provides a unified interface for the sensor data. Further, it takes care of the sensor data management and offers rich querying mechanism, based on, for example, location of the feature of interest, time, type of sensor, and/or observed property.

When applying the SensorThings API model to beAWARE, for example to a water level gauge in a river section, the *Thing* could be the river section of which a *Sensor* measures the water level. The *Thing* would have a *Location*, with a polygon describing the layout of the river section. Since river sections usually do not move much, there would be no *HistoricalLocation*. The *Sensor* entity would describe the exact properties of the water level gauge, like brand, type and accuracy. *ObservedProperty* would be named "water level" and contains an exact reference to a water level definition. For this set of *Thing*, *Sensor* and *ObservedProperty*, there would be a *Datastream* that groups the water level observations of the sensor. Each value measured by the *Sensor* would be stored as an *Observation*. Since the *Sensor* is static, each *Observation* is linked to the same *FeatureOfInterest*, which has the exact coordinates of the *Sensor*. For a temperature sensor located at the same spot as the water level gauge, the same *Thing*, *Location* and *FeatureOfInterest* entities would be applied. Only the new entities *Sensor*, *ObservedProperty* and *Datastream* would have to be added.

In Figure 8, the locations of all sensors around Vicenza, a city in the north east of Italy, that are available in the server are displayed on a map.

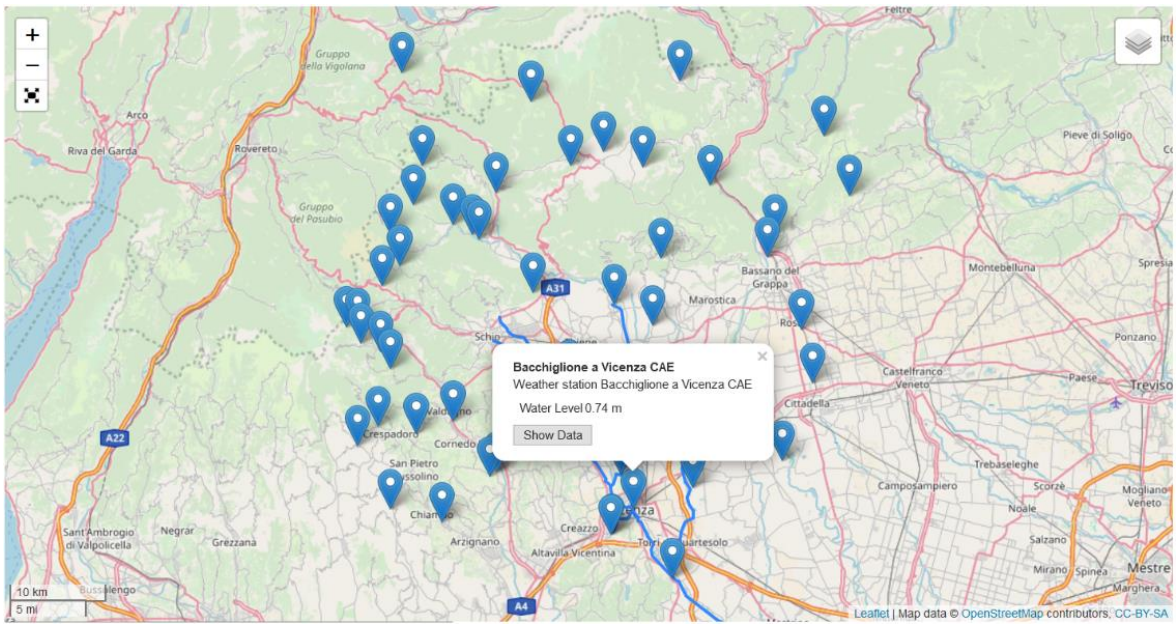

Figure 8: Map with water level and weather station sensors

By clicking on a sensor in the map, the data of the sensor, which is stored in FROST, will be visualized in a graph (see Figure 9). By selecting several sensors or by choosing several *ObservedProperties* more than one graph can be displayed in the diagram, which allows for a direct comparison of the data.

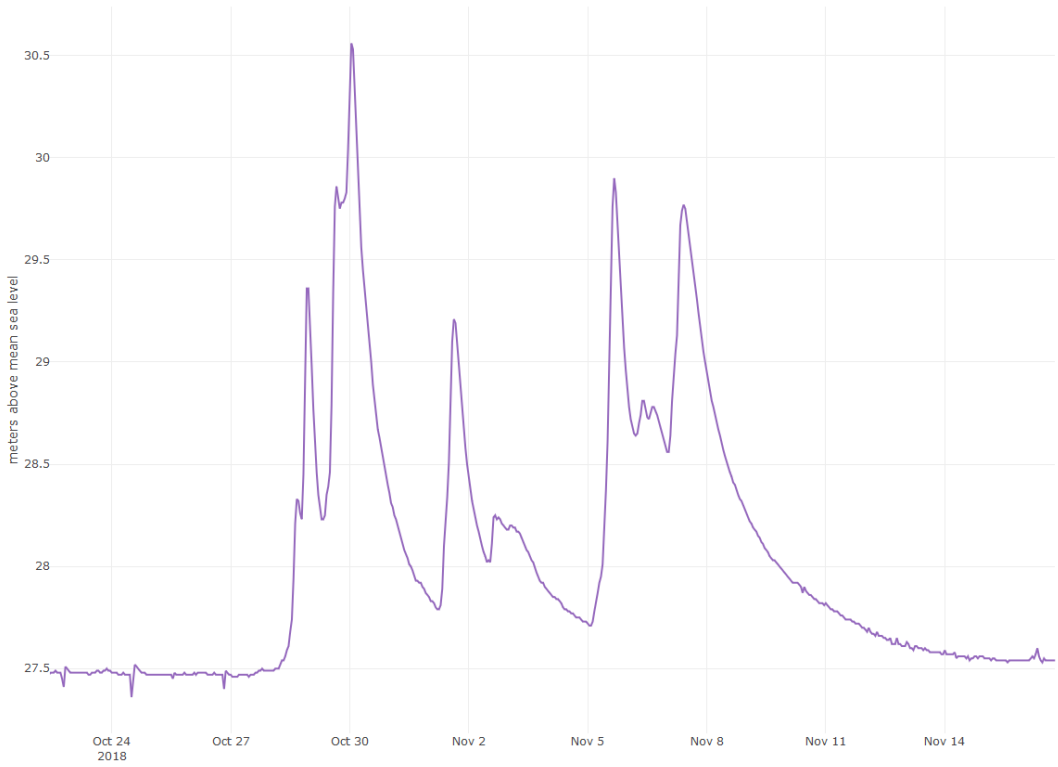

Figure 9: A graphical visualization of water-level values

HERACLES

The main objective of the HERACLES project is to design, validate and promote responsive systems and solutions for effective resilience of Cultural Heritage (CH) against climate change effects, considering as a mandatory premise a holistic, multidisciplinary approach through the involvement of different expertise (industry, scientists, conservators, social experts, decision, and policy makers). This is pursued through the development of a system exploiting an ICT platform able to collect and integrate multisource information in order to provide complete and updated situational awareness and support decision making for innovative measurements improving CH resilience, including new solutions for maintenance and conservation. The HERACLES effectiveness is validated in two test beds, namely Heraklion in Crete with the Knossos palace and the Sea Fortress and Gubbio in Italy with Palazzo dei Consoli and the medieval town walls. These test beds represent key study cases for the climate change impact on European CH assets.

FROST IN THE CONTEXT OF HERACLES

All time-series data collected in the HERACLES Project is stored in a FROST instance. Live data coming from sensors installed by the project-partners in the test-bed locations are automatically imported into the server. As in beAWARE, the sampling frequency of the sensors can differ a lot, for example, one data point per minute from a weather station to several 100 data points per second for an accelerometer. Therefore, automatic aggregations are applied for each sensor, in hourly and daily intervals. A visualization component in the HERACLES user interface will automatically choose the aggregation level that fits best to the time interval the user wants to see.

The following picture shows a crack on a building (Crack *C1*) that is monitored in the context of the HERACLES project. A thermocouple *T1* measures the surrounding temperature, while the linear variable differential transformer *LVDT1* measures the width of a crack (see Figure 10). The exceeding of threshold values or the predicted exceeding of a threshold alerts the cultural heritage manager, who triggers appropriate conservation or mitigation actions.

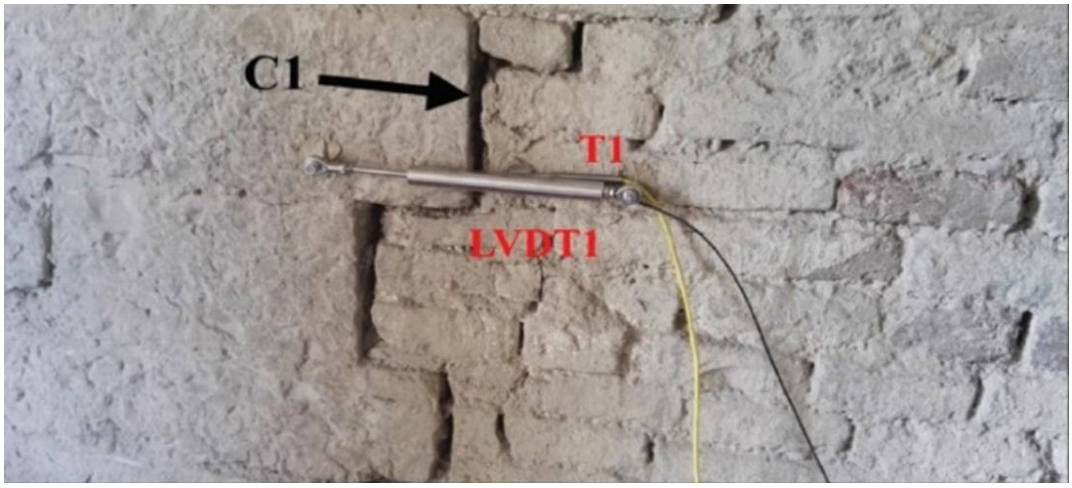

Figure 10: The observation of a crack by measuring temperature and crack width

The changing environmental parameters are collected through the sensors and posted once per day through an HTTP-POST request to the FROST instance deployed within the HERACLES cloud environment. The project partners can access both the raw data stored on the server for further analysis, or access a graphic visualization generated within the HERACLES user interface.

FUTURE DEVELOPMENT

The SensorThings API standard is comparatively new but the number of products implementing the standard is growing quickly. In parallel, the OGC continues developing the SensorThings API standard. Until now, by using the term SensorThings API, we referred to the first part of multiple specifications. We only considered the management of sensors, their metadata and measurements. Still, in the context of IoT more and more devices appear that not only monitor the environment but also manipulate the real world, by executing tasks. The OGC specification "OGC SensorThings API Part II - Tasking Core", released for public comment on February 20, 2018, adds actuation to the SensorThings API (Open Geospatial Consortium, 2018). In terms of the SensorThings API Part II - Tasking Core standard, such a device is called an *Actuator*. This can be any entity that accepts a defined parameter and uses it for further processing. Thus, an *Actuator* is not limited to hardware devices; an actuator can be anything that accepts a task to be executed, including, but not limited to, mathematical models (e.g. calculating min/max or the average of measurements during a time interval). The generic approach is analogue to the presented SensorThings API. The genericity allows the integration of many different devices as *Actuators* and it is possible to describe all necessary parameters to control actuators, and to define tasks that an actuator can execute.

The Tasking Core adds three entities to the SensorThings API: *Actuator*, *TaskingCapability* and *Task*:

- **Actuator:** A hardware or software component that can execute tasks.
- **TaskingCapability:** Describes the parameters that can be supplied to a task.
- **Task:** A task to be executed, with values for the parameters described in the linked *TaskingCapability*.

In the context of Crisis Management, Part II of the SensorThings API promises a lot of potential that needs to be investigated. Since every crisis needs a specific and tailored-to-its-needs response, the used equipment differs from crisis to crisis. In the context of IoT, new tools for first responders emerge such as remote controlled or autonomous robots. Rigos et al., for example, discuss the usage of a remote controlled robot that can crawl into collapsed buildings. The usage of state-of-the-art tools during a crisis is desirable, but imposes challenges: as we already motivated in the introduction, just like sensors, actuators require a standardized data model. This allows making efficient use of many actuators and unfolds synergy effects through their coordinated operation. These synergy effects can be exploited, when a single point for command and control exists, where the management of every controllable piece of equipment is possible. Another use-case is the automated response upon certain events: if a sensor posts an observation to a FROST instance, that exceeds a specific limitation, a standardized operation procedure can be executed upon this. Set the case, a water level sensor observes a value over a certain threshold; the automatic routine could imply the opening of a floodgate within the river area, to reduce the water pressure against its riverbed. The FROST development to support Part II of the specification is already ongoing. A generic data model for actuators could act as base for this single command and control point.

EXERCISES AND INSTRUCTIONS

The live demo will be split in two sections focusing on both technical users and end-users.

Firstly, the installation of a FROST instance will be demonstrated with the help of a Docker image. The link to the required image can be found under *https://github.com/FraunhoferIOSB/FROST-Server*. Secondly, sensors will be inserted into the data model. Thirdly, a short script in the will post data via HTTP to FROST. Instead of using live sensor data, a random generator will provide data values. An example query to retrieve data will be executed.

After that, more sophisticated use-cases from the beAWARE project will be presented, like accessing live data weather data or analysis results. Access to sensors will be provided through maps; with the help of graphs and forecasts, end-users can intuitively assess the situation and give a suggestion, if preparedness actions should be started for a city.

CONCLUSION

In this paper, we have motivated the use of an open standard to manage time-series data in emergencies to enhance situational awareness and to organize an efficient response. We argue that time-series data bears huge potential but also challenges. Though the challenges imminent in IoT-data are complex, they can be overcome by rigorously adopting standards and implementing a uniform interfacing concept. In this context, the standard SensorThings API has proven itself beneficial, since it provides an open, geospatial-enabled and unified way to interconnect sensors, data and applications of the Web. The FROST implementation is a lightweight and open-source implementation of this standard and can be utilized to collect and disseminate time-series data in an emergency. The exploitation of the data improves situational awareness in different settings, including natural disasters. The applicability of FROST has been demonstrated in the projects beAWARE and HERACLES. Furthermore, through the open-source approach of FROST, a vibrant ecosystem has already formed and supplementary software is publicly available.

ACKNOWLEDGMENTS

Funding: The project leading to this application has received funding from the European Union's Horizon 2020 research and innovation programme under Grant No. 700395 and Grant No. 700475.

Conflicts of Interest: The funders had no role in the design of the study; in the collection, analyses, or interpretation of data; in the writing of the manuscript, and in the decision to publish the results.

References

Fielding, R. T. (2000). *Architectural Styles and the Design of Network-based Software Architectures.* Dissertation, University of California. Irvine, CA.

Hunkeler, U., Truong, H. L., & Stanford-Clark, A. (2008, January 06-10). MQTT-S - A publish/subscribe protocol for Wireless Sensor Networks. *2008 3rd International Conference on Communication Systems Software and Middleware and Workshops*.

Jazayeri, M. A., Liang, S. H., & Huang, C.-Y. (2015, September 22). Implementation and Evaluation of Four Interoperable Open Standards for the Internet of Things. *Sensors, 15 (9)*, pp. 24343-24373.

Lee, I., & Lee, K. (July-August 2015). The Internet of Things (IoT): Applications, investments, and challenges for enterprises. In *Business Horizons, Volume 58, Issue* (pp. 431-440). Indiana: Elsevier.

Leonard, A., Mitchell, T., Masson, M., Moss, J., & Ufford, M. (2014). OData Source. In *SQL Server Integration Service Design Patterns, Second Edition* (pp. 251-260). Berkely, CA: Springer.

Li, T., Liu, Y., Tian, Y., Shen, S., & Mao, W. (2012, November 20-23). A Storage Solution for Massive IoT Data Based on NoSQL. *2012 IEEE International Conference on Green Computing and Communications*, pp. 50-57.

Mumbaikar, S., & Padiya, P. (2013, May). Web Services Based on SOAP and REST Principles. *International Journal of Scientific and Research Publications, Vol. 3 Issue 5*, pp. 1-4.

Open Geospatial Consortium. (2012, 04 20). *opengeospatial.org.* Retrieved November 30, 2018, from https://portal.opengeospatial.org/files/?artifact_id=47599

Open Geospatial Consortium. (2016, July 26). *OGC SensorThings API Part 1: Sensing*. Retrieved November 30, 2018, from http://docs.opengeospatial.org/is/15-078r6/15-078r6.html

Open Geospatial Consortium. (2016, Januar 11). *opengeospatial.org*. Retrieved November 30, 2018, from http://www.opengeospatial.org/resource/products/details/?pid=1371

Open Geospatial Consortium. (2018, August 01). *opengeospatial.org*. Retrieved November 30, 2018, from http://www.opengeospatial.org/standards/requests/162

Rigos, A., Sofianos, D., Sourlas, V., Sdongos, E., Koutsokeras, M., & Amditis, A. (2018). A resilient, multi-access communication solution for USaR operations: the INACHUS approach. *14th International Conference on Wireless and Mobile Computing, Networking and Communications (WiMob)*, (pp. 255-261). Limassol: IEEE.

Statista.com. (2019). (Statista) Retrieved November 30, 2018, from https://www.statista.com/statistics/471264/iot-number-of-connected-devices-worldwide/

zur Muehlen, M., Nickerson, J. V., & Swenson, K. D. (2005, July). Developing web services choreography standards—the case of REST vs. SOAP. *Decision Support Systems Vol. 40, Issue 1*, pp. 9-29.