Enabling Efficient Semantic Stream Processing across the IoT Network through Adaptive Distribution with DIVIDE

Mathias De Brouwer[1*], Filip De Turck[1] and Femke Ongenae[1]

[1*]IDLab, Ghent University – imec, iGent Tower – Department of Information Technology, Technologiepark-Zwijnaarde 126, B-9052 Ghent, Belgium.

*Corresponding author(s). E-mail(s): mrdbrouw.DeBrouwer@UGent.be;
Contributing authors: Filip.DeTurck@UGent.be;
Femke.Ongenae@UGent.be;

Abstract

In the Internet of Things (IoT), semantic IoT platforms are often used to solve the challenges associated with the real-time integration of heterogeneous IoT sensor data, domain knowledge and context information. Existing platforms mostly have a static distribution and configuration of queries deployed on the platform's stream processing components. In contrast, the environmental context in which queries are deployed has a very dynamic nature: real-world set-ups involve varying tasks, device resource usage, networking conditions, etc. To solve this mismatch, this paper presents DIVIDE, an IoT platform component built on Semantic Web technologies. DIVIDE has a generic design containing multiple subcomponents that monitor the environment across a cascading architecture. By monitoring the use case context, DIVIDE adaptively derives the appropriate stream processing queries in a context-aware way. Using a Local Monitor deployed on edge devices, situational context parameters are measured and aggregated. The Meta Model allows modeling these measurements, and meta-information about devices and deployed stream processing queries. Through the definition of application-specific Global Monitor queries that are continuously evaluated centrally on the Meta Model, end users can dynamically configure how the situational context should influence the window parameter configuration and distribution of queries in the network. The paper evaluates a first implementation of DIVIDE on a homecare monitoring use case. The results show how DIVIDE can successfully adapt to

varying device and networking conditions, taking into account the use case requirements. This way, DIVIDE allows better balancing use case specific trade-offs and achieves more efficient stream processing.

Keywords: network monitoring, device monitoring, RDF stream processing, Internet of Things, adaptive queries, window parameters

1 Introduction

The Internet of Things (IoT) is characterized by a high variety of internet-connected devices and sensors that continuously generate and process data. A big advantage of the IoT is that processing devices and applications can easily combine and integrate existing domain knowledge and contextual information with the generated real-time sensor data streams, in order to perform complex processing tasks in a context-aware manner [1]. However, this integration of multiple data sources is challenging due to their high volume, variety and velocity [2].

Semantic IoT platforms solve the challenges associated with the real-time integration of IoT sensor data, domain knowledge and context information [3, 4]. They do this by deploying these tasks on a uniform, aggregated data model. Typically, Semantic Web technologies are employed, where this data model is represented by ontologies that formally define the application-specific concepts and their relationships and properties. Stream processing components in the IoT platform continuously evaluate semantic queries on this aggregated data model, using existing stream reasoning techniques [5].

In currently existing semantic IoT platforms, the distribution and configuration of these stream processing and stream reasoning tasks is rather static [3]. This means that the tasks are translated to semantic queries that are deployed across the network's processing components according to a static configuration, i.e., on a fixed location with a fixed set of properties. However, in contrast, the environmental context in which the tasks are deployed has a very dynamic nature. This is true for both use case specific context and the situational context. The latter includes external factors such as network properties, utilization of device resources, properties of the data stream such as the number of events that need to be processed, and the availability of device resources to the stream processing components. Hence, a static distribution and configuration of tasks is not optimal in a dynamic IoT context with a variety of tasks and nodes with varying resources, across a fluctuating network. This is illustrated in Figure 1.

To illustrate this with an example, consider a homecare monitoring use case in the healthcare domain, which includes multiple smart homes and an alarm center managing patient calls. In this example, the use case context includes both the Electronic Health Record (EHR) of patients and the location of patients in their smart homes. Both impact which monitoring tasks should be performed, but continuously evolve over time. In addition, a trade-off between cost and patient security needs to be made to optimally distribute the required monitoring tasks across the network. To achieve optimal patient security, all raw sensor data is ideally available on the central servers

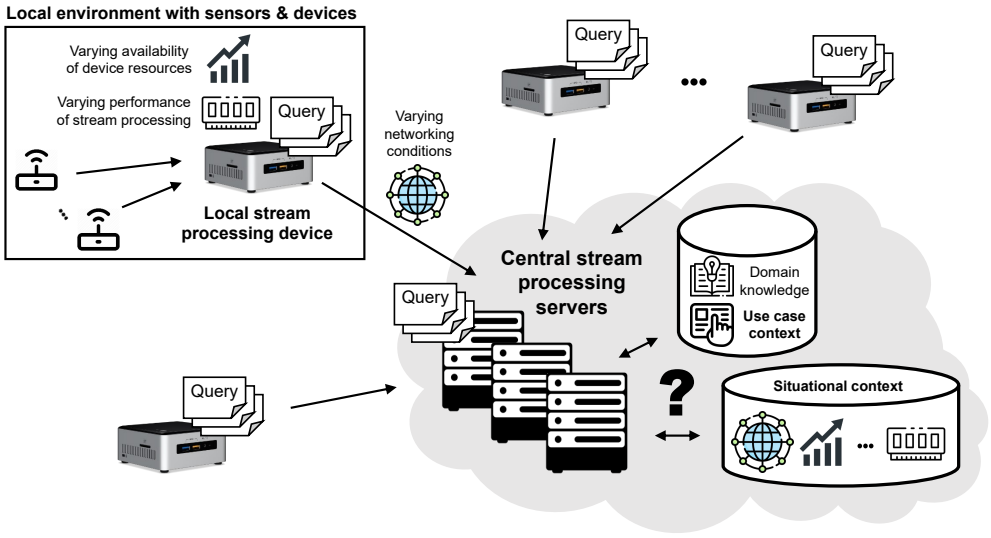


Figure 1: Illustration of the static distribution and configuration of stream processing queries across the full network in a semantic IoT platform, in a very dynamic environment. Use case context and various external situational context parameters can vary over time. This requires an adaptive, dynamic distribution and configuration of queries as well, in an efficient way. Achieving this is the main objective of this paper.

of the alarm center, in order to motivate decisions and make detailed analyses whenever needed. This requires the processing tasks to be performed centrally. However, to reduce central server costs, less delicate tasks can be executed locally as well, especially when networking conditions do not allow the efficient forwarding of all raw sensor data to the central servers. This is however only feasible if the local devices have enough available resources to perform the processing tasks. Since both network conditions and local device resource usage are situational context parameters that can heavily fluctuate, a static configuration and distribution of the homecare monitoring tasks cannot take all the given requirements into account in a dynamically evolving environment. Instead, a dynamic distribution and configuration of those tasks is required to optimally balance the presented trade-off.

Similarly for other use cases, the dynamic nature of the environmental context is not suited for a static distribution of processing tasks. In the smart cities domain, the network traffic can largely vary over time, highlighting the need for a smart adaptation of task distribution across the network. Moreover, in asset monitoring, adequate follow-up of assets is required without overdoing local device resources, preferring a dynamic configuration of processing task properties such as their execution frequency.

These different examples of various IoT application domains address the key need for the dynamic configuration and distribution of stream processing tasks across the network in a semantic IoT platform. First of all, only relevant tasks should be deployed at all times. Ideally, the location of the deployed individual semantic tasks (queries) can be adaptively shifted between central devices and local & edge devices. Moreover, the

semantic queries should also have dynamic properties such as their execution frequency and the size of the data window on which they are executed. All aforementioned decisions should be made dynamically based on the environmental context in which the tasks are deployed. Importantly, as the examples also illustrate, it can differ for different use cases what parameters of the situational context exactly influence this task configuration and distribution, and how.

Hence, in summary, the main research objective of this paper is to design a semantic IoT platform component that can be deployed in a semantic IoT architecture to dynamically define, distribute and configure the relevant stream processing tasks based on the environmental context. In other words, the designed component will dynamically decide which tasks are executed across the network, where in the network, and how. It will achieve this by monitoring both the situational and use case context. More specifically, we set the following research objectives for the design of this component:

1. The component should enable the monitoring of various situational context parameters such as the network characteristics, resource usage on the stream processing devices, data stream properties and real-time performance of the stream processing components.
2. The component should allow dynamically updating the location of stream processing queries across the IoT network and the window parameters of the stream processing queries (execution frequency, size of data window), based on the monitored situational context.
3. The component should allow dynamically configuring for each individual use case how the situational context influences the location and/or window parameters of the deployed stream processing queries, in order to optimally balance use case specific trade-offs and achieve efficient stream processing.
4. The component should have a generic design that easily allows monitoring additional properties of the situational context.
5. The component should ensure that only the relevant stream processing queries are deployed on all components of the IoT network at all times, based on existing domain knowledge and the current use case context.

The remainder of this paper is structured as follows. Important background information is provided in Section 2. Section 3 presents the full methodology of the research that endeavors to achieve the research objectives of this work. Details on our implementation of this methodology are given in Section 4. Moreover, relevant related work for this paper is presented in Section 5. Sections 6 and 7 describe the set-up and results of the performed evaluations. Section 8 further discusses the evaluation results. Finally, Section 9 concludes the main findings of the paper and highlights future work.

2 Background

The presented IoT platform component builds further on DIVIDE, which is the result of previous research in our research group [6, 7]. DIVIDE itself is built on existing Semantic Web technologies. This section discusses background information on both aspects that is relevant to the remainder of this paper.

2.1 Semantic Web technologies

The Resource Description Framework (RDF) [8] and the Web Ontology Language (OWL) [9] are two existing standards that allow modeling heterogeneous data sources in a uniform, aggregated data model using ontologies [4]. RDF data is represented as a graph of triples, where every triple connects a subject to an object with a predicate. Different data formats exist to express and store RDF data. Popular examples are RDF/Turtle and N-Triples. Similarly, Manchester syntax is a compact syntax to represent OWL 2 ontologies. The SPARQL Protocol and RDF Query Language (SPARQL) can be used to write and evaluate queries on RDF data [10].

Semantic reasoning is a technique to derive new knowledge from a set of asserted facts and axioms defined in ontologies. Different OWL 2 language profiles exist [11]. Every profile gives an OWL 2 ontology a level of expressivity to define axioms that can be used by a semantic reasoner. The OWL 2 RL profile allows defining axioms that can be evaluated by a rule engine.

Stream reasoning focuses on adopting semantic reasoning techniques for streaming data [5]. RDF Stream Processing (RSP) engines such as C-SPARQL continuously process RDF data streams by evaluating RSP queries [12, 13]. These queries are evaluated on a data window that is put on the data streams. Based on the window parameters of the data windows defined in the query, a window is triggered at specific times to evaluate the query. The window parameters include the size and (sliding) step of the window. The latter defines the period between data window triggers and thus the query execution frequency. RSP-QL is used within DIVIDE for representing RSP queries, as it is a reference model that unifies the semantics of existing RSP approaches [14]. Finally, cascading reasoning is an approach that allows expressive semantic reasoning over high-velocity data streams by introducing a processing hierarchy of reasoners [15–17].

2.2 DIVIDE

DIVIDE is a semantic component that can automatically and adaptively derive and manage the relevant queries for the stream processing components in an IoT platform [6]. It does this in a context-aware way, by monitoring the use case context relevant to the different components in the network. Whenever DIVIDE observes a change to the use case context that is relevant to a specific component, it derives the stream processing queries that are contextually relevant given the updated use case context. DIVIDE performs semantic reasoning on the current use case context to derive the relevant queries. This way, DIVIDE ensures that no more real-time reasoning is required while evaluating the resulting stream processing queries. Hence, these queries can be efficiently executed in comparison with real-time reasoning approaches, also on low-end IoT devices with few resources. Moreover, by managing the stream processing queries in an automated, adaptive and context-aware way, it reduces the manual, labor-intensive effort required to (re)configure those queries.

In its methodological design, DIVIDE uses the rule-based Notation3 (N3) Logic [18], which is a superset of RDF/Turtle [8]. Hence, the semantic reasoner used by DIVIDE supports N3 and can reason within the OWL 2 RL reasoning profile. Moreover, DIVIDE uses the concepts of a DIVIDE component and a DIVIDE query.

A DIVIDE component is an entity in the IoT network on which a single RSP engine runs. Every DIVIDE component is linked to several named graphs in the use case context. Updates to this relevant use case context result in a new DIVIDE query derivation for that component, for all DIVIDE queries registered to the system. A DIVIDE query is a generic template definition of an RSP query that should perform a real-time processing task on the RDF data streams generated by the different local components in the system. The internal representation of a DIVIDE query contains a generic RSP-QL query pattern that typically uses generic ontology concepts in its subparts to allow representing multiple possibly contextually relevant tasks at once. Moreover, the internal representation includes multiple semantic rules that are used by the rule reasoner during the query derivation to construct the resulting RSP queries for the DIVIDE component's RSP engine. The first rule is the goal, which defines the semantic output that should be filtered by the resulting RSP queries. The sensor query rule is the main rule of a DIVIDE query and contains a semantic definition of input variables and static window parameters. The input variables are all variables in the generic RSP-QL query pattern that are dependent on the use case context, while the static window parameters can either have default values or be dependent on the use case context as well. During the query derivation, both sets of parameters are substituted in the RSP-QL query pattern for every set of relevant values, based on the current use case context. Hence, DIVIDE also allows updating the window parameters (window size and sliding step) of the deployed stream processing queries.

The DIVIDE query derivation process for a given combination of a DIVIDE component and DIVIDE query consists of different sequentially executed steps. First, the updated context relevant to that component is enriched by executing any defined context-enriching queries on the data model (step 1). These queries are part of the definition of a DIVIDE query, and can extend the context with additional triples. These triples can include dynamic window parameters, which are prioritized in the substitution over static window parameters. Consequently, semantic reasoning is performed on the enriched context and domain knowledge to construct a proof that contains the details of the derived queries and how to instantiate them (step 2). Next, the derived queries are extracted from the proof (step 3) and the instantiated input variables of these derived queries are substituted into the generic RSP-QL query pattern of the DIVIDE query (step 4). Moreover, window parameters are substituted in a similar way (step 5). This window parameter substitution step is split in two parts: first, dynamic window parameters are substituted, *if* present in the enriched context. Second, static window parameters are substituted for only those window parameter variables that have not yet been substituted. Finally, the resulting RSP-QL queries are translated to the query language of the DIVIDE component's RSP engine and the active, registered RSP queries on this local RSP engine are updated (step 6).

Looking at the research objectives of this work outlined in Section 1, it is clear that the first version of DIVIDE resulting from previous research already solves research objective 5: it already derives and manages the RSP queries in an adaptive and context-aware way, based on domain knowledge and use case dependent context. However, DIVIDE cannot yet monitor the situational context and leverage these

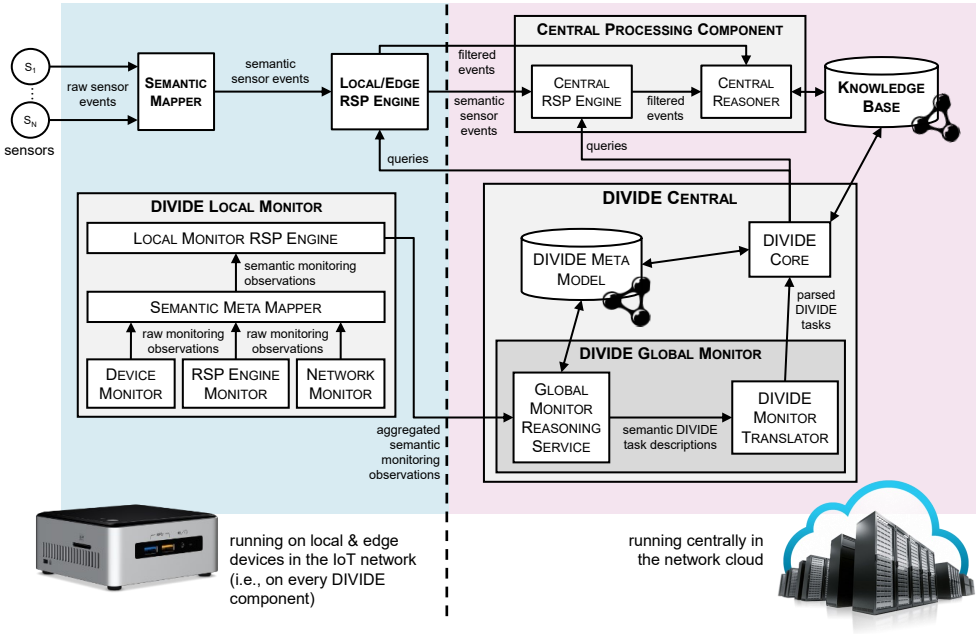


Figure 2: Overview of the different subcomponents of DIVIDE in the architecture of a typical cascading reasoning set-up in an IoT network

monitored properties to manage the configuration and distribution of the stream processing queries across the IoT network in an intelligent, dynamic, use case specific way. Moreover, DIVIDE currently always deploys RSP queries associated to a DIVIDE component in the IoT network on the RSP engine that is running on this component's device. It can adaptively update the registered queries, but cannot update the location of the queries by for example moving queries between edge and cloud. Hence, this research focuses on an updated version of DIVIDE, where its design and implementation is updated and extended to fulfill the research objectives of this work.

3 Methodology

To achieve the main research objectives of this paper, the design of DIVIDE is updated to allow performing automated monitoring of the situational context in which semantic queries are deployed across the IoT network, and to allow defining use case specific rules that automatically update the window parameters or distribution of the deployed queries across the network. This section zooms in on the design of DIVIDE by first presenting the overall cascading architecture in which the different subcomponents of DIVIDE are deployed. Moreover, the most important methodological details of these subcomponents are further discussed.

3.1 Monitoring architecture

Figure 2 provides an overview of the overall architecture of a typical cascading reasoning set-up in an IoT network, in which DIVIDE should be deployed. This architecture is split up in two parts: a central part with components that run centrally in the cloud, and a local part containing components that are deployed on local or edge devices of the IoT network. The central part contains the Central Processing Component, the Knowledge Base and DIVIDE Central. The local part contains the DIVIDE Local Monitor, as well as a Semantic Mapper and a Local (or Edge) RSP Engine. In a typical IoT network, multiple devices exist that contain the local components. In the context of DIVIDE, there is one set of local components for every DIVIDE component. In contrast, there is only one set of central components.

Centrally, the Knowledge Base contains the semantic representation of all domain knowledge and use case context in the system. This data is semantically stored in an RDF-based knowledge graph. Moreover, the components can be clearly split in two groups: the components in the semantic data processing flow (Semantic Mapper, Local/Edge RSP Engine, Central Processing Component) and the subcomponents of DIVIDE (DIVIDE Central, DIVIDE Local Monitor). Both groups are discussed in the following subsections.

3.1.1 Semantic data processing flow

The upper part of Figure 2 demonstrates how the data flows through the different components, following a cascading reasoning approach. On every DIVIDE component, different sensors generate raw sensor events. These observations are semantically annotated and forwarded as semantic RDF events to the data streams that are registered to the Local RSP Engine. Depending on the query distribution for the corresponding DIVIDE component, the Local RSP Engine can perform different tasks. This distribution is managed by DIVIDE. For every DIVIDE query, the Local RSP Engine can either continuously evaluate the RSP queries derived from that DIVIDE query, or forward the semantic sensor events on the streams to the Central Processing Component. The latter applies whenever DIVIDE decides that the RSP queries derived from a DIVIDE query should be deployed centrally. In that case, the Central RSP Engine receives all semantic sensor events forwarded by the Local RSP Engine on its data streams, and continuously evaluates the derived RSP queries.

The Local RSP Engines of the different DIVIDE components and the Central RSP Engine all forward the filtered events in the RSP query outputs to the Central Reasoner. This Central Reasoner is the final component in the semantic data processing flow. It is responsible for further processing these events and acting upon them, depending on the use case requirements. To do so, it can interact with all domain knowledge and contextual data in the Knowledge Base. Moreover, the Central Reasoner can also update any relevant use case context in the Knowledge Base.

3.1.2 DIVIDE subcomponents

The updated design of DIVIDE contains multiple subcomponents. On every DIVIDE component in the platform, a DIVIDE Local Monitor is deployed. Moreover, on the

central server, the DIVIDE Central component is active, which consists of three main entities: DIVIDE Core, the DIVIDE Meta Model and the DIVIDE Global Monitor.

The DIVIDE Core component represents the first version of DIVIDE resulting from our previous research, as discussed in Section 2.2. Hence, it is responsible for monitoring contextual changes in the Knowledge Base and triggering the query derivation whenever changes relevant to a DIVIDE component are observed. In addition, the design of DIVIDE Core is updated to allow it to modify the distribution of queries and configuration of query window parameters through DIVIDE tasks forwarded by the DIVIDE Global Monitor. Hence, DIVIDE Core is responsible for managing the queries of both the Central RSP Engine and all Local RSP Engines in the IoT platform.

The DIVIDE Meta Model is an additional internal RDF-based knowledge graph maintained by DIVIDE Core. It contains the Meta Model ontology that allows modeling all relevant meta-information about DIVIDE. This meta-information includes the different DIVIDE components and DIVIDE queries in the system, and the current configuration and distribution of the RSP queries across the network. Moreover, the Meta Model ontology enables the monitoring subcomponents of DIVIDE to model all monitoring observations. The aggregation of all this data in the Meta Model is essential in the design of DIVIDE, since it allows using the monitoring information to manage the configuration and distribution of RSP queries, taking the current configuration and distribution together with the other meta-information into account.

On every DIVIDE Component, a DIVIDE Local Monitor is deployed to perform the actual monitoring of the situational context. This DIVIDE Local Monitor contains multiple individual monitors that each continuously monitor specific parameters on the given component. In its current design, DIVIDE supports the monitoring of network properties through the Network Monitor, device resource usage and availability through the Device Monitor, and characteristics and performance of the Local RSP Engine through the RSP Engine Monitor. All monitoring observations are semantically annotated using the Meta Model ontology, aggregated by the Local Monitor RSP Engine, and forwarded to the DIVIDE Global Monitor.

Finally, the DIVIDE Global Monitor consists of a Global Monitor Reasoning Service that processes all aggregated monitoring observations received from the different DIVIDE Local Monitors. By evaluating use case specific Global Monitor queries on the DIVIDE Meta Model enriched with this monitoring data, it decides how the distribution of RSP queries and the configuration of the queries' window parameters is adaptively altered. These decisions are outputted in the form of semantic tasks descriptions, which are parsed by the DIVIDE Monitor Translator to actual tasks executable by DIVIDE Core.

3.2 DIVIDE Meta Model

The DIVIDE Meta Model is an RDF-based knowledge graph that contains the Meta Model ontology. This ontology is an OWL ontology consisting of two main modules: `DivideCore` and `Monitoring`. `DivideCore` contains all constructs that allow modeling meta-information about DIVIDE, while `Monitoring` allows representing the properties monitored by the DIVIDE Local Monitor and their observations. The `Monitoring` module builds further on the `DivideCore` module by importing it. This section zooms

Listing 1: Overview of all prefixes used in the listings with semantic content in this paper, and in the DIVIDE Meta Model ontology overview in Figure 3

```
# DIVIDE Meta Model ontology modules
@prefix divide-core:
        <https://divide.idlab.ugent.be/meta-model/divide-core/> .
@prefix monitoring:
        <https://divide.idlab.ugent.be/meta-model/monitoring/> .

# existing, imported ontologies
@prefix saref-core: <https://saref.etsi.org/core/> .
@prefix om: <http://www.ontology-of-units-of-measure.org/resource/om-2/> .

# generic prefixes
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix owl: <http://www.w3.org/2002/07/owl#> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .
```

in on both ontology modules by discussing the concepts and relationships defined in the modules, highlighting imported existing ontologies, and explaining how meta-information and monitoring data can be represented in the Meta Model.

Figure 3 presents an overview of the DIVIDE Meta Model ontology. It shows the most important classes and properties in `DivideCore` and `Monitoring`, as well as how the concepts are linked to existing ontologies. For reference purposes, Listing 1 gives an overview of all prefixes used in this figure and in all listings with semantic content in this paper.

3.2.1 Imported ontologies

The DIVIDE Meta Model reuses concepts from existing, well-known ontologies as much as possible. To this end, multiple such existing ontologies are imported by the modules of the Meta Model ontology.

First, the Meta Model ontology imports the existing Smart Applications REFerence (SAREF) ontology [19]. This ontology is an ETSI standard that already defines multiple concepts in the smart applications domain and their relationships and properties. This includes the concept of devices, observable properties, and measurements of those properties in relation to certain features of interest. The latter is especially relevant for the `Monitoring` module.

Second, the Meta Model ontology imports the Ontology of units of Measure [20]. This is an OWL ontology describing the full domain of quantities and units of measure. By integrating this ontology with SAREF in the `Monitoring` module, the quantitative monitoring results can be easily described.

Moreover, several other existing models and ontologies were used as inspiration when designing the overall Meta Model ontology structure. These are discussed in Section 5 as related work.

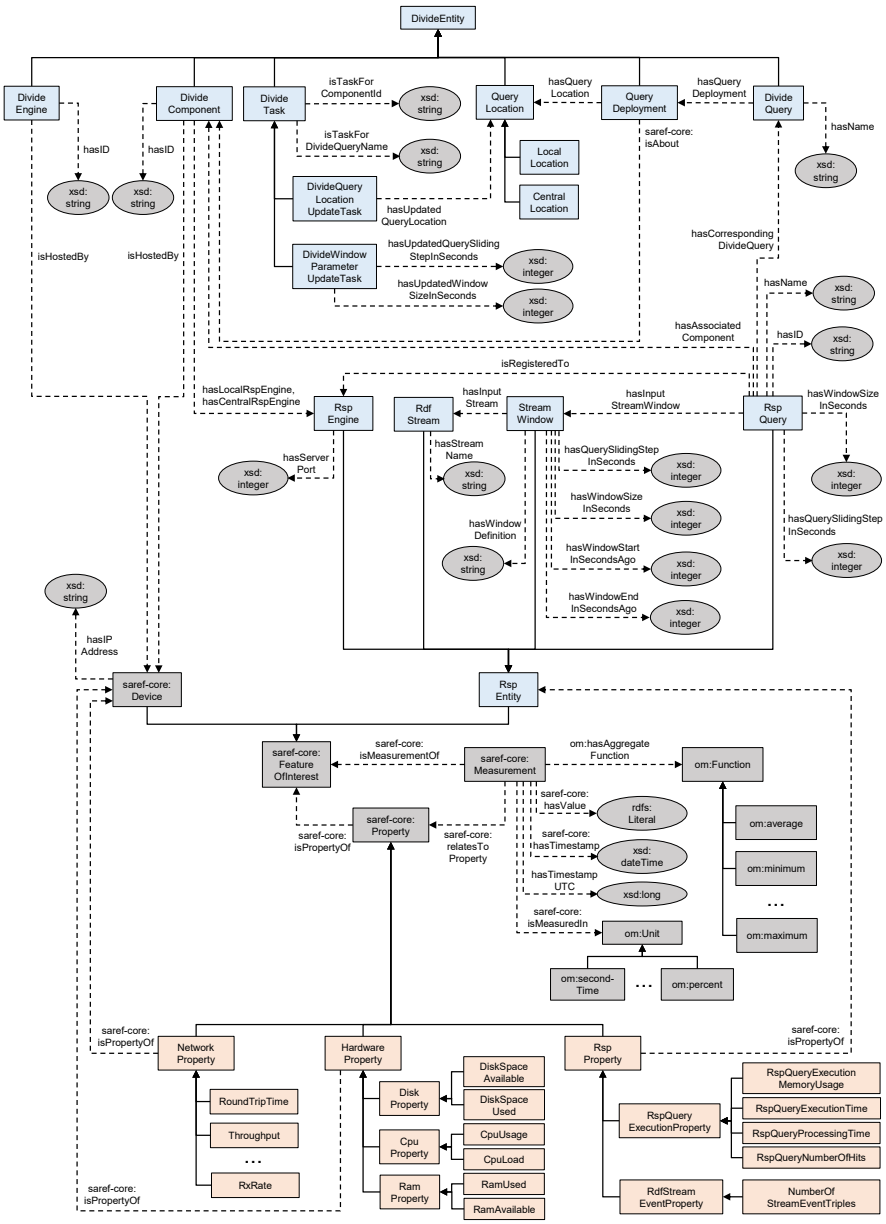


Figure 3: Overview of the main concepts in the DIVIDE Meta Model ontology. Items in blue are part of the `DivideCore` module, items in orange are part of the `Monitoring` module, and items in gray are part of imported ontologies. Rectangles represent ontology classes, circles represent literals of the given datatype. Dashed arrows represent object or data properties, the property name is added to each dashed arrow (without prefix for properties in the `DivideCore` and `Monitoring` modules, with prefix for imported ontologies). Full arrows indicate inheritance between classes.

3.2.2 `DivideCore` ontology module

`DivideCore` contains all concepts needed to model relevant meta-information about the IoT platform in which DIVIDE is deployed. This includes all properties and relations between DIVIDE entities, as well as the configuration and distribution of RSP queries across the different RSP engines in the network.

To achieve this, `DivideCore` uses the `DivideEntity` and `RspEntity` classes. As shown in Figure 3, there are multiple subclasses of `DivideEntity`. In the Meta Model, there is always one `DivideEngine` and typically multiple instances of the `DivideComponent` and `DivideQuery` class.

To keep track of the distribution of `RspQuery` instances across the network, the `QueryDeployment` concept is used. All instances of `RspQuery` that originate from the same `DivideQuery` and are associated to the same `DivideComponent`, have a single `QueryDeployment` instance. This `QueryDeployment` is linked to a `QueryLocation`, which can either be a `LocalLocation` or `CentralLocation`. This represents whether the associated `RspQuery` instances are deployed on the local or central `RspEngine` associated to the `DivideComponent`.

To model the configuration of an `RspQuery`, multiple other subclasses of `RspEntity` exist. An `RspQuery` has one or more `StreamWindow` instances defined as input stream window. Such a `StreamWindow` is linked to an `RdfStream`, and has a certain window definition. This window definition string contains the actual window parameters of the `StreamWindow`. Every `StreamWindow` has a query sliding step, while the description of the other window parameters depends on the exact window definition string. For example, a window definition `RANGE PT60S STEP PT10S` in RSP-QL translates to value 10 for the `hasQuerySlidingStepInSeconds` property of the `StreamWindow` and value 60 for `hasWindowSizeInSeconds`, while the stream window with window definition `FROM NOW-PT20M TO NOW-PT10M STEP PT2M` leads to value 1200 for `hasWindowStartInSecondsAgo`, value 600 for `hasWindowEndInSecondsAgo`, and value 120 for `hasQuerySlidingStepInSeconds`. Moreover, whenever an `RspQuery` has only one input `StreamWindow`, the values of the `hasWindowSizeInSeconds` and `hasQuerySlidingStepInSeconds` properties of this `StreamWindow` are also linked to `RspQuery` using the same properties. For conformity and uniformity, all window parameters are modeled in seconds.

In the methodological design of DIVIDE, all relevant meta-information of DIVIDE is continuously kept up-to-date in the DIVIDE Meta Model, using the aforementioned concepts in the Meta Model ontology. As this meta-information continuously evolves throughout the runtime of DIVIDE in an IoT platform set-up, one DIVIDE subcomponent is responsible for updating it whenever changes occur. This is DIVIDE Core, as it represents the core subcomponent of DIVIDE that manages the registered DIVIDE queries, DIVIDE components, and derived RSP queries for these DIVIDE components.

Appendix A illustrates with an example how the relevant meta-information of DIVIDE is stored as semantic triples in the DIVIDE Meta Model using the presented concepts of the `DivideCore` ontology module.

Finally, the `DivideCore` ontology module also contains the `DivideTask` class. This class is used by the DIVIDE Global Monitor in the output of the Global Monitor

queries to semantically describe the tasks for the DIVIDE engine to update the distribution (query deployment) of the RSP queries associated to a certain DIVIDE component and DIVIDE query, or the configuration of the window parameters of these queries.

3.2.3 `Monitoring` ontology module

The `Monitoring` module of the DIVIDE Meta Model ontology is an extension of the `DivideCore` module. It specifically focuses on how the situational context in which DIVIDE operates can be semantically described.

The module heavily uses existing concepts in the imported SAREF ontology to represent individual and aggregated monitoring observations. Every such observation is an instance of the `saref-core:Measurement` class. An instance of `saref-core:Measurement` is related to a certain `saref-core:Property`, and is measured for a specific `saref-core:FeatureOfInterest`. Through the ontology definition, every subclass of `saref-core:Property` can be linked to a specific subclass of `saref-core:FeatureOfInterest` to ensure that measurements of the property are always linked to the given feature of interest type. Moreover, a `saref-core:Measurement` has a value, a timestamp (both as a string and a UTC millisecond timestamp), and a unit represented with the Ontology of units of Measure. Finally, a measurement can also be defined as an aggregate measurement by linking it to an aggregate `om:Function` such as `om:average`.

The `Monitoring` module defines several subclasses of `saref-core:Property`: `NetworkProperty` for network characteristics, `HardwareProperty` for resource usage and availability of the hardware of devices, and `RspProperty` for specific RSP engine characteristics. Several subclasses of those classes are defined as well. Note that this list could be easily extended with other properties that are also of relevance to be monitored. Moreover, the ontology module also defines that every measurement related to a `NetworkProperty` and `HardwareProperty` is always linked to a `saref-core:Device` as a feature of interest. This will be the device on which the Local Monitor RSP Engine and thus also the Local Monitor is active. For an `RspProperty` measurement, the feature of interest of is always an `RspEntity`. More specifically, stream characteristics (`RdfStreamEventProperty` measurements) and RSP query performance measures (`RspQueryExecutionProperty` measurements) collected by the RSP Engine Monitor are linked to the corresponding RDF streams and RSP queries, respectively.

Appendix A presents an example of how a monitoring observation can be semantically described with the concepts of the `Monitoring` ontology module.

3.3 DIVIDE Local Monitor

The DIVIDE Local Monitor performs the actual monitoring of the situational context in which RSP queries are deployed across the IoT network. On every DIVIDE component registered to DIVIDE, a DIVIDE Local Monitor is deployed. A DIVIDE Local Monitor consists of multiple subcomponents: several individual monitors, a Semantic Meta Mapper, and a Local Monitor RSP Engine. The following subsections discuss their design.

3.3.1 Individual monitors

The design of the DIVIDE Local Monitor deliberately decouples the individual monitors from the semantic components. This way, the individual monitors can be implemented independently. This modular design allows easily updating or replacing the implementation of the individual monitors, without having to modify any of the other components.

The current methodological design of the DIVIDE Local Monitor includes three individual monitors: a Network Monitor, a Device Monitor and an RSP Engine Monitor. They all monitor important situational context information that is of relevance in multiple IoT application domains when deciding on the distribution of RSP queries across an IoT network and the configuration of their window parameters, as indicated by the examples in Section 1.

The purpose of the Network Monitor is to keep track of relevant characteristics about the available network. This is relevant for the query distribution across the network, so that the data flow over the network balances the use case specific requirements with the networking conditions. Therefore, the network monitor should focus on the characteristics of the networking link that connects the DIVIDE component's device (i.e., the device on which the Local Monitor is running) with the central server on which the Central Processing Component is deployed. Potentially relevant networking properties to be monitored include network round-trip time (RTT), throughput, latency, available bandwidth, incoming networking packets received vs. dropped, outgoing networking packets sent vs. dropped, delay, jitter, etc.

The goal of the Device Monitor is to analyze the used and available resources of the local or edge device on which the Local Monitor is running. This can be important information in a use case to decide whether specific RSP queries can be deployed on the Local RSP Engine, or whether they should be moved to the Central RSP Engine. For example, RSP queries operating on large data windows might require a certain quantity of Random Access Memory (RAM) to be available, while other queries with a high execution frequency require a low average Central Processing Unit (CPU) load. Thus, relevant device resource properties to monitor include current CPU usage (either per individual CPU core or overall), CPU average load over a certain time period, used vs. available physical RAM and swap memory, used vs. available disk storage, and possibly others.

Finally, the RSP Engine Monitor is included in the design of the Local Monitor to monitor data stream characteristics and the performance of the continuous query execution on the Local RSP Engine. Stream characteristics include the number of triples per stream event or the number of triples sent on the data stream per time unit. Examples of possibly relevant performance metrics are the execution time of RSP queries, the processing time of these queries (i.e., the time from the query's data window trigger until the generation of the query result), the amount of RAM used by the query execution, and the number of query results. Similarly to the other individual monitors, these different monitored properties can influence whether the query window parameters should be modified, or whether a query should be moved to the Central RSP engine. For example, if the processing time of a query exceeds the period between

two query executions (i.e., the query's sliding step), the query execution frequency and/or data window size might need to be lowered.

3.3.2 Semantic Meta Mapper and Local Monitor RSP Engine

The design of the DIVIDE monitoring subcomponents, and thus also the DIVIDE Local Monitor, is built upon Semantic Web technologies. Therefore, the DIVIDE Local Monitor contains two semantic components: the Semantic Meta Mapper and the Local Monitor RSP Engine.

In the data flow of monitoring observations, the individual monitors forward the raw monitoring observations to the Semantic Meta Mapper. This component is a general semantic mapper that uses the `Monitoring` module of the DIVIDE Meta Model ontology to semantically annotate these raw monitoring observations. These observations are described as measurements of the monitored property, as explained in Section 3.2.3.

The Semantic Meta Mapper forwards all semantic monitoring observations to the Local Monitor RSP Engine. The purpose of this RSP engine is to aggregate the different monitoring observations. Possible aggregations are averaging and taking the maximum or minimum value. These aggregations are performed by continuously evaluating one or more aggregation queries and sending the results of these queries to the central Global Monitor. An example of such an aggregation is shown in Listing 9 of Appendix A. It is important to note that the aggregation queries are simple filtering queries: they are executed on a data model that only contains the individual semantic monitoring observations, and require no semantic reasoning during their evaluation.

The rationale behind only sending aggregations to the Global Monitor is twofold. First and foremost, this approach limits and controls the number of events sent over the network. This ensures that the Local Monitor will not further stress the network too much in case of congestion. Second, individual outliers in the observations are leveled out by some aggregations such as averaging. This way, a broader view in time on the monitored observations can be considered by the Global Monitor in its decision making, avoiding constant changes in the query distribution and configuration when this is not desirable.

3.4 DIVIDE Global Monitor

The DIVIDE Global Monitor is a subcomponent of DIVIDE that is deployed on the central server, together with the DIVIDE Meta Model and DIVIDE Core. It is the actuator of DIVIDE, as it is responsible for making actual decisions that update the distribution or window parameter configuration of the RSP queries deployed across the IoT network. To do this, it intelligently processes the aggregated semantic monitoring observations received from the Local Monitor instances in the platform.

The DIVIDE Global Monitor consists of two subcomponents: the Global Monitor Reasoning Service, and the DIVIDE Monitor Translator. In short, the Global Monitor Reasoning Service continuously executes Global Monitor queries to decide which tasks to update the RSP query distribution or configuration should be performed. These tasks will be semantically described in the output of the Global Monitor queries. In the

current methodological design of DIVIDE, two concrete tasks can be defined: a task to update the location in the network where queries are being executed (distribution update), and a task to update the window parameters of RSP queries (configuration update). The DIVIDE Monitor Translator parses the semantic task descriptions outputted by the Global Monitor queries and translates them to tasks that can then be executed by DIVIDE Core.

This section zooms in on the details of the DIVIDE Global Monitor. First, the general design of the Global Monitor Reasoning Service and the concept of Global Monitor queries is further detailed. Consequently, the two query distribution and configuration update tasks in the design of DIVIDE are discussed. Finally, a user-friendly grammar to specify the Global Monitor queries through actuation rules is presented.

3.4.1 Global Monitor Reasoning Service

The Global Monitor Reasoning Service is a stream-based reasoning service that combines rule reasoning with the continuous evaluation of stream processing queries. It maintains a data stream that is continuously receiving the aggregated monitoring observations from the different Local Monitor instances deployed on every DIVIDE component in the network. The reasoning service works with a tumbling window of a configured interval: every interval, a data model is constructed that contains all aggregated monitoring observations in the window. This data model is temporarily added to the DIVIDE Meta Model, which contains an up-to-date version of all relevant meta-information of DIVIDE. This is maintained by DIVIDE Core, as explained in Section 3.2.2. This way, the DIVIDE Meta Model contains an aggregated view on this meta-information, combined with the current monitoring data. The Global Monitor Reasoning Service then performs rule reasoning with an OWL 2 RL reasoner on this aggregated data model, and executes all activated Global Monitor queries in a defined order. Query outputs are forwarded to the DIVIDE Monitor Translator, after which the window of monitoring observations is removed from the DIVIDE Meta Model and OWL 2 RL reasoning is again performed.

The Global Monitor queries that are activated on the Global Monitor Reasoning Service need to be configured by the end user of DIVIDE. This is a deliberate design choice, since the conditions of how the system should update the distribution and window parameter configuration of the system, can be very use case specific, as discussed in Section 1. Through the design of DIVIDE an end user can define a Global Monitor query to take into account any information that is present in the DIVIDE Meta Model. This includes information about all devices and components in the network, the current configuration and distribution of RSP queries, and any information about the situational context that is captured by the Local Monitors.

A Global Monitor query should be defined by the end user as a regular SPARQL query. To take into account all information in the DIVIDE Meta Model, the concepts of the Meta Model ontology should be used. Similarly, to define the concrete tasks in the output of a Global Monitor query, the `DivideTask` class of the `DivideCore` ontology module can be used, as explained in Section 3.2.3. It is important to note that by design, such a task is always defined for a combination of a DIVIDE component and DIVIDE query. This means that a query location update task or window parameter

Listing 2: Template to specify a query location update task in the output of a Global Monitor query, for a move to the Central RSP Engine

```
[ a divide-core:DivideQueryLocationUpdateTask ;
  divide-core:isTaskForDivideQueryName ?divideQueryName ;
  divide-core:isTaskForComponentId ?componentId ;
  divide-core:hasUpdatedQueryLocation [ a divide-core:CentralLocation ] ]
```

update task is always performed for all RSP queries derived from the given DIVIDE query, for the given DIVIDE component. Details of how both types of tasks can be defined and how the design of DIVIDE Core is updated to allow performing them, are presented in the following sections.

3.4.2 Query location update task

The first task supported in the current methodological design of DIVIDE is a query location update task, which allows updating the RSP query distribution across the network. More specifically, for a given DIVIDE component, this task changes the deployment location of all RSP queries derived from a specific DIVIDE query. Two possible locations exist: local and central. The local deployment location represents the Local RSP Engine on the local or edge device. This is the device associated to the DIVIDE component, on which the Local Monitor is also deployed. The central location represents the Central RSP Engine of the Central Processing Component on the server in the cloud. In the architecture, this is typically the same device on which the Global Monitor Reasoning Service is active.

Definition in Global Monitor query output

Listing 2 specifies the template of how to define a query location task in the output of a Global Monitor query, using the `DivideCore` module of the Meta Model ontology. It includes the name and ID of the DIVIDE query and component, respectively, and the new query deployment location.

Design changes to DIVIDE Core to perform task

To support the execution of a query location update task, the design of DIVIDE Core has been extended. Performing a query location update does not start a query derivation process, since the actual RSP queries that should be evaluated have not changed. Instead, when moving all RSP queries derived from a DIVIDE query from the Local RSP Engine of a certain DIVIDE component to the Central RSP Engine, DIVIDE Core unregisters these RSP queries from the Local RSP Engine. Moreover, new uniquely identifiable data streams are registered to the Central RSP Engine for all data streams defined in the input stream windows of the original RSP queries. DIVIDE Core then instructs the Local RSP Engine to forward all data on the original streams to those new corresponding streams on the Central RSP Engine. If multiple queries have the same data streams in their input stream windows, DIVIDE Core ensures that the data of one data stream is only forwarded once, to avoid duplicate network traffic. The input stream windows in the original RSP queries are altered to

the new input streams, after which these RSP queries with updated stream windows are registered to the Central RSP Engine. Finally, DIVIDE Core ensures that the observers of the outputs of the original RSP queries on the Local RSP Engine are also registered as observers of the outputs of the corresponding new RSP queries on the Central RSP Engine. During the full process of performing the query location update task, DIVIDE Core instructs the Local RSP Engine to buffer all data on the involved data streams, to avoid data loss.

To perform the opposite task of moving RSP queries derived from a DIVIDE query from the Central RSP Engine to the Local RSP Engine of a DIVIDE component, the aforementioned actions are reverted. This means that the RSP queries are unregistered from the Central RSP Engine, the original RSP queries and their observers are registered again to the Local RSP Engine, and the data forwarding is disabled for those local data streams that do not have any associated data streams in the input stream windows of other queries on the Central RSP Engine.

3.4.3 Window parameter update task

The window parameter update task is the second task that can be defined in the output of a Global Monitor query. It allows updating the window parameters of the RSP queries that are derived from a DIVIDE query, for a specific DIVIDE component. A window parameter update task does not alter the actual content of the RSP queries or their deployed location, but only modifies the defined window parameters. In its current design, the DIVIDE Meta Model ontology supports updating either the window size of the input stream window of the derived RSP queries, the sliding step, or both at once.

Definition in Global Monitor query output

To define a window parameter update task in the output of a Global Monitor query, the `DivideCore` module of the DIVIDE Meta Model ontology should be used. The template for this definition is presented in Listing 3. Besides the name and ID to define the respective DIVIDE query and component, an updated value for the window size and/or sliding step of the input stream window of the derived RSP queries can be defined.

As explained in Section 3.2.2, if an RSP query only has a single input stream window, the window size and sliding step of this stream window are also semantically linked to the RSP query itself in the DIVIDE Meta Model. If not, no window parameters are associated to the RSP query instance in the Meta Model. Moreover, the Meta Model ontology also links the updated window parameters in the window parameter update task description to the name of the DIVIDE query, and not individually to the different stream windows that are part of the DIVIDE query's input. This is shown in the template in Listing 3. Hence, in its current design, the Meta Model ontology only supports window parameter update tasks for DIVIDE queries that only have a single input stream window in their RSP-QL query template. It is however easily possible to extend the design of DIVIDE in the future to also support queries with multiple input stream windows.

Listing 3: Template to specify a window parameter update task in the output of a Global Monitor query

```
[ a divide-core:DivideWindowParameterUpdateTask ;
  divide-core:isTaskForDivideQueryName ?divideQueryName ;
  divide-core:isTaskForComponentId ?componentId ;
  divide-core:hasUpdatedQuerySlidingStepInSeconds ?updatedSlidingStep ;
  divide-core:hasUpdatedWindowSizeInSeconds ?updatedWindowSize ]
```

Design changes to DIVIDE Core to perform task

DIVIDE Core is responsible for performing any window parameter update task that is forwarded by the DIVIDE Monitor Translator. Such a task includes a semantic data model that describes the updated window parameters as dynamic window parameters for the query. This description is identical to how dynamic window parameters can be described in the output of context-enriching queries.

To execute a window parameter update task, DIVIDE Core exploits the existing design of the query derivation process that is discussed in Section 2. This process consists of different sequential steps. Updating the window parameters of the derived RSP queries can be regarded as a query derivation in which the updated context only differs in the defined dynamic window parameters. Hence, there is no need to perform steps 1 to 4 (context enrichment, semantic reasoning, query extraction and input variable substitution) again. Instead, only step 5 and 6 need to be performed again. First, the new window parameters are substituted into the saved output of the input variable substitution step. By defining the new window parameter values received from the Global Monitor Reasoning Service as dynamic window parameters, the DIVIDE Monitor Translator ensures that those values are substituted first by DIVIDE Core. As a consequence, if a certain window parameter does not receive a new value in the output of a Global Monitor query, the original value will still be substituted instead of being overruled. As a final step in the original query derivation process, the registration of the corresponding queries is updated on the corresponding DIVIDE component's Local RSP engine or the Central RSP Engine, depending on the currently defined query location.

3.4.4 User-friendly grammar to specify actuation rules

The design of DIVIDE allows end users to define specific rules of how the distribution or window parameter configuration of RSP queries in the IoT network should be updated according to the situational context. As explained before, such actuation rules can be defined through Global Monitor queries. However, writing those SPARQL queries requires knowledge about Semantic Web technologies. In addition, an end user should also know how the concepts used within DIVIDE are semantically represented in the DIVIDE Meta Model. Hence, users for whom this is too complicated, as they do not have this knowledge or time to acquire it, might prefer a grammar-like syntax to specify the actuation rules. For this, a BNF (Backus–Naur form) grammar could be designed and used, which would then be used by an automatic parser of the Global

Listing 4: Example of a Global Monitor query that reduces the window size of all queries on a DIVIDE component with 10%, if the query is running on the component's Local RSP Engine *and* if the available RAM on the local device drops below 20%

```
CONSTRUCT {
    [ a divide-core:DivideWindowParameterUpdateTask ;
      divide-core:isTaskForDivideQueryName ?divideQueryName ;
      divide-core:isTaskForComponentId ?componentId ;
      divide-core:hasUpdatedWindowSizeInSeconds ?minUpdatedWindowSize ]
}
WHERE {
    { SELECT ?componentId ?divideQueryName
             (MIN(?updatedWindowSize) AS ?minUpdatedWindowSize)
      WHERE {
          ?device a saref-core:Device ;
                  divide-core:hosts ?component .
          ?component a divide-core:DivideComponent ;
                     divide-core:hasID ?componentId ;
                     divide-core:hasLocalRspEngine ?rspEngine .
          ?rspEngine divide-core:hasRegisteredQuery ?rspQuery .
          ?rspQuery divide-core:hasCorrespondingDivideQuery ?divideQuery ;
                    divide-core:hasAssociatedComponent ?component ;
                    divide-core:hasWindowSizeInSeconds ?windowSize .
          ?divideQuery divide-core:hasName ?divideQueryName .

          ?measurement a saref-core:Measurement ;
                       saref-core:hasValue ?avgRamAvailablePercentage ;
                       om:hasAggregateFunction om:average ;
                       saref-core:isMeasuredIn om:percent ;
                       saref-core:relatesToProperty [
                           a monitoring:RamAvailable ] ;
                       saref-core:isMeasurementOf ?device .

          FILTER (?avgRamAvailablePercentage < xsd:float(20))

          BIND(xsd:integer(FLOOR(xsd:integer(?windowSize) *
                                 xsd:float(0.9))) AS ?updatedWindowSize)
      }
      GROUP BY ?componentId ?divideQueryName }
}
```

Monitor to automatically translate the actuation rule into the appropriate SPARQL query for the Global Monitor Reasoning Service.

As an example, consider a Global Monitor query that reduces the window size of all queries on a DIVIDE component with 10%, if the query is running on the component's Local RSP Engine *and* if the available RAM on the local device drops below 20%. Listing 4 presents this actuation rule as an actual Global Monitor SPARQL query. Note that, for every DIVIDE query, it calculates the window size reduction from the smallest window size of all RSP queries derived from this DIVIDE query. A mock-up example of how this actuation rule could be represented with such a BNF grammar is shown in Listing 5.

4 Implementation

To implement the subcomponents in the methodological design of DIVIDE, we have updated our original implementation of DIVIDE [6]. This original implementation

Listing 5: Mock-up example of how the Global Monitor query in Listing 4 could be represented as an actuation rule with a BNF grammar

```
IF DIVIDE_QUERY(?query) = ?divide_query
   AND COMPONENT(?divide_query) = ?component
   AND AVG(RAM_AVAILABLE_PERCENTAGE(DEVICE(?component))) < 20
THEN UPDATE(?divide_query, ?component,
            {"WINDOW_SIZE": FLOOR(WINDOW_SIZE(?query) * 0.9)})
```

includes different modules that together compose the DIVIDE Core component on Figure 2. Our updated version of the DIVIDE implementation includes an update to our implementation of the DIVIDE Core module, and an implementation of the DIVIDE Local Monitor and the DIVIDE Global Monitor. This section specifies some details of this implementation.

4.1 DIVIDE Local Monitor

The DIVIDE Local Monitor is implemented as an executable Java JAR. This way, it can be independently started on every DIVIDE component.

4.1.1 Configuration of the DIVIDE Local Monitor

The DIVIDE Local Monitor should be configured using a JSON file. It defines the ID of the DIVIDE component on which the monitor is running, which individual monitors need to be activated, the URL of the Global Monitor Reasoning Service to which aggregated measurements should be sent, and some specific properties relevant to the individual monitors. An example of a JSON configuration of the DIVIDE Local Monitor is provided in Appendix B.

4.1.2 Implementation of the Local Monitor RSP engine

The Local Monitor RSP Engine is implemented with the C-SPARQL RSP engine [13]. A C-SPARQL aggregation query is deployed that is executed every 20 seconds on a window of 60 seconds on the monitoring data stream, to which all semantic monitoring observations are sent by the Semantic Meta Mapper. This query calculates the average, minimum & maximum of all measured properties, and sends them to the API of the Global Monitor Reasoning Service. This aggregation query is presented in Appendix B.

4.1.3 Implementation of the individual monitors

The current Local Monitor implementation includes a first version of the Network Monitor, Device Monitor and RSP Engine Monitor. All individual monitors continuously forward their observations as JSON messages to a generic monitor observer. This observer forwards every received observation to the Semantic Meta Mapper. To correctly link the monitoring observations to their associated features of interest, the implementation ensures that the individual monitors work with the same IDs of devices and RSP entities as the implementation of the DIVIDE Meta Model and

DIVIDE Core. This is achieved by the DIVIDE Global Monitor, which manages the configuration and state of the individual Local Monitor instances.

Network monitor

The Network Monitor is implemented as a Python script. This implementation serves as a Proof-of-Concept (PoC) that monitors the networking conditions in two ways.

First, the script manages a Bash `ping` process that sends a ping message (echo request) every second to the central server on which the Central Processing Component is running. This way, it continuously measures the RTT of sending a message from the DIVIDE component's device to the central server.

Second, the script uses the cross-platform `psutil` library [21] to monitor system-wide network I/O statistics over 5 second intervals. These statistics include the network Tx and Rx rate (rate of transmitted and received data), the number of packets sent and received, and the number of dropped packets. They are monitored on the network interface that is used by the DIVIDE component's device to communicate with the central server.

Device Monitor

The Device Monitor is implemented as a Python script that uses the cross-platform `psutil` library [21]. The script is called every 5 seconds and measures CPU usage and load, memory usage and disk space usage.

RSP Engine Monitor

The implementation of the RSP Engine Monitor consists of two parts: a general part included in the DIVIDE Local Monitor JAR implementation, and an external RSP engine specific part.

The RSP engine specific part is required to extract the relevant monitoring information of the semantic data streams and continuous query executions. To facilitate this, we have implemented an RSP engine wrapper that also includes the implementation of the RSP engine's server API. This wrapper is based on the existing RSP Service Interface for C-SPARQL [22]. The wrapper hosts a WebSocket server on which relevant monitoring information is sent as JSON messages. In our implementation, two types of JSON messages are posted: stream events and query executions. A stream event contains the number of triples posted on a certain stream, while a query execution contains all relevant information of the continuous execution of a registered query: memory usage, query execution time, query processing time and the number of query results. Currently, we have implemented the RSP Engine Monitor for the C-SPARQL RSP engine. To retrieve the monitored information of a query execution, we have modified the source code of the C-SPARQL implementation to send the relevant information via callbacks to our wrapper implementation.

The general part of the implementation included in the DIVIDE Local Monitor JAR consists of a WebSocket client that actively keeps an open connection with the WebSocket server hosted by the RSP engine wrapper. It converts JSON messages received over the WebSocket to individual monitoring JSON observations, which comprise the output of the RSP Engine Monitor.

4.2 DIVIDE Global Monitor

The DIVIDE Global Monitor is implemented as an additional module of the original, central DIVIDE implementation. It is thus integrated into the executable Java JAR of DIVIDE Central.

4.2.1 Configuration of the DIVIDE Global Monitor

The DIVIDE Global Monitor should be configured in the main JSON file that is used for the configuration of DIVIDE Central. Specifically for the monitor, it defines whether the monitoring subcomponents should be active, a path to the built JAR file of the DIVIDE Local Monitor, and a list of files that contain the Global Monitor queries that should be evaluated.

4.2.2 Integration of the DIVIDE Meta Model

On implementation level, the DIVIDE Meta Model is part of the Global Monitor Reasoning Service: this reasoning service maintains a data model with ontology axioms extracted from the Meta Model ontology, and all meta-information about DIVIDE Core represented as contextual data triples.

The integration of the DIVIDE Meta Model into the DIVIDE Core subcomponent is implemented through a level of abstraction: the DIVIDE Meta Model exposes an interface to DIVIDE Core that is called for all relevant meta-information updates. These updates include the addition or removal of a DIVIDE query or DIVIDE component, an update to the RSP queries registered to a DIVIDE component, and the update of the query deployment of a DIVIDE query on a DIVIDE component. Upon every update, a collection of triples is constructed based on templates that are predefined based on the Meta Model ontology concepts. These triples are then added to or removed from the data model maintained by the Global Monitor Reasoning Service.

4.2.3 Implementation of the Global Monitor Reasoning Service

The Global Monitor Reasoning Service is implemented using the Apache Jena rule reasoner [23]. It continuously executes all registered Global Monitor queries in the order defined in the JSON configuration, on a tumbling data window of 20 seconds on the stream of aggregated observations received from the deployed Local Monitor instances.

4.2.4 Management of the DIVIDE Local Monitor instances

The implementation of the DIVIDE Global Monitor is also responsible for managing the configuration and state of the DIVIDE Local Monitor instances deployed on the DIVIDE components in the IoT network. To achieve this, the SSH and SCP protocols are used. To this end, our implementation assumes that every DIVIDE component in the network is reachable and allows incoming SSH connections using SSH public key authentication with a predefined username. Upon start-up of DIVIDE Central, the DIVIDE Local Monitor JAR and its configuration are copied over SCP to every

DIVIDE component, and the JAR is started over an SSH connection. This full process is implemented in Python.

4.3 DIVIDE Core

The DIVIDE Core component includes the DIVIDE engine, the DIVIDE reasoning module and the DIVIDE server. For details about the original implementation of these modules, we refer to our previous paper on DIVIDE [6]. To accommodate the DIVIDE Core modules for the extension of DIVIDE with the monitoring subcomponents, several updates have been implemented. First, DIVIDE Core is extended to alert the necessary updates of meta-information to the DIVIDE Meta Model, as explained before. Second, an implementation is provided for the distribution and configuration update tasks to the RSP queries deployed across the network.

The DIVIDE engine maintains a blocking task queue and a dedicated processing thread to execute these tasks, for every registered DIVIDE component. Existing task types are a task to derive the RSP queries for a DIVIDE query whenever a context change relevant to that component is detected, and a task to remove a DIVIDE query from the component. Two additional DIVIDE engine tasks have been implemented that correspond to the tasks forwarded by the DIVIDE Monitor Translator in the methodological design of DIVIDE: a query location update task, and a window parameter update task.

Query location update task

To implement the query location update task, the main configuration of DIVIDE Central was updated to include details about how to communicate with the API of the Central RSP Engine. Furthermore, we have implemented an RSP engine wrapper that exposes an API to manage the RSP engine and retrieve information from it. Concretely, this API allows retrieving details about registered streams, queries and their observers. Moreover, it also allows registering and unregistering a data stream, registering and unregistering an RSP query, and registering or unregistering an observer URL to an RSP query. Finally, it supports enabling and disabling the forwarding of all data on a registered stream over a WebSocket connection to a registered data stream of another RSP engine. Note that this wrapper also includes a part of the implementation of the RSP Engine Monitor, as discussed before.

To use our implementation of DIVIDE, we require that all Local RSP Engine instances and the Central RSP Engine are deployed with this wrapper, or at least offer an API with semantically and syntactically equivalent endpoints. To demonstrate and evaluate our system, we have currently integrated the C-SPARQL RSP engine into our wrapper implementation. Other RSP engines can however be easily integrated in the future.

Window parameter update task

The implementation of the window parameter update task makes use of the implementation of the DIVIDE query derivation. To implement this, the DIVIDE reasoning module keeps track in memory of intermediate query results for every combination of DIVIDE query and DIVIDE component. Such an intermediate query result contains

the output of the input variable substitution step of the DIVIDE query derivation (step 4), which is explained in Section 2.2. Whenever a window parameter update task is then executed for a given DIVIDE query and DIVIDE component, the query derivation is started in step 5 with this saved intermediate query result as input.

5 Related work

In this section, our examination of existing literature and research areas is presented. It shows for different areas how this paper is positioned within existing research. The section focuses on three relevant areas related to this work: semantic technologies to process heterogeneous data, ontologies in the domain of monitoring various situational context parameters, and the usage of monitoring techniques to dynamically distribute processing tasks across an (IoT) network.

5.1 Semantic technologies to process heterogeneous data

To process heterogeneous data in an IoT context, semantics can be leveraged [4]. Semantic Web technologies, endorsed by the World Wide Web Consortium (W3C), enable the semantic enrichment and integration of heterogeneous data sources [24]. To this end, different standards and protocols such as RDF, OWL, and SPARQL are part of this set of technologies [8–10].

In general, within semantics, a distinction can be made between bottom-up and top-down approaches [25]. Top-down semantics start with a broad understanding of a certain domain and then apply it to specific instances, emphasizing a hierarchical structure in the ontology design. In contrast, bottom-up semantics begin with specific instances or details and gradually build up an ontology to form a comprehensive understanding, prioritizing a more data-driven or empirical approach. The top-down approach is deductive, moving from general principles to specifics, whereas the bottom-up approach is inductive, deriving generalizations from specific observations [26].

Zooming in more specifically on solutions that integrate semantics in IoT platforms, many dedicated platforms and architectures exist. These different existing solutions deal with the heterogeneity challenge in the IoT in different ways [27]. For example, Antunes et al. present a semantic-based solution that allows tackling the heterogeneity of sensors and data by relying on an information retrieval system to resolve dynamic semantic queries [28]. This set-up uses the publisher/subscribe message pattern and is especially well-suited for machine to machine (M2M) scenarios. Furthermore, Corral-Plaza et al. propose an architecture that streamlines the processing and analysis of data from diverse sources within IoT scopes [29]. By combining real-time stream processing and complex event processing, the architecture enables seamless information processing, transformation, and analysis of large volumes of data in real-time, allowing researchers to focus on data analysis without concerns about data source structures. Moreover, Kalamares et al. introduce an architecture for big data analytics in large-scale systems with multiple IoT platforms, containing a semantic interoperability layer featuring semantic mappings and a unified ontology [3]. It also includes a cloud-based data management layer that enables web-based data analytics and visual analytics methods that remain platform-agnostic.

In addition, dedicated semantic IoT platforms have been designed as well. VICINITY [30], INTER-IoT [31], and BIG IoT [32] emphasize flexible IoT APIs and gateway architectures. Taking a step beyond, SymbIoTe creates a virtual IoT environment across diverse cloud-based IoT platforms, abstracting existing IoT platforms in the process [33]. Sofia2, on the other hand, serves as a semantic middleware platform facilitating interoperability among various systems and devices for smart IoT applications [34]. Addressing the need for open platforms in IoT systems development, bIoTope comes into play [35]. FIWARE, offering a range of APIs, can become a platform of choice for deploying IoT applications [36].

Zooming in on a specific IoT domain such as healthcare, multiple semantic IoT platforms specifically focusing on interoperability across heterogeneous devices within this domain already exist as well [37–40]. This is the case for multiple other IoT domains as well, such as smart cities [41–44].

In summary, a wide range of semantic solutions exist, designed both generically and for specific IoT application domains, ranging from full-fledged semantic IoT platforms to novel architectures. They all use semantics in some way to achieve interoperability across heterogeneous IoT devices and data. These different solutions all offer the opportunity to investigate how the stream processing architecture presented in this paper can be fused together or be integrated to combine their different strengths and improve the overall system quality.

5.2 Monitoring ontologies

Multiple ontologies exist in the domains of network monitoring [45]. MonONTO is a domain ontology that bridges the domains of network performance monitoring and application quality of service [46]. It specifically focuses on incorporating domain knowledge through inference rules. In addition, a dedicated ontology for traffic monitoring in IP networks was designed within the European project MOMENT [47]. Similarly, the EU-funded NOVI project resulted in multiple ontologies to describe and monitor network resources [48]. Moreover, Silva et al. have presented an ontology that allows defining a network measurement topology and sampling techniques to enable context-aware network monitoring [49]. Multiple other approaches focus specifically on designing an ontology for telecommunications network management and monitoring [50–52].

In the domain of device monitoring, Funika et al. present an ontology-based approach to perform the monitoring of resource usage in multi-scale platforms [53]. Connecting the domains of device monitoring and IoT, Ryabinin et al. demonstrate an ontology-based approach to manage and monitor resource-constrained Edge Computing devices [54]. The Comprehensive Ontology for IoT (COIoT) tries to build an interoperable knowledge base for IoT environments by reusing core concepts from existing ontologies and adds additional concepts to support the monitoring of context and services [55].

To the best of our knowledge, most of the described ontologies are not available through the cited publications or in well-known ontology repositories[1]. Therefore, we have used the concepts and ontology structures described and presented as figures in the cited publications as inspiration to create the DIVIDE Meta Model ontology presented in Section 3.2. In this process, we have focused on the ontology structures that were needed in the methodological design of the monitoring subcomponents of DIVIDE, to achieve the research objectives of this work presented in Section 1.

Some additional models and ontologies were used as inspiration or direct imports in the designed Meta Model ontology of DIVIDE. The DEN-ng model is a semantic model for the management of computer networks [56], translated to an ontology file by Jeroen Famaey et al. [57, 58]. Moreover, the SAREF ontology is an ETSI standard that focuses on the smart applications domain [19]. In addition, the Ontology of units of Measure defines all possible quantities and units of measures [20]. Finally, the Computer Hardware Components Ontology is a small ontology that defines useful concepts such as monitoring devices, resources, network topologies, network addresses, etc. [59].

5.3 Monitoring to dynamically distribute tasks across networks

Existing research already focuses on performing monitoring of situational context such as network and device conditions, to dynamically distribute knowledge and processing tasks across the network. This involves both ontology-based and other approaches.

Zooming in on ontology-based solutions, Keeney et al. have presented an approach that tries to automatically decentralize the number of required semantic reasoning tasks on semantically enriched data within a network [60]. This approach mainly uses streaming network and service monitoring data that is modeled through the End-User Service Analysis and Optimization (EUSAO) ontology [61]. It uses RDF stream reasoning to perform real-time event correlation on these data streams. It explores the potential of parallelizing and/or distributing these reasoning tasks based on required expressivity and by combining it with the partitioning of rules and data. Similarly, Keeney et al. have presented an approach that efficiently tries to distribute heterogeneous knowledge to network nodes that express interest in certain knowledge sources, by making use of ontology-based semantics [62]. These different approaches proof the applicability and usefulness of designing and deploying an ontology-based monitoring system within a broader semantic system.

Multiple, non-ontology based solutions exist as well in the domain of dynamic monitoring-based network task distribution. AIOLOS is a mobile middleware framework that considers the resources of the server and the conditions of the network to determine at runtime whether some tasks of a mobile application need to be offloaded to a nearby server in the network [63]. Moreover, Sebrechts et al. have presented a fog native architecture that intelligently decides how microservice-based applications can be distributed over a network [64]. This approach takes into account device and network conditions, as well as meta-information about the available infrastructure, end

[1]The following online repositories were consulted: https://lov.linkeddata.es/dataset/lov/, https://bioportal.bioontology.org/ontologies, https://www.ebi.ac.uk/ols/ontologies, and http://www.sensormeasurement.appspot.com/?p=ontologies.

user requirements, application tasks and more, in order to improve overall performance according to those application conditions. This way, it combines the advantages of edge computing and cloud native microservice applications. Similarly, Santos et al. have designed a scheduling approach for container-based applications within IoT application domains, that incorporates real-time information of the status of the network infrastructure [65]. As a proof-of-concept, they have investigated how such a network-aware scheduling approach can be implemented as a scheduling extension within Kubernetes. Furthermore, Idrees et al. have designed a protocol for fog computing architectures that intelligently assigns tasks to edge devices to minimize the network communication costs and the energy usage of edge devices [66]. It specifically zooms in on the energy-efficient transmission and aggregation of data, which is important due to the limited energy of sensor devices in an IoT context.

As such, it is clear that different approaches already focus on the dynamic distribution of processing tasks in IoT networks. Most of them mainly integrate the dynamic nature based on networking characteristics, while some approaches also look further to aspects like energy usage. The existence of actual distribution frameworks, algorithms, and protocols highlights the need to further examine dynamic and generic monitoring frameworks within an IoT context, which is a challenge addressed in this paper. Moreover, this also highlights the opportunity to integrate such solutions into these existing assets, which could be a topic for future research.

Looking at this topic from a broader point of view, a wide variety of auto-scaling techniques for IoT-based cloud applications exists within literature [67]. As defined by Verma et al., such auto-scaling techniques mainly consist of a monitoring and a scaling component, which scales resources according to the requirements in relation to the monitored information. Both schedule-based and rule-based methods exist. Rule-based methods can especially be of relevance in ontology-based systems like the one presented in this paper, since these rules can be complementary to the semantic queries presented in this work that update the task distribution. Related to that, reinforcement learning based techniques are becoming increasingly popular as well [68]. In general, this proves that it is worth investigating how the monitoring, auto-scaling, and dynamic task distribution can be fused together.

6 Evaluation set-up

To validate and demonstrate our implementation of the methodological design of DIVIDE, it is evaluated on a homecare monitoring use case. This section zooms in on this use case, the compared technical set-ups, and the different scenarios of the evaluation.

6.1 Evaluation use case

The evaluation is performed on a homecare monitoring use case in healthcare, which is a well-known IoT application domain [40]. This section zooms in on this use case by describing it, and discussing the ontology, use case context and DIVIDE query that are considered for the evaluation. Furthermore, the section explains which dataset is used for the simulation of realistic homecare IoT data in the evaluation scenarios.

6.1.1 Use case description

This section discusses three relevant aspects of the homecare monitoring use case: its storyline, the technical set-up, and the specific homecare monitoring task that is considered in the evaluation scenarios.

Storyline

Consider a homecare monitoring IoT environment with an alarm center that is responsible for the monitoring of different service flats spread out across the city. Every service flat is equipped with a wide range of monitoring sensors and devices. Moreover, every patient is wearing a wearable to monitor the patient's in-house location, acceleration and physiological parameters such as heart rate. Furthermore, a nurse call system is installed in every service flat. This nurse call system allows patients to generate an alarm to the alarm center whenever they are in need in of assistance. A patient can do this by pushing a button on a dedicated wearable device. This alarm is then received by a team of human call operators in the alarm center, who should decide which intervention strategy is required. Possible intervention strategies are calling an ambulance, sending a doctor or a nurse with a certain priority, or calling an informal caregiver to pay a visit to the patient.

To help the human operators with choosing the most optimal intervention strategy, the homecare monitoring installation can be used. In this system, lots of individual measurements are generated by the installed lifestyle monitoring devices, environmental sensors, wearable sensors, and possibly others. By reasoning on these measured parameters in combination with existing medical domain knowledge and use case specific context information such as the patient's disease profile, detailed insights can be generated. Examples of relevant insights that help the human call operators are the activity level of the patient, performed in-home activities, medical conditions, and many others.

Whenever a call is generated by a patient, detailed dashboards should be available to the call operators that can be analyzed to correctly assess the situation. These dashboards are also relevant to the other healthcare professionals, such as the nurses that might be called to visit the patient. Importantly, the dashboards should not only show the insights generated by the different algorithms, but they should also include timeline visualizations of any relevant raw sensor data. To achieve this, as much of the raw sensor data as possible should be available on the central servers of the alarm center. This server-side availability of raw data is especially important for patient security as well, for two main reasons. First, the raw data can be used to motivate why certain decisions were made by the call operators. Second, whenever an intervention is chosen by a call operator that later turns out to be the wrong choice, the raw data also allows analyzing in detail why the wrong intervention was chosen.

Importantly, the use case requirement of patient security should be well-balanced with cost. Sending over all raw sensor data from all service flats to the central server to run all the processing tasks centrally, would require a high-end server infrastructure and thus incur high costs. If the budget does not allow these costs, overusing the existing server-side resources would imply a risk of the server going down and being unavailable at times. For obvious reasons, this is unacceptable in the considered

healthcare context. Hence, to reduce costs and ensure the system is not at risk of going down, less delicate tasks for which the central need of raw sensor data is less high, should be executed locally instead.

Technical set-up

The set-up of the homecare monitoring system uses the cascading reasoning architecture presented in Section 3.1. This architecture involves the deployment of the different local and central subcomponents of DIVIDE. The different homecare monitoring tasks are deployed as RSP queries across the network with DIVIDE by defining them as DIVIDE queries. The central components are running in a server environment in the facilities of the alarm center. The edge components are deployed locally in the service flats of the patients, on the available devices on which the running nurse call system is deployed. In other words, there is one DIVIDE component per patient (i.e., per service flat), with a single instance of the Local RSP Engine and the DIVIDE Local Monitor running on the local nurse call system device.

Concerning the deployed nurse call system, different types and versions of the software exist. Some include more services than others, implying that a different amount of resources is required to run the nurse call system across the different service flats. Hence, different local devices are used to deploy the nurse call system, with a different amount of resources. Since the DIVIDE Local Monitor and Local RSP Engine are also deployed on these devices, the system should be able to adapt to this resource variability in a flexible way.

As the service flats are spread out over the city, public networks are being used for the communication between the service flats and the server infrastructure of the alarm center. This implies that the communication is prone to varying networking conditions over time and across the different service flats. Hence, DIVIDE should take these networking conditions into account when balancing the aforementioned trade-off between patient security and cost. Especially when the network is too slow to allow efficient forwarding of raw sensor data to the central servers, more homecare monitoring tasks might need to be deployed locally. This is obvious: having up-to-date aggregated insights about the patient without being able to inspect the raw data, is still a better situation for the alarm center compared to receiving no up-to-date insights at all.

Evaluation homecare monitoring task

This evaluation focuses on one specific part of the in-home monitoring of patients: monitoring the patient's level of activity. This is an important monitoring task for a variety of medical diseases and medical conditions. This includes fall-prone patients, heart patients and patients with dementia. Depending on the condition, the level of granularity that is required for insights into the activity level of the patient differs. The required granularity level influences the priority of the need for the server-side availability of the raw sensor data. For fall-prone patients, fine-grained insights into the activity level at every point in time are required, to precisely detect whenever this patient would fall. For heart patients, some level of granularity is also desirable, as heart conditions might vary over time. For patients with dementia, less granularity is

needed, as the healthcare professionals are mostly interested in knowing whether the patient is still moving over time or not.

In this evaluation, the activity level of the patients is measured by calculating the activity index value of the patient's acceleration, which is continuously measured by the patient's wearable. This index is defined as the mean variance of the acceleration over the three axes [69]. The higher this value, the more active the person has been in the considered time window.

6.1.2 Ontology, use case context and DIVIDE query

The ontology used for the evaluation is an additional module built upon the Data Analytics for Health and Connected Care (DAHCC) ontology [70]. This DAHCC ontology is a publicly available, in-house designed ontology with different modules that allow connecting data analytics to healthcare knowledge in an IoT environment. It connects several existing ontologies in these domains such as SAREF [19], the SAREF extension for the eHealth Ageing Well domain (SAREF4EHAW) [71], and the Execution-Executor-Procedure (EEP) ontology [72]. Through different models, the DAHCC ontology allows capturing metadata about IoT sensors and observations, different AI algorithms, insights about patient health derived from those algorithms, and how these insights are related to the medical condition of the patients.

The additional module built upon the DAHCC ontology contains a description of a health parameter calculator system. Such a system can be configured with different rules about how to measure certain health parameters. The relevance of these parameters can then be linked to the medical condition of the patients. Specifically for the homecare monitoring task of this evaluation, the ontology defines that the activity index is a relevant health parameter that needs to be monitored for patients that require movement monitoring. This movement monitoring requirement is defined for several medical conditions, such as being fall-prone, being a heart patient, or having dementia.

The use case context for the evaluation scenarios contains a patient diagnosed with one of the aforementioned three medical conditions. This patient is living in a smart home that contains a wide range of IoT sensors, and has a wearable that at least measures 3-axis acceleration. Throughout the course of the evaluation scenarios presented in this paper, the use case context is considered static: it does not change, as the focus is on the varying situational context such as networking conditions and device resource usage.

The DIVIDE query that is registered to the DIVIDE engine in the evaluation scenarios can be instantiated to an RSP query that calculates the patient's activity index. Given the combination of ontology, use case context and DIVIDE query, this instantiation will happen for the patient described in the use case context. The resulting RSP query will thus be deployed for the DIVIDE component corresponding to this patient's service flat.

Appendix C provides the semantic details of the evaluation use case. It presents relevant definitions from the designed ontology module, an overview of important triples in the use case context, and details of the internal representation of the discussed DIVIDE query.

6.1.3 Realistic dataset for simulation

The evaluations in this paper are performed using simulations of IoT sensor data in a smart home environment. To ensure that the evaluations are representative, a real-world dataset is employed for these simulations. This dataset is the result of a large scale data collection process in the imec-UGent HomeLab. The HomeLab is a standalone house that can be used as a unique residential testing environment for homecare monitoring use cases. It is equipped with different sensors that measure localization, environmental conditions, user actions and much more. During the data collection process, patients were equipped with an Empatica E4 wearable [73]. This device has a 3-axis accelerometer with a frequency of 32 Hz, as well as multiple other physiological sensors.

For the evaluation scenarios of this work, an anonymous representative part is extracted from the data collected from a random patient. This simulation dataset is left unchanged, except for shifting the observation timestamps to real-time timestamps. In total, it contains an average of 186 observations per second. In fact, for the presented homecare monitoring task, the availability of wearable acceleration data is the only requirement. Nevertheless, by using the realistic dataset for simulation, the volume and variety of the simulated raw sensor data is representative for a real-world service flat.

6.2 Compared set-ups

The evaluation scenarios are evaluated on different technical set-ups.

1. **DIVIDE Monitoring set-up:** This is the baseline set-up that deploys all subcomponents of DIVIDE, according to the architecture presented in Section 3.1. The set-up uses our implementation presented in Section 4.
2. **DIVIDE Local set-up:** This set-up considers the architecture presented in Section 3.1, but without the monitoring subcomponents of DIVIDE. This means that the set-up includes the Semantic Mapper and Local RSP Engine on the local device, and the Central Processing Component, Knowledge Base and DIVIDE Core components on the central device. For DIVIDE Core, our implementation of this component as discussed in Section 4 is used. However, its task to keep the DIVIDE Meta Model up-to-date is deactivated, as this set-up does not include the Meta Model. Since no subcomponents of the DIVIDE Monitor are deployed, the RSP queries derived by DIVIDE Core always remain active on the Local RSP Engine after DIVIDE Core has registered them to it.
3. **DIVIDE Central set-up:** This set-up is identical to the DIVIDE Local set-up, except for one change: all RSP queries derived by DIVIDE Core are always registered to the Central RSP Engine. Hence, the Local RSP Engine only forwards the monitoring observations and does not evaluate any RSP queries. This is implemented by programmatically issuing a query location update task to the Central RSP Engine for all derived queries, after they are initially registered to the Local RSP Engine.

All implementations of the RSP engine components use the C-SPARQL RSP engine, with the RSP engine wrapper discussed in Section 4. The implementation of the

Central Reasoner is mocked, since it has no actual task in the evaluation scenarios, apart from exposing an API endpoint for its data stream to which the outputs of the RSP queries can be forwarded.

It is important to note that all evaluation set-ups include the DIVIDE Core subcomponent. This decision is made to ensure that the evaluation investigates the benefits gained by deploying the monitoring subcomponents of DIVIDE. Analyzing the advantages of using DIVIDE Core over other set-ups that do not include DIVIDE, has already been done in our previous work and is therefore considered out of scope for this work [6].

6.3 Evaluation scenarios

Two evaluation scenarios are designed for the described homecare monitoring use case. They are discussed in the following subsections.

6.3.1 Evaluation scenario 1: updating the RSP query window parameters based on RSP monitoring

The first evaluation scenario focuses on modifying the window parameters of the deployed RSP query that monitors the patient's activity index. The goal of the scenario is to demonstrate how DIVIDE allows dynamically adapting the query window parameters to external factors that prevent the healthcare monitoring from running smoothly.

The focus in this evaluation scenario is on the components that run on the local devices in the network. Hence, for simplicity, only a single DIVIDE component is registered to DIVIDE Central.

Scenario timeline

The evaluation scenario takes 5 minutes. In the beginning of the scenario, the nurse call system has no active processes on the local device. After on average 60 seconds into the scenario, a resource-intensive process is started by the nurse call system. 30 seconds later, it starts three more processes. Together, the nurse call processes consume at most 450 MB of RAM, and are very CPU-intensive. These processes are simulated with the Unix workload generator tool *stress*. All started processes continue running for the remainder of the scenario.

This scenario assumes that the networking conditions are too bad to forward the raw accelerometer data to the central servers of the alarm center. This remains the case during the full scenario timeline. Hence, the RSP query is always deployed on the Local RSP Engine and cannot be moved centrally.

Data simulation

To simulate the IoT data for the evaluation scenario, a 5-minute chunk of realistic IoT sensor data is extracted from the real-world dataset described in Section 6.1.3. During every evaluation run, this chunk is replayed in real-time by a data simulation component. This component is running on an external device that is connected to the considered local device via a local network. This way, the simulation component

realistically represents the different IoT sensor gateways. During an evaluation run, the simulation component opens a client connection to the WebSocket server exposed by the wrapper of the Local RSP Engine. Every second, it creates a single-message batch containing all sensor observations of that second, triggers the Semantic Mapper to semantically annotate the messages in the batch, and sends the batch over the WebSocket to the semantic data stream that is registered to the Local RSP Engine. The RDF triple language used in the simulation is N-Triples.

Set-ups

The presented scenario is evaluated on the DIVIDE Monitoring set-up and the DIVIDE Local set-up presented in Section 6.2. The DIVIDE Central set-up is not considered as the activity index RSP query will always be registered on the Local RSP Engine, also in the DIVIDE Monitoring set-up. For every set-up, the DIVIDE query to calculate the patient's activity index is registered to DIVIDE Core. The static window parameters of the DIVIDE query define a window size of 80 seconds and a query sliding step of 10 seconds.

Global Monitor query

In the DIVIDE Monitoring set-up, one Global Monitor query is defined by the end user to be evaluated on the Global Monitor Reasoning Service. This query is presented in Listing 6. It monitors the execution time of all RSP queries deployed on the Local RSP Engine of a DIVIDE component. It checks whether the maximum processing time of an RSP query exceeds its sliding step, which would mean that the previous query execution is still ongoing when the next execution needs to start. If this happens, the Global Monitor query defines a window parameter update task for the DIVIDE query corresponding to this RSP query. Both window parameters are updated: the query sliding step is doubled, and the window size is halved. Note that the usage of `GROUP BY` and the aggregations in the Global Monitor query ensure that only one task is outputted for every combination of DIVIDE query and DIVIDE component.

Measurements

For every run of the evaluation scenario, the measurements performed by the individual monitors of the Local Monitor are saved. This includes the CPU and RAM usage of the local device, and the processing time on the Local RSP Engine of the deployed RSP query that measures the patient's activity index. For the DIVIDE Local set-up, the Python script of the Device Monitor is manually run so that the CPU and RAM usage is also measured. Moreover, the number of observations in the data window upon every RSP query execution is collected as well.

Technical specifications

This evaluation scenario is executed on physical IoT devices. The central device hosting DIVIDE Central and the Central Processing Component is an Intel NUC, model D54250WYKH. It has a 1300 MHz dual-core Intel Core i5-4250U CPU (turbo frequency 2600 MHz) and 8 GB DDR3-1600 RAM. The local device with the Local RSP Engine and the DIVIDE Local Monitor is a Raspberry Pi 3, Model B. This Raspberry

Listing 6: Global Monitor query deployed on the DIVIDE Monitoring set-up, in evaluation scenario 1 that updates the RSP query window parameters based on the monitored RSP query processing time

```
CONSTRUCT {
    [ a divide-core:DivideWindowParameterUpdateTask ;
      divide-core:isTaskForDivideQueryName ?divideQueryName ;
      divide-core:isTaskForComponentId ?componentId ;
      divide-core:hasUpdatedQuerySlidingStepInSeconds
          ?maxUpdatedSlidingStep ;
      divide-core:hasUpdatedWindowSizeInSeconds ?minUpdatedWindowSize ]
}
WHERE {
    { SELECT ?componentId ?divideQueryName
             (MAX(?updatedSlidingStep) as ?maxUpdatedSlidingStep)
             (MIN(?updatedWindowSize) AS ?minUpdatedWindowSize)
      WHERE {
          ?component a divide-core:DivideComponent ;
                     divide-core:hasID ?componentId ;
                     divide-core:hasLocalRspEngine ?rspEngine .
          ?rspEngine divide-core:hasRegisteredQuery ?rspQuery .
          ?rspQuery divide-core:hasCorrespondingDivideQuery ?divideQuery ;
                    divide-core:hasAssociatedComponent ?component ;
                    divide-core:hasQuerySlidingStepInSeconds ?querySlidingStep ;
                    divide-core:hasWindowSizeInSeconds ?windowSize .
          ?divideQuery divide-core:hasName ?divideQueryName .

          ?measurement a saref-core:Measurement ;
                       saref-core:hasValue ?maxProcessingTime ;
                       om:hasAggregateFunction om:maximum ;
                       saref-core:isMeasuredIn om:second-Time ;
                       saref-core:relatesToProperty [
                           a monitoring:RspQueryProcessingTime ] ;
                       saref-core:isMeasurementOf ?rspQuery .

          FILTER (xsd:float(?querySlidingStep) <
                  xsd:float(?maxProcessingTime))

          BIND(xsd:integer(?querySlidingStep) * xsd:float(2)
               AS ?updatedSlidingStep)
          BIND(xsd:integer(FLOOR(xsd:integer(?windowSize) / xsd:float(2)))
               AS ?updatedWindowSize)
      }
      GROUP BY ?componentId ?divideQueryName }
}
```

Pi model has a Quad Core 1.2 GHz Broadcom BCM2837 64bit CPU, 1 GB RAM and MicroSD storage.

6.3.2 Evaluation scenario 2: updating the RSP query location based on network monitoring

The goal of the second evaluation scenario is to evaluate how DIVIDE is able to optimally distribute the RSP queries across the IoT network based on real-time networking conditions. To this end, the scenario focuses on updating the location of the RSP query that monitors the patient's activity index.

This evaluation scenario focuses on both the local and central devices in the network. To this end, it considers an IoT network with a single central device and three local devices. Every local device contains one DIVIDE component that is registered

to DIVIDE Central, and represents a single service flat. In the scenario timeline and evaluation measurements, only the interaction between the central device and a single DIVIDE component is considered.

Scenario timeline

The duration of this evaluation scenario is 12 minutes. Throughout the full scenario, the properties of the network interface that connects the considered local device with the central device are constantly varied, to simulate varying networking capacity. This simulation adaptively alternates periods with normal (baseline) and worse networking capacity conditions. More specifically, worse networking conditions apply at peak level during the following time periods of the scenario (approximately): minute 1 to 3, minute 5 to 7, and minute 9 to 11. Compared to the baseline capacity, the second period imposes the highest amount of capacity restrictions, while the third period imposes the smallest amount of capacity restrictions.

For simplicity purposes, the networking conditions for the other two DIVIDE components are not varied. Hence, baseline capacity conditions apply. As a consequence, throughout the full scenario, the RSP queries monitoring the activity index for the patients associated to these other two DIVIDE components, are evaluated on the Central RSP Engine.

Data simulation

During the runs of this evaluation scenario, the IoT data is simulated in an identical way to the scenario discussed in Section 6.3.1. This scenario however uses a 12-minute chunk of IoT from the simulation dataset. The simulation is performed for all three DIVIDE components.

Set-ups

This scenario is evaluated on all three set-ups presented in Section 6.2: DIVIDE Monitoring, DIVIDE Local and DIVIDE Central. At the start of the simulation for the DIVIDE Monitoring set-up, the RSP query that monitors the patient's activity index will be deployed on the Central RSP Engine. For every set-up, the DIVIDE query to calculate the patient's activity index is registered to DIVIDE Core. The static window parameters of the DIVIDE query define a window size of 60 seconds and a query sliding step of 10 seconds.

Global Monitor query

In the DIVIDE Monitoring set-up, the configuration of the DIVIDE Global Monitor contains two Global Monitor queries defined by the end user. The first Global Monitor query is shown in Listing 7. It monitors the average network RTT for the connection between every local device and the central device in the IoT network. If this RTT exceeds the threshold of 2 seconds on a device for which queries of the corresponding DIVIDE component are running on the Central RSP Engine, a query location update task is issued that moves this query to the Local RSP Engine. The second Global Monitor query monitors the reverse situation: it ensures that RSP queries deployed

Listing 7: Global Monitor query deployed on the DIVIDE Monitoring set-up, in evaluation scenario 2 that updates the RSP query location based on the monitored network RTT

```
CONSTRUCT {
    [ a divide-core:DivideQueryLocationUpdateTask ;
      divide-core:isTaskForDivideQueryName ?divideQueryName ;
      divide-core:isTaskForComponentId ?componentId ;
      divide-core:hasUpdatedQueryLocation [ a divide-core:LocalLocation ] ]
}
WHERE {
    { SELECT DISTINCT ?componentId ?divideQueryName
      WHERE {
          ?device a saref-core:Device ;
                  divide-core:hosts ?component .
          ?component a divide-core:DivideComponent ;
                     divide-core:hasID ?componentId  ;
                     divide-core:hasCentralRspEngine ?rspEngine .
          ?rspEngine divide-core:hasRegisteredQuery ?rspQuery .
          ?rspQuery divide-core:hasCorrespondingDivideQuery ?divideQuery ;
                    divide-core:hasAssociatedComponent ?component .
          ?divideQuery divide-core:hasName ?divideQueryName ;
                       divide-core:hasQueryDeployment [
                           saref-core:isAbout ?component ;
                           divide-core:hasQueryLocation
                               [ a divide-core:CentralLocation ] ] .

          ?measurement a saref-core:Measurement ;
                       saref-core:hasValue ?avgRtt ;
                       om:hasAggregateFunction om:average ;
                       saref-core:isMeasuredIn om:second-Time ;
                       saref-core:relatesToProperty [
                           a monitoring:RoundTripTime ] ;
                       saref-core:isMeasurementOf ?device .

          FILTER (xsd:float(?avgRtt) >= xsd:float(2.0))
      } }
}
```

on the Local RSP Engine are moved back to the Central RSP Engine whenever the average RTT returns back to 1.6 seconds or lower.

Measurements

During the runs of the evaluation scenario, three time durations are measured for every evaluation of the RSP query that measures the patient's activity index: the local processing time, the networking overhead, and the central processing time. The definition of those metrics depends on whether the RSP query is running on the Local RSP Engine or on the Central RSP Engine.

- Query running on the Local RSP Engine:
  - Local processing time: time between the data window trigger of the query evaluation on the Local RSP Engine, *and* the Local RSP Engine sending the generated query result to the Central Reasoner
  - Networking overhead: time between sending the query result by the Local RSP Engine, *and* receiving this result on the Central Reasoner
  - Central processing time: value is 0, since the query is running locally
- Query running on the Central RSP Engine:

- Local processing time: value is 0, since the query is running centrally
- Networking overhead: time between receiving a 1-second sensor data batch on the Local RSP Engine, *and* receiving this batch on the Central RSP Engine (forwarded by the Local RSP Engine), averaged over all batches that are included in the triggered data window of the query evaluation
- Central processing time: time between the data window trigger of the query evaluation on the Central RSP Engine, *and* the Central Reasoner receiving the generated query result

Furthermore, the network RTTs measured by the Network Monitor of the Local Monitor are saved during the evaluation runs. To also measure these RTTs for the DIVIDE Local and DIVIDE Central set-ups, the Python script of the Network Monitor is manually run for those set-ups. Finally, the evaluation measures the number of triples that are sent over the network by the Local RSP Engine in every outgoing network event.

Technical specifications

To properly simulate a networking context, this evaluation scenario is run on virtual devices using the in-house iLab.t Virtual Wall environment [74]. All devices are virtual nodes with two 2.40 GHz hexacore Intel Xeon E5645 CPUs and 24 GB DDR3 1333 MHz RAM. For the three local devices, the RAM available to the processes of the device components is limited to 1 GB using Linux Control Groups (Cgroups). All devices are connected via a local area network of which the link characteristics are adaptively configured based on the scenario timeline.

6.3.3 Performance evaluation of the DIVIDE Local Monitor and DIVIDE Central

In the two presented evaluation scenarios, the general performance of the monitoring subcomponents of DIVIDE is also measured, in addition to the specific measurements relevant to each evaluation scenario.

Performance evaluation of the DIVIDE Local Monitor

For the DIVIDE Local Monitor, the measured performance metrics are the CPU & RAM usage, the number of triples in the output of the Local Monitor aggregation query that is sent over the network, and the processing times of this aggregation query. Similarly to the processing time measured by the RSP Engine Monitor, this metric is defined as the time between the query's data window trigger and the generation of the query results. These performance metrics are all measured during the runs of the first evaluation scenario presented in Section 6.3.1.

Performance evaluation of DIVIDE Central

Related to the DIVIDE Global Monitor of DIVIDE Central, two performance metrics are calculated.

First, the event processing times of the Global Monitor Reasoning Service are measured. This event processing time is defined as the sum of three parts:

(i) the time from the data window trigger of the Global Monitor query executions, until the data window with all aggregated observations from all Local Monitor instances is added to the DIVIDE Meta Model *and* OWL 2 RL reasoning on the DIVIDE Meta Model is performed;
(ii) the actual execution times of all Global Monitor queries, which are evaluated on the DIVIDE Meta Model;
(iii) and the time from ending the execution of the final Global Monitor query, until the data window with all aggregated observations is removed from the DIVIDE Meta Model *and* OWL 2 RL reasoning is again performed.

Second, the duration is measured of the two tasks that can be issued by the DIVIDE Global Monitor: a query location update task and a window parameter update task. The start of this duration is defined as the data window trigger of the Global Monitor query execution that leads to this task. The end is defined as the time at which the registration of the corresponding RSP engines and their observers at the respective RSP engines is completed by DIVIDE Core.

All performance metrics are measured during the runs of the first evaluation scenario presented in Section 6.3.1. The only exception is the duration of the query location update tasks, which is measured during the runs of the second evaluation scenario described in Section 6.3.2.

7 Evaluation Results

This section presents the results of the evaluations described in Section 6. All results contain data of multiple evaluation runs. More details about how the results are aggregated for the evaluations on the DIVIDE Monitoring set-up are presented in Appendix D.

7.1 Evaluation scenario 1: updating the RSP query window parameters based on RSP monitoring

Figure 4 shows the results of the first evaluation scenario, which is discussed in Section 6.3.1. The results show the evolution over time of the processing time of the RSP query that monitors the patient's activity index, together with the real-time RAM and CPU usage of the local device. In addition, the value of the query window parameters is shown. Moreover, Figure 5 shows the number of observations in the data window for every evaluation of the considered RSP query. The measurements on the graphs of both figures are averaged over the evaluation runs in time and value.

For the DIVIDE Monitoring set-up, Figure 4 shows that the query window parameters are updated on average 25 seconds after the first RSP query processing time exceeds the query sliding step of 10 seconds: the query sliding step is doubled and the query window size is halved. After the adaptations, the query is correctly executed every 20 seconds, and the average query processing time remains well below this current sliding step of 20 seconds. Hence, no further adaptations to the window parameters are issued. Moreover, Figure 5 proves that the queries are executed on the expected number of observations, also after the window parameters are updated: this value should be around 7,440 (on average 186 observations per second on a window of 40 seconds).

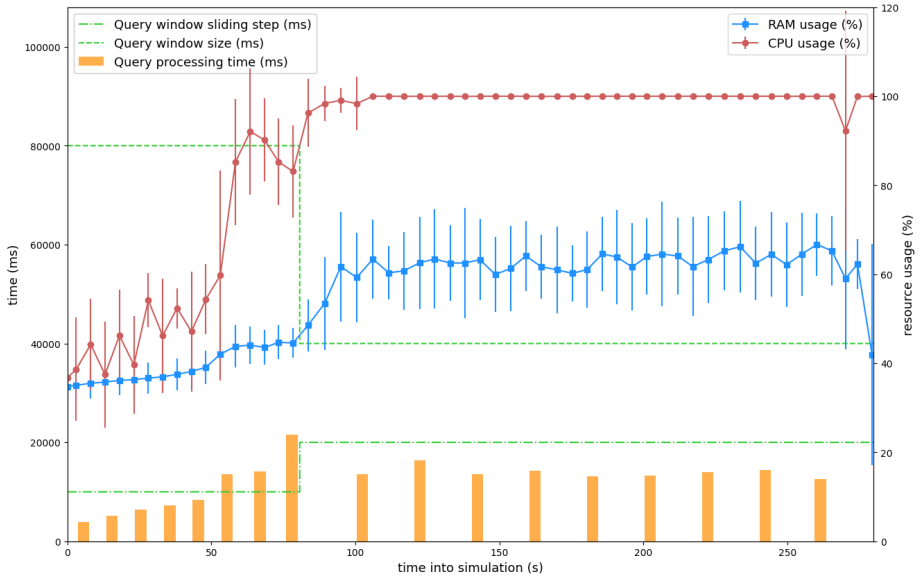
Throughout the scenario, the RAM and CPU usage increase. After on average 58 seconds, the CPU usage shows its largest increase to more than 85% on average. This increase is approximately at the same time at which the first nurse system process is started on the device. After more than 100 seconds into the scenario, the CPU usage reaches and almost constantly remains 100%.

For the DIVIDE Local set-up, a similar trend in the resource usage can be observed on Figure 4. However, as the monitoring subcomponents of DIVIDE are not deployed, no adaptations to the window parameters are made. As a consequence, after more than 100 seconds into the evaluation, the RSP query is no longer correctly executed every 10 seconds. Often, the period between successive query executions is larger (at most on average more than 22 seconds) or smaller (on average less than 7 seconds at the smallest), causing an irregular pattern. Moreover, the query processing times are irregular as well: they are larger than 30 seconds on average on seven executions, with an average maximum value of more than 39 seconds. Figure 5 demonstrates that the number of observations in the data window on which the RSP query is executed, is also way lower than expected. The value should be around 14,880 (on average 186 observations per second on a window of 80 seconds), but is often way lower. This means that the set-up cannot keep up with the data velocity, and is thus ignoring many sensor observations in most query evaluations.

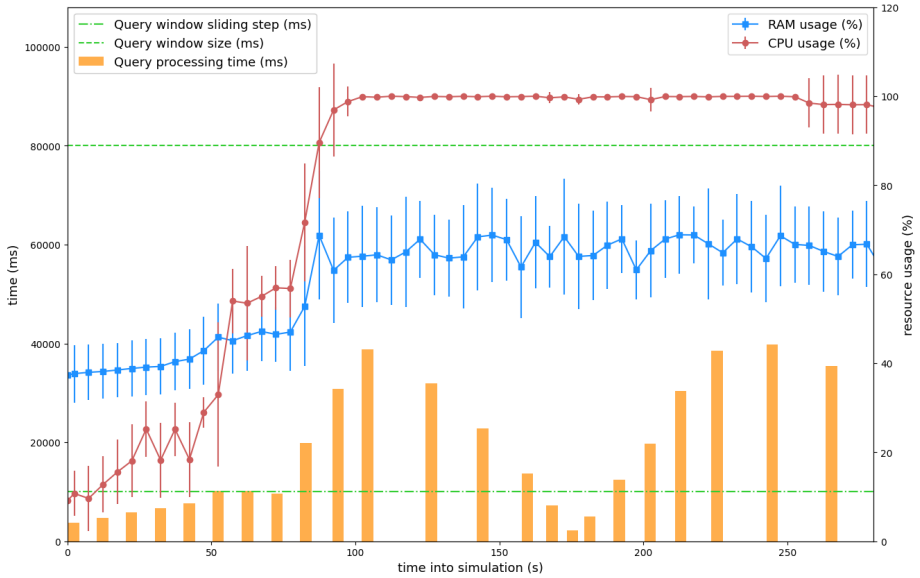
7.2 Evaluation scenario 2: updating the RSP query location based on network monitoring

Figure 6 shows the results of the second evaluation scenario, which is discussed in Section 6.3.2. The graph visualizes the evolution over time of the local processing time, network overhead and central processing time related to the evaluation of the RSP query that monitors the patient's activity index. Moreover, the network RTT between the local and central device, and the number of triples sent over the network in every outgoing event is plotted as well. All measurements are averaged over the evaluation runs in time and value.

For the DIVIDE Monitoring set-up, the RSP query is moved between the Local RSP Engine and Central RSP Engine four times. It is moved from the central to the local device during two of the three simulated periods of bad networking conditions, and moved back to the central device after the network conditions have improved again. The delay between the start or end of the period of bad networking conditions and the actual query move varies between approximately 50 and 80 seconds on average. This delay is caused by the Global Monitor queries, which require the average RTT to be above or below a certain threshold. This average is calculated by the Local Monitor RSP Engine on a data window of 60 seconds. Therefore, queries are only moved if bad or good networking conditions persist for a certain period. The network overhead of the RSP query processing varies with a similar pattern as the measured network RTT. Concretely, it varies between 109 ms and 10,412 ms on average, with an average value of 2,339 ms (SD 1,993 ms). The local processing times and central processing times vary less: they are on average 232 ms (SD 330 ms) and 370 ms (SD 302 ms), respectively, with respective maximums of on average 840 ms and 771 ms.



(a) DIVIDE Monitoring set-up



(b) DIVIDE Local set-up

Figure 4: Part 1 of the results of evaluation scenario 1 that updates the RSP query window parameters based on the monitored query processing time. The processing times are shown for every evaluation of the RSP query, on the timestamp of the data window trigger of the query execution. Moreover, the RAM & CPU usage of the local device is shown. All results are averaged on both axes over the evaluation runs. The error bars represent standard deviations.

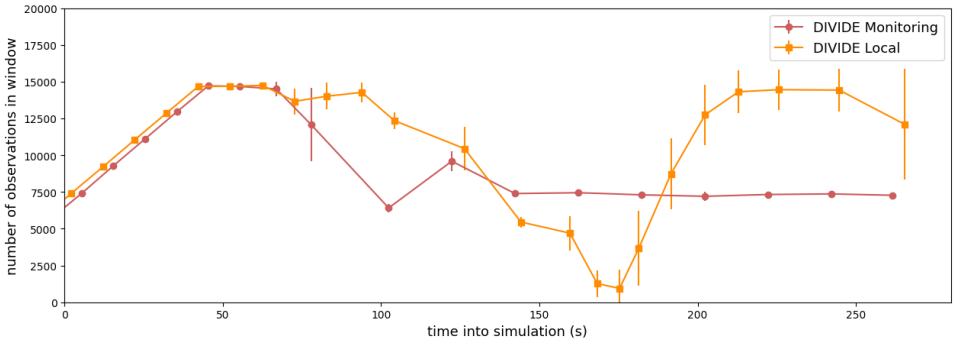


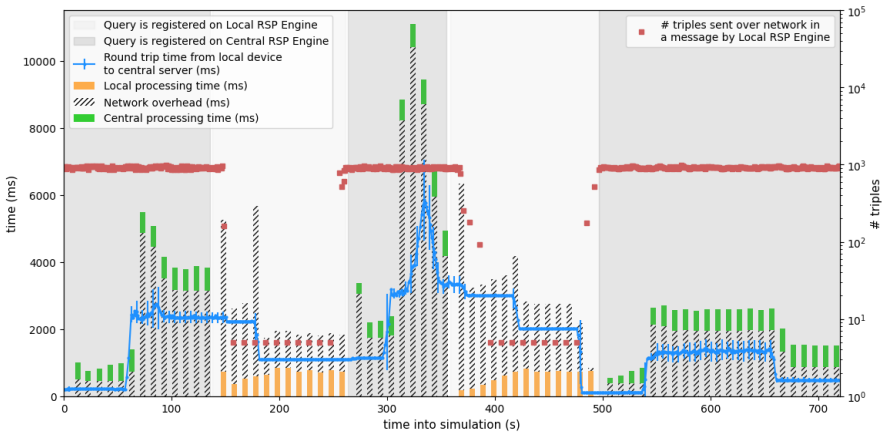
Figure 5: Part 2 of the results of evaluation scenario 1 that updates the RSP query window parameters based on the monitored query processing time. The graph plots the number of observations in the data window of every evaluation of the RSP query, on the timestamp of the data window trigger of the query execution. All results are averaged on both axes over the evaluation runs. The error bars represent standard deviations.

Considering the query processing times for the DIVIDE Local and DIVIDE Central set-up, the average values for those metrics are more constant than in the DIVIDE Monitoring set-up. Concretely, the average local processing time for the DIVIDE Local set-up is 716 ms (SD 77 ms), while the average central processing time for the DIVIDE Central set-up is 647 ms (SD 73 ms). Similarly to the DIVIDE Monitoring set-up, the network overhead largely follows the same pattern as the measured RTT in both set-ups.

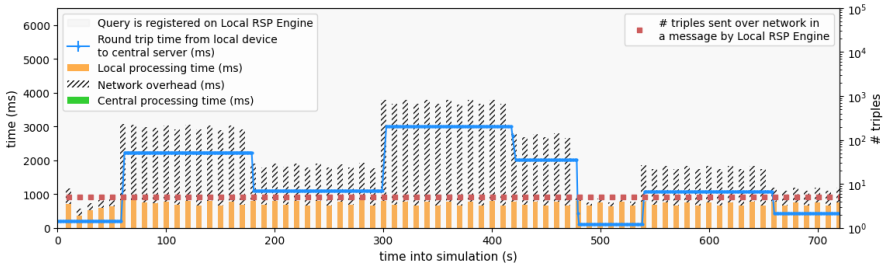
Focusing on the number of triples sent over the network in messages by the Local RSP Engine, big differences can be observed between the different set-ups. For the DIVIDE Monitoring set-up, this number is on average 443,656 triples in total over the full scenario. For the DIVIDE Local and DIVIDE Central set-ups, this average sum of triples is 355 and 658,775, respectively.

Figure 7 shows additional results of the second evaluation scenario, specifically for the DIVIDE Monitoring set-up. For these results, there is only one change to the evaluation set-up, compared to the set-up presented in Section 6.3.2 of which the results are shown in Figure 6. This change is the value of the threshold for the network RTT in the Global Monitor queries deployed on the DIVIDE Global Monitor. In the original scenario, the RSP query is moved to the Local RSP Engine when the RTT is higher than the threshold of 2 seconds, and it is moved back to the Central RSP Engine when the RTT is lower than the threshold of 1.6 seconds.

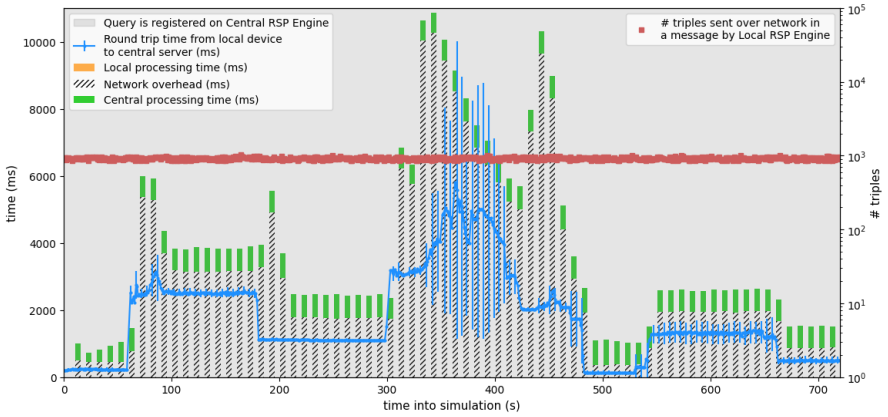
- Figure 7a shows the results of changing these thresholds to 1 and 0.8 seconds, respectively. With this change, the location of the RSP queries changes four times. In the first period with bad networking conditions, the query is moved to the Local RSP Engine. Thereafter, the query does not move back to the Central RSP Engine during the period with better network conditions, because the average RTT does not get below the lower threshold of 0.8 seconds. Only after the second period of bad networking conditions is finished, the RSP query is moved back to



(a) DIVIDE Monitoring set-up



(b) DIVIDE Local set-up



(c) DIVIDE Central set-up

Figure 6: Results of evaluation scenario 2 that updates the RSP query location based on the monitored network RTT. The time duration measurements are shown for every RSP query evaluation, on the timestamp of the data window trigger of the query execution. Moreover, the graphs plot the network RTT and number of triples in outgoing network events. All results are averaged on both axes over the evaluation runs. The error bars represent standard deviations.

the Central RSP Engine. Finally, the query is moved one more time to the Local RSP Engine during the third period of bad networking conditions, and back to the Central RSP Engine after this period. In this set-up, on average 217,713 triples are sent over the network by the Local RSP Engine.
- Figure 7b shows the results of changing these thresholds to 3 and 2.4 seconds, respectively. With this configuration, the RSP query is only moved once to the Local RSP Engine and back, caused by the second period of bad networking conditions that poses the largest restriction on the network capacity. The average total number of triples sent over the network in this set-up is 564,142.

7.3 Performance evaluation of the DIVIDE Local Monitor and DIVIDE Central

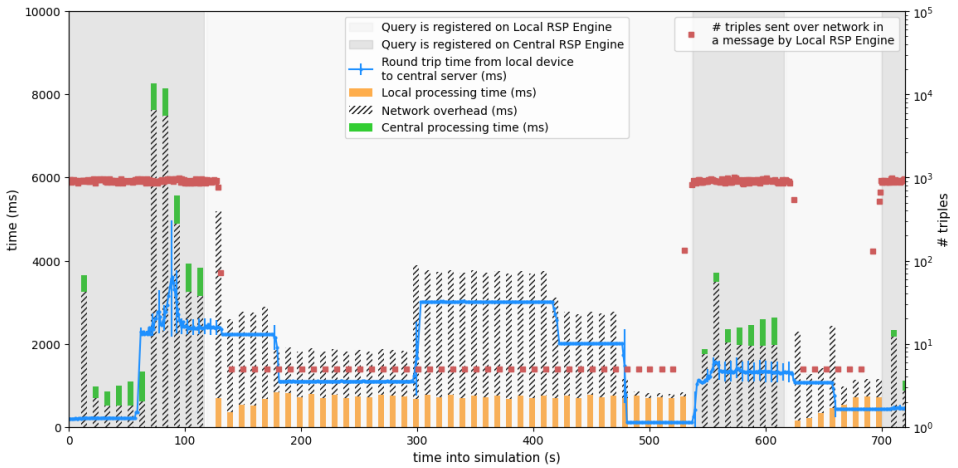
Figure 8 shows the results of the performance evaluation of the DIVIDE Local Monitor, measured on evaluation scenario 1. This figure shows that the CPU usage of the Local Monitor is 10% or lower for on average 72% of the measurements throughout the evaluation runs. Only in on average 1% of the measurements, the CPU usage exceeds 30%. Moreover, the average RAM used by the Local Monitor is 100 MB (SD 2 MB). The execution time of the Local Monitor aggregation query is 1,022 ms on average (SD 349 ms). The average execution time increases after 60 seconds when the nurse call system processes are started in the evaluation scenario.

Figure 9 shows the results of the performance evaluation of the DIVIDE Global Monitor Reasoning Service. It shows the distribution of the event processing time, which is on average 1,066 ms (SD 316 ms). The distribution of this time over the three parts is also shown: on average 52% of the time is spent on adding the Local Monitor events to the DIVIDE Meta Model and performing reasoning, only 1% is spent on average on the execution of the Global Monitor queries, and the remaining 47% is spent on removing the events from the DIVIDE Meta Model and performing reasoning again.

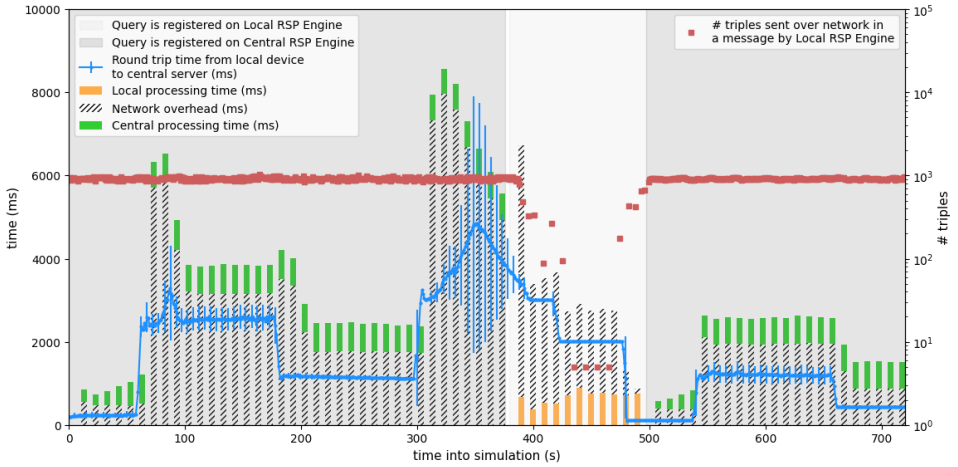
Finally, Figure 10 presents the distribution of the duration of the tasks that can be issued by the DIVIDE Global Monitor, over multiple runs. For a query location update task, moving a query to the Local RSP Engine and the Central RSP Engine takes on average 18,254 ms (SD 1,661) and 6,395 ms (SD 3,383 ms), respectively. For a window parameter update task, this value is on average 7,093 ms (SD 2,134 ms). Specifically for this task, the part of the task duration spent on the performed window parameter substitution of the query derivation (step 5 as explained in Section 2.2) is 184 ms (SD 28 ms).

8 Discussion

DIVIDE is a semantic component that can be deployed in a semantic IoT platform to derive and manage the relevant queries for the stream processing components of the platform in an automatic, adaptive and context-aware way. In our previous work on DIVIDE, we have already demonstrated the added value of using DIVIDE Core over other state-of-the-art set-ups that involve real-time semantic reasoning on IoT

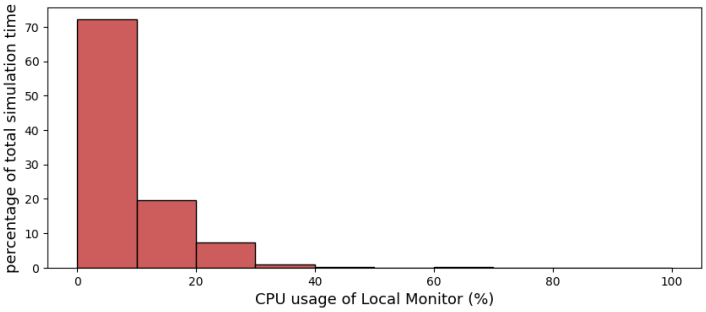


(a) Lower RTT threshold in Global Monitor queries: move to central if value is 1 second or higher, move to local if value is 0.8 seconds or lower

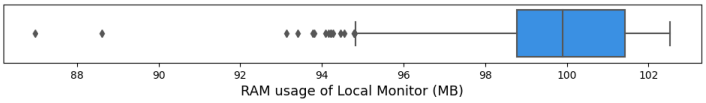


(b) Higher RTT threshold in Global Monitor queries: move to central if value is 3 seconds or higher, move to local if value is 2.4 seconds or lower

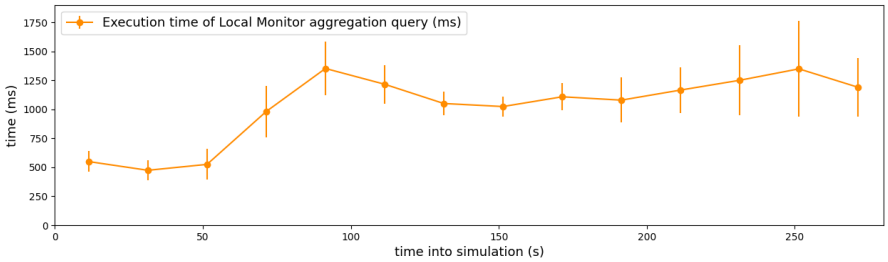
Figure 7: Additional results of evaluation scenario 2 that updates the RSP query location based on the monitored network RTT. These results are only shown for the DIVIDE Monitoring set-up. The only change to the evaluation set-up compared to the results shown in Figure 6 are the thresholds for the RTT in the Global Monitor queries that decide when the RSP query should be moved between the Local RSP Engine and Central RSP Engine. The time duration measurements are shown for every RSP query evaluation, on the timestamp of the data window trigger of the query execution. Moreover, the graphs plot the network RTT and number of triples in outgoing network events. All results are averaged on both axes over the evaluation runs. The error bars represent standard deviations.



(a) Histogram with distribution of Local Monitor CPU usage over the full scenario and multiple runs



(b) Distribution of Local Monitor memory (RAM) usage over the full scenario and multiple runs

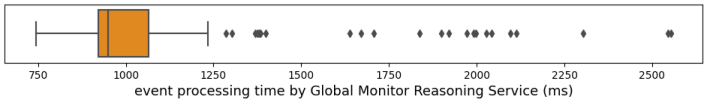


(c) Timeline of execution time of Local Monitor aggregation query, averaged over multiple runs of the scenario (error bars represent standard deviations)

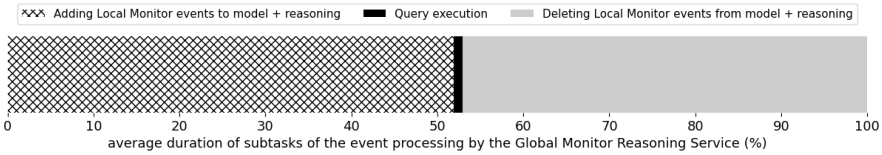
Figure 8: Results of the performance evaluation of the DIVIDE Local Monitor, measured on evaluation scenario 1

data streams [6]. Hence, the focus of this paper and its evaluations is on the improved methodological design of DIVIDE, involving the different central and local subcomponents that allow performing situational context monitoring to manage the configuration and distribution of the stream processing queries across the network. Therefore, the different evaluations performed in this paper compare a set-up involving all subcomponents of DIVIDE (DIVIDE Monitoring) with two set-ups that only involve DIVIDE Core and do not perform any situational context monitoring (DIVIDE Central and DIVIDE Local).

The first evaluation scenario focuses on adapting the window parameter configuration of queries, based on the performance of the query evaluation at the Local RSP Engine. The local device in the evaluation is a Raspberry Pi, which is a typical IoT



(a) Distribution of the event processing times over the full scenario and multiple runs



(b) Average duration of subtasks of the event processing times, averaged over the full scenario and multiple runs

Figure 9: Results of the performance evaluation of the Reasoning Service of the DIVIDE Global Monitor, measured on evaluation scenario 1

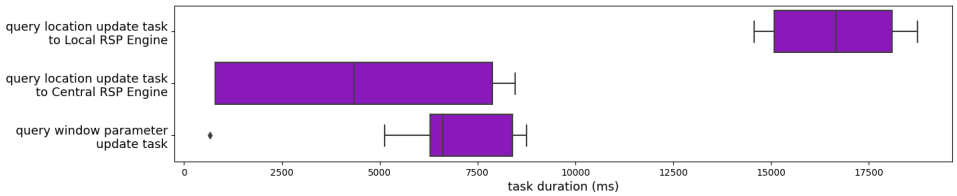


Figure 10: Distribution of the duration of the tasks that can be issued by the DIVIDE Global Monitor, over multiple runs. The duration of the window parameter update task is measured in the runs of evaluation scenario 1, the duration of the query location updates in the runs of evaluation scenario 2.

device with few resources. The evaluation results in Figure 4 and 5 show that the DIVIDE Monitoring set-up can intelligently update the query window parameters, according to the end user definitions in the Global Monitor queries. Moreover, the results show that by lowering the data window size and execution frequency (i.e., increasing the sliding step) whenever the device does not have enough resources to keep up with the data, DIVIDE helps ensuring that the query remains running correctly in terms of frequency and data in its input window. In comparison, the DIVIDE Local set-up cannot dynamically alter the window parameters, and thus cannot keep up with the data volume and velocity, as soon as the resource usage on the Raspberry Pi starts increasing because of the nurse call system processes. This might lead to ignored data and incorrect, delayed data processing, as shown in the evaluation results. In the worst case, it could even lead to crashes of the Local RSP Engine.

Looking at the Global Monitor query that updates the window parameters, it should be noted that it currently does not prevent the given DIVIDE task from being issued multiple times for the same DIVIDE component and DIVIDE query. This could

lead to an undesirable configuration of the window parameters where the window size is smaller than the sliding step, causing data to be ignored in the calculation of the activity index. Hence, depending on the use case, the Global Monitor query could be altered to avoid this. Nevertheless, it is important to realize that the Global Monitor queries can avoid the system from crashing. Hence, depending on the use case, it might be allowed to ignore some data to prevent the system from crashing and thus processing no data at all. Moreover, although this was assumed to be impossible in the evaluated scenario, note that moving the RSP query to the central device might also be a valid solution if the local query performance keeps on decreasing. Hence, in ideal circumstances, multiple Global Monitor queries are deployed that jointly consider RSP performance and networking conditions.

The second evaluation scenario focuses on updating the query location in the network based on the monitored network properties. By varying the properties of the relevant network links, a realistically varying networking capacity has been simulated. The results in Figure 6 demonstrate the differences between the set-ups. In the DIVIDE Local set-up, the networking overhead is the smallest off all set-ups throughout the scenario, since only the query results should be forwarded to the central device. The local processing time is also rather constant and well below 1 second. However, in this particular use case, this set-up is not preferred. This is because the alarm center wants to have as much raw data on the central servers as possible, to ensure patient security. The alternative set-up without the monitoring subcomponents of DIVIDE is DIVIDE Central. As shown in the results, this set-up is not ideal either, for multiple reasons. First, since all raw data needs to be forwarded to the central server, the network is heavily burdened. In some evaluation runs (as shown in Appendix D), this leads to heavily accumulating delays in the data processing, which is problematic when real-time follow-up is required. In the scenario timeline, the bad networking conditions did not take longer than 2 minutes. In contrast, these bad conditions could persist for a much longer period of time in real-world scenarios, posing the risk of the networking overhead and thus processing delays to even further accumulate. A second disadvantage of the DIVIDE Central set-up is the associated server cost. In the evaluation results, the processing times are well below 1 second. However, it should be noted that the evaluation was performed with only three local devices. In a real-world set-up, this number is likely to be at least an order of magnitude larger. This would put much higher requirements on the central server resources to guarantee the same level of performance. In contrast to the DIVIDE Local and DIVIDE Central set-ups, the DIVIDE Monitoring set-up combines the best of both worlds by adaptively changing the query distribution based on the network conditions, taking into account the use case requirements configured by the end users in the Global Monitor queries. Specifically for the considered homecare monitoring use case, this set-up optimally balances the trade-off between patient security and server cost. It does this in a dynamic environment, taking situational context into account when balancing the trade-off: if the network does not allow forwarding raw data even if this is to be preferred, moving the query locally and forwarding only aggregated insights is still better than having no up-to-date, correct insights centrally. In other words, given the current situational

context across the IoT platform *which cannot be altered*, DIVIDE allows optimally balancing all use case requirements defined through Global Monitor queries *by altering the query distribution and configuration.*

The evaluation use case of this paper focuses on the homecare monitoring task of calculating a patient's activity index. This is an aggregation of raw accelerometer data, and thus inherently entails less information. This means that the same activity index value calculated over a data window of 60 seconds could correspond to different activity patterns over the course of that minute. Depending on a patient's medical condition, the priority of knowing this exact activity pattern and thus having access to the raw data differs. Therefore, the second evaluation scenario has been run for different versions of the Global Monitor queries, containing other RTT thresholds for moving the query between the Local and Central RSP Engine. Compared to the original results presented in Figure 6, Figure 7a and Figure 7b show the results for running the evaluation scenario with lower and higher RTT thresholds, respectively. In essence, higher RTT thresholds in these Global Monitor queries correspond to a higher priority of having the raw data centrally and thus running the RSP query centrally. As explained in Section 6.1.1, the higher thresholds could correspond to a fall-prone patient, where access to the raw data is the most important and thus only sacrificed when the networking conditions are really bad. On the other hand, for a dementia patient, lower granularity insights into the patient's activity pattern suffice and thus lower RTT thresholds, that allow moving RSP queries to the local device more quickly, are allowed.

In the networking evaluation, the results about the number of the triples sent over the network in the different set-ups clearly demonstrate the impact of the query location on the network burden. In this scenario, when running the query locally, only 5 triples are sent over the network every 10 seconds. Instead, when running the query centrally, on average 9300 triples are sent over the network in 10 seconds (on average 186 observations per second for 10 seconds, with 5 triples per observation). This demonstrates, for a realistic data volume and velocity, the relevance and importance of DIVIDE allowing to move the RSP query locally when networking conditions get worse. For completeness, it should be noted that the choice of RDF triple language can also impact the evaluation results. By using another language than N-Triples such as RDF/Turtle and optimally exploiting the usage of prefixes, the size of the messages can be lowered to increase network efficiency. However, this would happen at the expense of requiring more central resource-intensive parsing.

Inspecting the results of the performance evaluation of the DIVIDE Local Monitor in Figure 8, it is clear that the Local Monitor has an acceptable usage of CPU and RAM resources, even on a low-end device such as a Raspberry Pi. As can be observed, the execution times of the aggregation queries are impacted by the other active processes on the system, but are still below 1.5 seconds on average when the CPU usage is 100%.

The performance evaluation results for the Global Monitor in Figure 9 show that the event processing times on the Global Monitor Reasoning Service are just above 1 second on average, with a few outliers. On average 99% of this time is spent on adding the monitoring data to the DIVIDE Meta Model and removing it again. Both steps involve semantic reasoning by Apache Jena. Implementing this Reasoning Service with

more efficient state-of-the-art semantic reasoners would allow a significant decrease of the total event processing times, especially if the reasoner supports incremental reasoning. Such reasoners only need to perform semantic reasoning on the updated parts of the data model, instead of having to perform the whole semantic reasoning process again on every update like Apache Jena does.

Inspecting the distribution of the duration of the tasks that can be issued by the DIVIDE Global Monitor in Figure 10, it is clear that these tasks take a while. This duration involves the event processing on the Global Monitor Reasoning Service until the output of the Global Monitor query is generated, the parsing by the DIVIDE Monitor translator, and the actual task execution by DIVIDE Core. The query location update tasks mainly involve communication over the network with both RSP engines. The move to the Local RSP Engine is issued in bad networking conditions, which explains the high durations with an average of more than 18 seconds. For the query window parameter update task, the final query derivation steps performed by DIVIDE Core take only 184 ms on average. Hence, the remaining seconds are also spent on sending the query updates to the Local RSP Engine. The high task duration can thus be explained by taking into account the 100% CPU usage on the local device when the query update requests are received. In addition, all high task durations should be put into perspective. First of all, during the query updates, all received streaming data is buffered by the Local RSP Engine, so that no data is ignored or removed. Moreover, the configuration of the Global Monitor queries ensures that the tasks are issued for a reason: even if there is a single larger gap than desired between two query executions, this is still better than not executing the task and leaving the RSP query configuration and distribution as is. By letting the end user define the Global Monitor queries and thus the thresholds of the monitored situational context properties, one can intelligently tweak when and how often the query configuration and distribution is updated.

The Global Monitor queries are the main tool for end users to configure the behavior of the monitoring subcomponents of DIVIDE. Hence, it is important to make the process of defining those Global Monitor queries as user-friendly as possible. In this paper, we have made a first attempt in improving the user-friendliness by suggesting a BNF grammar to define the actuation rules for the Global Monitor queries. The goal of this grammar is to hide the inner semantic details of how the monitor measurements, the DIVIDE meta-information and the issued tasks are described with the DIVIDE Meta Model ontology. These semantic details are irrelevant for end users, and require knowledge of Semantic Web technologies such as RDF and SPARQL. Using such a grammar, an end user only needs to know the DIVIDE terminology, what tasks can be issued, and what properties are being monitored. In addition, BNF grammar rules are less verbose than SPARQL queries, increasing the user-friendliness.

To put DIVIDE into production, further steps in improving the user-friendliness should be made. In general, user-friendly interfacing is required to optimally configure the system with all DIVIDE components, DIVIDE queries and Global Monitor queries. Such an interfacing system could also automatically suggest relevant Global Monitor queries based on the configured use case requirements, for example to avoid

the RSP engine from crashing or accumulating processing delay in case of high resource usage. An interesting addition would be a priority-based system, where the Global Monitor query conditions and thresholds for updating the query distribution would be dependent on assigned priorities of having central access to the raw sensor data. Such a priority could be assigned based on the window parameters of the RSP queries, as they define the degree of information loss between the raw data and the outputs of the RSP queries: the lower the size of the data window, the smaller the information loss would be if only the aggregated query outputs are available and not the original raw data. Other options could be to assign priorities per DIVIDE query, or even based on the medical conditions in the patient profile in healthcare applications. Furthermore, in some applications, changing RSP query configuration details such as window parameters is not allowed. This is for example the case if the query outputs are processed by a machine learning algorithm that requires certain input features to be aggregations made on a data window of a specific size. The interfacing system should then allow an end user to define such priorities and conditions in a user-friendly way, and automatically translate them into the correct Global Monitor queries. It should be noted that some of these suggestions would require some small changes to the design of DIVIDE, for example to add some specific parameters or priorities in the semantic descriptions of DIVIDE queries or DIVIDE components. As for the rest, everything is readily available in the methodological design of DIVIDE to make such more user-friendly interfacing possible.

As the design of the subcomponents of DIVIDE is generic and modular, it is possible to easily extend its functionality in the future. Adding new parameters to be monitored to the existing individual monitors only requires a few adaptations. First, the new properties should be added to the `Monitoring` module of the DIVIDE Meta Model ontology. Second, the implementation of the DIVIDE Local Monitor should be updated: the semantic mapper should include the new properties, and the individual monitors should continuously collect measurements for the new properties and output them in the required JSON message format. Due to the modular design, the implementation of the individual monitors can also be easily replaced with another implementation, without requiring further changes. New individual monitors can also be integrated in a similar way. Some relevant new monitoring properties to be included as future work are the energy consumption of the processing devices, and whether or not the network connection between the local and central device is metered. The latter can be especially relevant when extending the architecture to mobile devices, to avoid that high volumes of raw data are being sent over a metered connection. Focusing on the Local Monitor aggregation queries, new aggregations can also be easily added to the existing implementation. The current query already generically aggregates all properties using the DIVIDE Meta Model ontology, such that no changes are required if new properties are being monitored.

Future extensions to the methodological design of DIVIDE could also consist of allowing the Global Monitor to take additional meta-information into account. For example, in a more sophisticated cascading architecture, more details about the data and query deployment might be relevant to be taken into account by the Global Monitor queries. To enable this, several changes are required. First, the `DivideCore` module

of the Meta Model ontology needs to be extended. Moreover, the implementation of the integration of the DIVIDE Meta Model needs to manage the corresponding new triples in the Meta Model, and the implementation of DIVIDE Core needs to be altered to keep the new meta-information in the Meta Model up-to-date at all times. Another possible extension to the Global Monitor design could be to perform monitoring on the central device as well. This could include monitoring the performance of the Central RSP Engine, as well as the correct inner workings and network communication of the DIVIDE implementation. In addition, future research could also look into updating the query distribution task issued by the DIVIDE Global Monitor in a more fine-grained manner, e.g. by including multiple edge devices into the architecture. Related to this, new RSP or reasoning engines could also be deployed on or removed from components in the network, based on the monitored situational context. This would further increase the dynamism of the query distribution. The existing implementation of deploying Local Monitor instances over SSH across the IoT network could be leveraged for this.

To start using DIVIDE in a real-world production environment, several changes to its design and implementation are required. On design level, support should be added to update the window parameters of queries with multiple stream windows, and stream windows that specify the data window interval relative to the current time. This is not integrated into the current design of DIVIDE, as this was not required to demonstrate its capabilities and validate the research questions. On implementation level, several general and specific improvements are required. These improvements were out of scope for this work, as the focus of this research is on demonstrating the possibilities of the monitoring aspect of DIVIDE on a methodological level and validating it with a first implementation. A first specific possible improvement is the implementation of the network monitor. In a production environment, it should monitor multiple networking properties in a more sophisticated way. To achieve this, one could look into using existing network monitoring tools. Second, the implementation of the DIVIDE Monitor Translator could be improved to support multiple window parameters with different variable names. Moreover, the current chronological parsing and forwarding of issued tasks could be replaced by an improved algorithm that intelligently handles conflicting, duplicate and frequently recurring tasks. Third, enabling the RSP Engine Monitor requires integrating the original implementation of this RSP engine with our RSP engine wrapper implementation. This might require small adaptations to the source code of the RSP engine, which does not impose an issue as the source code of most engines is open source. If this is not the case, the required monitoring data could also be extracted from the logs of the RSP engine wrapper.

9 Conclusion

This paper has presented DIVIDE, which is a semantic component that can be deployed in a cascading reasoning architecture of a semantic IoT platform. In the work, the methodological design and a first implementation of DIVIDE is discussed and evaluated on a homecare monitoring use case. Looking back at the research objectives

outlined in Section 1, we can conclude that we have achieved those in the following ways:

1. The design of the monitoring subcomponents of DIVIDE enables the continuous monitoring of various relevant parameters of the situational context in which tasks are deployed across the stream processing components in the IoT network. Using the designed DIVIDE Meta Model ontology, the Local Monitor can monitor these parameters through multiple individual monitors: network characteristics with the Network Monitor, resource usage of the stream processing devices with the Device Monitor, and data stream properties and real-time performance of the stream processing components with the RSP Engine Monitor.
2. By forwarding aggregations of the Local Monitor observations to the Global Monitor Reasoning Service and continuously evaluating the configured Global Monitor queries on the DIVIDE Meta Model containing meta-information about the system, actions can be taken based on the monitored situational context. These actions can be defined by DIVIDE tasks specified in the output of the Global Monitor queries. Two tasks are supported in the current design of DIVIDE: updating the window parameter configuration of the deployed RSP queries (i.e., updating the query's window size and/or sliding step), and updating the distribution of those queries by moving them between the Local and Central RSP Engine. The presented evaluation results prove that these tasks can be successfully performed by our implementation of DIVIDE.
3. Through the definition and configuration of use case specific Global Monitor queries, an end user can dynamically configure how the situational context parameters should influence the RSP query configuration and distribution in the network. This way, the actuation can be mapped to the requirements of every individual use case. By suggesting a BNF grammar to define these queries as actuation rules, no end user knowledge of the Meta Model ontology concepts or Semantic Web technologies should be required. The evaluation results presented in this paper show that the dynamic approach to configuring RSP query window parameters and the RSP query distribution in the network, allows reacting to situational context parameters such as varying device resource usage or networking conditions. The results demonstrate that the usage of a set-up with the monitoring subcomponents of DIVIDE can better deal with use case specific requirements in such dynamic environments, compared to alternative static set-ups. In summary, DIVIDE can increase the percentage of time that an optimal balance of use case specific trade-offs is guaranteed. This way, it automatically achieves more efficient stream processing.
4. Through our design of DIVIDE, we have laid the foundation of monitoring the relevant situational context information on local and edge components, and updating the RSP query configuration and distribution based on this information in an automated way. Laying this foundation is done by generically designing the Meta Model ontology and subcomponents of DIVIDE. Given this generic and modular design, new properties can be easily monitored by extending the Meta Model ontology and locally extending or adding individual monitors.

5. By building further on the design and implementation of our previous work on DIVIDE [6], the presented design of DIVIDE allows deriving and managing the RSP queries in an adaptive and context-aware way, based on domain knowledge and use case specific context.

Future work presents multiple interesting directions. First, to put DIVIDE into production, the user-friendliness of the Global Monitor query definition should be further improved by designing an interfacing system that intelligently translates use case requirements and assigned priorities into the deployed queries. Second, several design and implementation improvements should be performed to make the system more robust and complete. Possible improvements include improving the monitoring of network characteristics and supporting queries with multiple input data windows. Third, the system design could be extended to allow for dynamic deployment of query engines across the IoT network depending on the monitored situational context.

Declarations

Funding

This research is part of the imec.ICON project PROTEGO (HBC.2019.2812), co-funded by imec, VLAIO, Televic, Amaron, Z-Plus and ML2Grow. This research is also partly funded by the FWO Project FRACTION (nr. G086822N) and by the BOF FRACTION (nr. BOF/24J/2021/388).

Competing interests

The authors declare no conflict of interest.

Authors' contribution

M.D.B. performed the full methodological design of DIVIDE, including the design of the Meta Model ontology and all subcomponents of DIVIDE. M.D.B. performed the implementation of DIVIDE, executed the evaluation experiments, analyzed the results and wrote the paper. F.O. was actively involved in the design, implementation and evaluation phase. F.O. and F.D.T. supervised the study, reviewed the paper and gave some valuable suggestions to the work and paper. All authors read and approved the final manuscript.

Data, material and code availability

Supportive information relevant to the evaluation set-ups of this paper is publicly available at https://github.com/IBCNServices/DIVIDE/tree/master/jnsm2023. This page also refers to multiple other publicly available pages. These include the DIVIDE Meta Model ontology at https://github.com/IBCNServices/DIVIDE/tree/master/meta-model, the source code of our first implementation of the different DIVIDE subcomponents at https://github.com/IBCNServices/DIVIDE/tree/master/src, additional details of the DAHCC ontology at https://dahcc.idlab.ugent.be, and the described evaluation dataset used for simulation at https://dahcc.idlab.ugent.be/

dataset.html. A dedicated tag of the DIVIDE repository is created at https://github.com/IBCNServices/DIVIDE/releases/tag/jnsm-2023, which represents the version of implementation (`src` folder) and Meta Model ontology (`meta-model` folder) resulting from the current work.

Editorial Policies for:

Springer journals and proceedings: https://www.springer.com/gp/editorial-policies

Nature Portfolio journals: https://www.nature.com/nature-research/editorial-policies

Scientific Reports: https://www.nature.com/srep/journal-policies/editorial-policies

BMC journals: https://www.biomedcentral.com/getpublished/editorial-policies

Appendix A Examples of the DIVIDE Meta Model triples

This appendix gives some examples of how the DIVIDE meta-information and monitoring observations can be represented in the DIVIDE Meta Model with the modules of the Meta Model ontology:

- Listing 8 shows how the relevant meta-information of DIVIDE is semantically described using the concepts of the `DivideCore` ontology module.
- Listing 9 illustrates how a measurement of the average execution time of an RSP query can be semantically described with the concepts of the `Monitoring` ontology module.

Listing 8: Example of how the `DivideCore` module of the Meta Model ontology is used to model all relevant meta-information about DIVIDE in the DIVIDE Meta Model. This example considers a DIVIDE engine with one DIVIDE query and one DIVIDE component. One RSP query derived from this DIVIDE query is registered to the Local RSP Engine of this DIVIDE component. The triples are presented in RDF/Turtle syntax.

```
# additional, temporary prefixes to make this listing more readable
@prefix divide-engine:
    <https://divide.idlab.ugent.be/meta-model/entity/engine/> .
@prefix divide-component:
    <https://divide.idlab.ugent.be/meta-model/entity/component/> .
@prefix divide-query:
    <https://divide.idlab.ugent.be/meta-model/entity/divide-query/> .
@prefix rsp-engine:
    <https://divide.idlab.ugent.be/meta-model/entity/rsp-engine/> .
@prefix device: <https://divide.idlab.ugent.be/meta-model/entity/device/> .

# DIVIDE engine
divide-engine:44c52eb3-3a03-4d11-8c96-e1854196e4e7
    a divide-core:DivideEngine ;
    divide-core:hasID "44c52eb3-3a03-4d11-8c96-e1854196e4e7" ;
    divide-core:isHostedBy device:10.42.0.112 .

# DIVIDE component
divide-component:10.42.0.35-8100-
```

```
    a divide-core:DivideComponent ;
    divide-core:hasID "10.42.0.35-8100-" ;
    divide-core:hasCentralRspEngine rsp-engine:central ;
    divide-core:hasLocalRspEngine rsp-engine:10.42.0.35-8100- ;
    divide-core:isHostedBy device:10.42.0.35 .

# device of DIVIDE component
device:10.42.0.35 a saref-core:Device ;
    divide-core:hasIPAddress "10.42.0.35" .

# Local RSP Engine of DIVIDE component
rsp-engine:10.42.0.35-8100- a divide-core:RspEngine ;
    divide-core:hasServerPort "8100"^^xsd:integer .

# DIVIDE query
divide-query:activity-index a divide-core:DivideQuery ;
    divide-core:hasName "activity-index" ;
    divide-core:hasQueryDeployment <https://divide.idlab.ugent.be/meta-model/entity
        /divide-query/activity-index/deployment/10.42.0.35-8100-> .

# RSP query on DIVIDE component, derived from DIVIDE query
<https://divide.idlab.ugent.be/meta-model/entity/rsp-engine/10.42.0.35-8100-/rsp-
    query/10f67219-36c9-4bda-97f4-dc9420537348>
    a divide-core:RspQuery ;
    divide-core:hasName "Q0mmlll" ;
    divide-core:hasID "10f67219-36c9-4bda-97f4-dc9420537348" ;
    divide-core:hasAssociatedComponent divide-component:10.42.0.35-8100- ;
    divide-core:hasCorrespondingDivideQuery divide-query:activity-index ;
    divide-core:isRegisteredTo rsp-engine:10.42.0.35-8100- ;
    divide-core:hasWindowSizeInSeconds "80"^^xsd:integer ;
    divide-core:hasQuerySlidingStepInSeconds "10"^^xsd:integer ;
    divide-core:hasInputStreamWindow <https://divide.idlab.ugent.be/meta-model/
        entity/rsp-engine/10.42.0.35-8100-/rsp-query/10f67219-36c9-4bda-97f4-
        dc9420537348/stream-window/http%3A%2F%2Fprotego.ilabt.imec.be%2Fidlab.
        homelab-RANGE+80s+STEP+10s> .

# local deployment of RSP query
<https://divide.idlab.ugent.be/meta-model/entity/divide-query/activity-index/
    deployment/10.42.0.35-8100->
    a divide-core:QueryDeployment ;
    saref-core:isAbout divide-component:10.42.0.35-8100- ;
    divide-core:hasQueryLocation <https://divide.idlab.ugent.be/meta-model/entity/
        divide-query/activity-index/deployment/10.42.0.35-8100-/location/
        LocalLocation> .
<https://divide.idlab.ugent.be/meta-model/entity/divide-query/activity-index/
    deployment/10.42.0.35-8100-/location/LocalLocation>
    a divide-core:LocalLocation .

# stream window of RSP query
<https://divide.idlab.ugent.be/meta-model/entity/rsp-engine/10.42.0.35-8100-/rsp-
    query/10f67219-36c9-4bda-97f4-dc9420537348/stream-window/http%3A%2F%2Fprotego.
    ilabt.imec.be%2Fidlab.homelab-RANGE+80s+STEP+10s>
    a divide-core:StreamWindow ;
    divide-core:hasInputStream <https://divide.idlab.ugent.be/meta-model/entity/rsp
        -engine/10.42.0.35-8100-/rdf-stream/http%3A%2F%2Fprotego.ilabt.imec.be%2
        Fidlab.homelab> ;
    divide-core:hasWindowDefinition "RANGE 80s STEP 10s" ;
    divide-core:hasWindowSizeInSeconds "80"^^xsd:integer ;
    divide-core:hasQuerySlidingStepInSeconds "10"^^xsd:integer .

# RDF stream in stream window of RSP query
<https://divide.idlab.ugent.be/meta-model/entity/rsp-engine/10.42.0.35-8100-/rdf-
    stream/http%3A%2F%2Fprotego.ilabt.imec.be%2Fidlab.homelab>
    a divide-core:RdfStream ;
    divide-core:hasStreamName
        "http://protego.ilabt.imec.be/idlab.homelab" .
```

Listing 9: Example of how the `Monitoring` module of the DIVIDE Meta Model ontology allows modeling the average execution time of an RSP query, represented in RDF/Turtle syntax

```
<https://divide.idlab.ugent.be/10.42.0.35-8100-/rsp_query_execution_time/avg/obs64>
    a saref-core:Measurement ;
    saref-core:hasValue "0.561"^^xsd:float ;
    om:hasAggregateFunction om:average ;
    saref-core:hasTimestamp "2023-04-02T08:33:22"^^xsd:dateTime ;
    saref-core:isMeasuredIn om:second-Time ;
    saref-core:relatesToProperty [ a monitoring:RspQueryExecutionTime ] ;
    saref-core:isMeasurementOf  <https://divide.idlab.ugent.be/meta-model/entity/
        rsp-engine/10.42.0.35-8100-/rsp-query/a76f39a8-ff76-453d-9589-a10b7d6ea942
        > .
```

Appendix B Additional implementation details

This appendix provides some additional details about our implementation of the DIVIDE Local Monitor:

- Listing 10 shows an example configuration of the DIVIDE Local Monitor.
- Listing 11 presents the C-SPARQL aggregation query that is deployed on our implementation of the Local Monitor RSP Engine.

Listing 10: Example JSON configuration of the DIVIDE Local Monitor

```
{
  "component_id": "10.10.145.9-8175-",
  "device_id": "10.10.145.9",
  "monitor": {
    "rsp": true,
    "network": true,
    "device": true
  },
  "local": {
    "rsp_engine": { "monitor": { "ws_port": 54548 } },
    "public_network_interface": "wlp1s0"
  },
  "central": {
    "monitor_reasoning_service": {
      "protocol": "http",
      "host": "10.10.145.233",
      "port": 54555,
      "uri": "/globalmonitorreasoningservice"
    }
  }
}
```

Listing 11: C-SPARQL aggregation query that is continuously evaluated on the implementation of the DIVIDE Local Monitor RSP engine

```
CONSTRUCT {
    [ a saref-core:Measurement ;
        saref-core:hasValue ?minV ;
        om:hasAggregateFunction om:minimum ;
        saref-core:isMeasuredIn ?unit ;
        saref-core:relatesToProperty [ a ?prop ] ;
        saref-core:isMeasurementOf ?featureOfInterest ;
        saref-core:hasTimestamp ?now ] .

    [ a saref-core:Measurement ;
        saref-core:hasValue ?maxV ;
```

```
        om:hasAggregateFunction om:maximum ;
        saref-core:isMeasuredIn ?unit ;
        saref-core:relatesToProperty [ a ?prop ] ;
        saref-core:isMeasurementOf ?featureOfInterest ;
        saref-core:hasTimestamp ?now ] .

    [ a saref-core:Measurement ;
        saref-core:hasValue ?avgV ;
        om:hasAggregateFunction om:average ;
        saref-core:isMeasuredIn ?unit ;
        saref-core:relatesToProperty [ a ?prop ] ;
        saref-core:isMeasurementOf ?featureOfInterest ;
        saref-core:hasTimestamp ?now ] .
}
FROM STREAM <https://divide.idlab.ugent.be/monitor/local> [RANGE 60s STEP 20s]
WHERE {
    BIND (NOW() as ?now)
    {
        SELECT ?featureOfInterest ?prop ?unit
               (MIN(?v) AS ?minV)
               (MAX(?v) AS ?maxV)
               (AVG(?v) AS ?avgV)
        WHERE {
            ?m a saref-core:Measurement ;
               saref-core:hasValue ?v ;
               saref-core:isMeasuredIn ?unit ;
               saref-core:relatesToProperty [ a ?prop ] ;
               saref-core:isMeasurementOf ?featureOfInterest .
        }
        GROUP BY ?featureOfInterest ?prop ?unit
    }
}
```

Appendix C Semantic details of the evaluation use case

This appendix provides the semantic details about the homecare monitoring use case of the evaluations performed in this paper, which is discussed in Section 6.1:

- Listing 12 gives an overview of additional prefixes used in the listings of this appendix. This concerns prefixes specific to the evaluation use case that were not yet listed in Listing 1.
- Listing 13 and 14 present relevant definitions contained in the additional ontology module of the DAHCC ontology that is designed for the evaluations.
- Listing 15 presents the most important triples in the use case context of the described evaluation scenarios.
- Listing 16 and 17 present the DIVIDE query that is added to DIVIDE Core during the executed evaluation scenarios. This DIVIDE query is used by DIVIDE to derive the RSP query that continuously measures the activity index of the monitored patient. Listing 16 presents the sensor query rule of this DIVIDE query, while Listing 17 presents its goal.

Listing 12: Semantic description of the additional prefixes used in the listings of this appendix

```
# DAHCC ontology including the additional ontology module
@prefix ActivityRecognition:
    <https://dahcc.idlab.ugent.be/Ontology/ActivityRecognition/> .
```

```
@prefix MonitoredPerson:
    <https://dahcc.idlab.ugent.be/Ontology/MonitoredPerson/> .
@prefix Sensors: <https://dahcc.idlab.ugent.be/Ontology/Sensors/> .
@prefix SensorsAndWearables:
    <https://dahcc.idlab.ugent.be/Ontology/SensorsAndWearables/> .
@prefix HealthParameterCalculation:
    <https://dahcc.idlab.ugent.be/Ontology/HealthParameterCalculation/> .

# instances in use case scenario
@prefix : <http://divide.ilabt.imec.be/idlab.homelab/> .
@prefix patients: <http://divide.ilabt.imec.be/idlab.homelab/patients/> .
@prefix Homelab:
    <https://dahcc.idlab.ugent.be/Homelab/SensorsAndActuators/> .
@prefix HomelabWearable:
    <https://dahcc.idlab.ugent.be/Homelab/SensorsAndWearables/> .

# additional imports of DAHCC ontology modules
@prefix saref4ehaw: <https://saref.etsi.org/saref4ehaw/> .
@prefix time: <http://www.w3.org/2006/time#> .

# definitions within DIVIDE
@prefix sd: <http://idlab.ugent.be/sensdesc#> .
@prefix sd-query: <http://idlab.ugent.be/sensdesc/query#> .
@prefix sh: <http://www.w3.org/ns/shacl#> .
```

Listing 13: Overview of how different subclass and equivalence relations are defined in the additional ontology module of the DAHCC ontology created for the evaluations in this paper. These definitions allow a semantic reasoner to derive when a certain health parameter is relevant to be monitored for a patient. To improve readability, all definitions are listed in Manchester syntax and the `HealthParameterCalculation:` prefix is replaced by the `:` prefix.

```
:ActivityIndex SubClassOf: :HealthParameter

:RelevantActivityIndex SubClassOf: :ActivityIndex
:RelevantActivityIndex EquivalentTo:
    :ActivityIndex and
    (:calculationMadeFor some :PatientThatRequiresMovementMonitoring)

:PatientThatRequiresMovementMonitoring EquivalentTo:
    saref4ehaw:Patient and
    (saref4ehaw:hasChronicDisease some (
        saref4ehaw:ChronicDisease and
        (:requiresMovementMonitoring value true)))
```

Listing 14: Overview of important ontology definitions in the additional ontology module of the DAHCC ontology created for the evaluations in this paper, presented in RDF/Turtle syntax. These definitions describe a health parameter calculator system that allows calculating a patient's activity index whenever relevant. To improve readability, the `HealthParameterCalculation:` prefix is replaced by the `:` prefix.

```
# define health parameter calculator
:HealthParameterCalculator rdf:type :Calculator ;
    eep:implements :HealthParameterCalculatorConfig1 .
:HealthParameterCalculatorConfig1
    rdf:type ActivityRecognition:Configuration ;
    :containsHealthParameterConfig :activity_index_config .

# define that calculation of activity index should be done
# using the wearable acceleration property
:activity_index_config rdf:type :HealthParameterConfig ;
```

```
    :forHealthParameter [ rdf:type :ActivityIndex ] ;
    :forProperty [ rdf:type SensorsAndWearables:WearableAcceleration ] .

# define different medical conditions that require movement monitoring
:HeartDisease rdf:type saref4ehaw:ChronicDisease ;
    :requiresMovementMonitoring "true"^^xsd:boolean .
:HighFallRisk rdf:type saref4ehaw:ChronicDisease ;
    :requiresMovementMonitoring "true"^^xsd:boolean .
:Dementia rdf:type saref4ehaw:ChronicDisease ;
    :requiresMovementMonitoring "true"^^xsd:boolean .
```

Listing 15: Overview of the most important triples in the use case context of the described evaluation scenarios

```
# patient lives in HomeLab
patients:patient373 rdf:type saref4ehaw:Patient ;
    MonitoredPerson:livesIn Homelab:homelab .

# patient has an Empatica wearable
patients:patient373 rdf:type saref4wear:Wearer .
HomelabWearable:empatica.E4.A03813
    saref4wear:isLocatedOn patients:patient373 ;
    MonitoredPerson:hasLocation Homelab:homelab .

# patient has a heart disease
patients:patient373 saref4ehaw:hasChronicDisease
    HealthParameterCalculation:HeartDisease .
```

Listing 16: Sensor query rule of the internal representation of the DIVIDE query that is used in the evaluation scenarios to derive an RSP query that continuously measures a patient's activity index. The sensor query rule also includes the template of this RSP query in RSP-QL syntax. During the query derivation, DIVIDE substitutes the input variables and window parameters into this template.

```
{
    ?calculator rdf:type HealthParameterCalculation:Calculator ;
        <https://w3id.org/eep#implements> [
            rdf:type ActivityRecognition:Configuration ;
            HealthParameterCalculation:containsHealthParameterConfig ?hpc ] .
    ?hpc rdf:type HealthParameterCalculation:HealthParameterConfig ;
        HealthParameterCalculation:forHealthParameter [
            rdf:type HealthParameterCalculation:ActivityIndex ] ;
        HealthParameterCalculation:forProperty [ rdf:type ?prop ] .

    ?prop rdfs:subClassOf HealthParameterCalculation:ConditionableProperty .

    ?sensor rdf:type saref-core:Device ;
        saref-core:measuresProperty [
            rdf:type ?prop ;
            SensorsAndWearables:hasAxis [
                rdf:type SensorsAndWearables:XAxis ] ] ;
        Sensors:analyseStateOf ?patient ;
        MonitoredPerson:hasLocation ?home .

    ?patient rdf:type saref4ehaw:Patient ; MonitoredPerson:livesIn ?home .
}
=>
{
    _:q rdf:type sd:Query ; sd:pattern sd-query:pattern ;
        sd:inputVariables (("?s_iri" ?sensor) ("?patient" ?patient)
                           ("?calculator" ?calculator)) ;
        sd:windowParameters (("?range" 80 time:seconds)
                             ("?slide" 10 time:seconds)) .
```

```
    _:p rdf:type HealthParameterCalculation:ActivityIndex ;
        HealthParameterCalculation:calculationMadeFor ?patient ;
        HealthParameterCalculation:calculatedBy ?calculator ;
        saref-core:hasTimestamp _:t ; saref-core:hasValue _:v .
} .

sd-query:prefixes-activity-index rdf:type owl:Ontology ;
    sh:declare [ sh:prefix "xsd" ; sh:namespace "http://www.w3.org/2001/XMLSchema
        #"^^xsd:anyURI ] ;
    sh:declare [ sh:prefix "saref-core" ; sh:namespace "https://saref.etsi.org/core
        /"^^xsd:anyURI ] ;
    sh:declare [ sh:prefix "ActivityRecognition" ; sh:namespace "https://dahcc.
        idlab.ugent.be/Ontology/ActivityRecognition/"^^xsd:anyURI ] ;
    sh:declare [ sh:prefix "HealthParameterCalculation" ; sh:namespace "https://
        dahcc.idlab.ugent.be/Ontology/HealthParameterCalculation/"^^xsd:anyURI ] .

sd-query:pattern-activity-index
    rdf:type sd:QueryPattern ;
    sh:prefixes sd-query:prefixes-activity-index ;
    sh:construct """
        CONSTRUCT {
            _:p a HealthParameterCalculation:RelevantActivityIndex ;
                saref-core:hasValue ?ai ;
                HealthParameterCalculation:calculationMadeFor ?patient ;
                HealthParameterCalculation:calculatedBy ?calculator ;
                saref-core:hasTimestamp ?now .
        }
        FROM NAMED WINDOW :win ON <http://protego.ilabt.imec.be/idlab.homelab> [
            RANGE ?{range} SLIDE ?{slide}]
        WHERE { WINDOW :win {
            BIND (NOW() AS ?now)

            { SELECT ?sensor (AVG(?var) AS ?ai)
              WHERE {
                  SELECT ?sensor ?p
                         (IF(?count=1, -1, (?sx / ?count)) AS ?var)
                  WHERE {
                      SELECT ?sensor ?p (SUM(?x) AS ?sx) ?count
                      WHERE {
                          SELECT ?sensor ?p ?v (xsd:float(?v) AS ?ni) ?mean
                                 (xsd:float(?ni - ?mean) AS ?nmean)
                                 (((?nmean) * (?nmean)) AS ?x) ?count
                          WHERE {
                              ?sensor saref-core:makesMeasurement [
                                  saref-core:hasValue ?v ;
                                  saref-core:relatesToProperty ?p ] .
                              { SELECT ?p (?s_iri AS ?sensor)
                                       (AVG(?v2) AS ?mean)
                                       (COUNT(?v2) as ?count)
                                WHERE {
                                    ?s_iri saref-core:makesMeasurement [
                                        saref-core:hasValue ?v2 ;
                                        saref-core:relatesToProperty ?p ] .
                                }
                                GROUP BY (?s_iri AS ?sensor) ?p }
                          }
                      } GROUP BY ?sensor ?p ?count
                  }
              }
              GROUP BY ?sensor }
        } }
        LIMIT 1
        """ .
```

Listing 17: Goal of the internal representation of the DIVIDE query that is used in the evaluation scenarios to derive an RSP query that continuously measures a patient's activity index. This goal contains the semantic output that should be filtered by the RSP queries derived from this DIVIDE query.

```
{
    ?p rdf:type HealthParameterCalculation:RelevantActivityIndex ;
       saref-core:hasValue ?v ;
       HealthParameterCalculation:calculationMadeFor ?patient ;
       HealthParameterCalculation:calculatedBy ?calculator ;
       saref-core:hasTimestamp ?t .
} => {
    _:p rdf:type HealthParameterCalculation:RelevantActivityIndex ;
        saref-core:hasValue ?v ;
        HealthParameterCalculation:calculationMadeFor ?patient ;
        HealthParameterCalculation:calculatedBy ?calculator ;
        saref-core:hasTimestamp ?t .
} .
```

Appendix D Additional results of the evaluation of DIVIDE

This appendix provides additional insights in the evaluation results of this paper, which are presented in Section 7.

- The results in this paper are aggregated over multiple evaluation runs. However, when running the same scenario multiple times on the DIVIDE Monitoring set-up, different patterns can sometimes be observed. This is the case for both the results in Section 7.1 (Figure 4a) and Section 7.2 (Figures 6a, 7a and 7b). We define a *pattern* as the sequence of RSP query executions combined with the query window parameters for the results in Section 7.1, and the sequence of RSP query executions combined with their query location in Section 7.1. Since the timeline visualizations for the DIVIDE Monitoring set-up are aggregated in time as well, it is impossible to aggregate different result patterns in one figure. Hence, the results in those figures aggregate the runs with the result pattern that occurred most frequently.

  To illustrate this with an example, consider the results for a single evaluation run of the second evaluation scenario on the DIVIDE Monitoring set-up in Figure 11. This run is not aggregated in the results of Figure 6a, since it has a different result pattern: after the first 13 query executions on the Central RSP Engine, it has only *10* query executions on the Local RSP Engine, before the query is moved again to the Central RSP Engine. In contrast, the aggregated evaluation runs in Figure 6a start with 13 query executions on the Central RSP Engine, followed by *12* executions on the Local RSP Engine.
- Figure 12 shows the results for a single evaluation run of the second evaluation scenario on the DIVIDE Central set-up. This scenario updates the RSP query location based on the monitored network RTT. This figure represents one of the runs that are included in the aggregated results of Figure 6c. It demonstrates that the simulated network capacity restrictions can lead to accumulating delays in the processing of the data streams, if the RSP query remains registered to

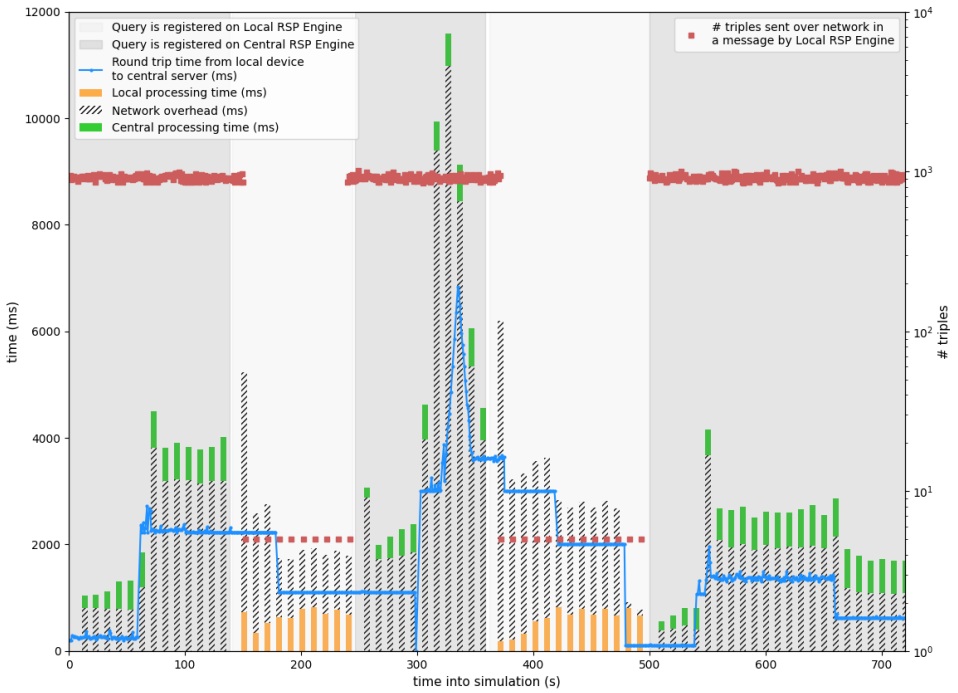


Figure 11: Results of a single evaluation run of evaluation scenario 2 that updates the RSP query location based on the monitored network RTT, for the DIVIDE Monitoring set-up. This run is not included in the aggregated results of Figure 6a, because it has a different result pattern.

the Central RSP Engine. For multiple executions of the RSP query, the network overhead is 25 seconds or higher, up to almost 33 seconds at its highest point.

References

[1] Su, X., Riekki, J., Nurminen, J.K., Nieminen, J., Koskimies, M.: Adding semantics to Internet of Things. Concurrency and Computation: Practice and Experience **27**(8), 1844–1860 (2015) https://doi.org/10.1002/cpe.3203

[2] Aggarwal, C.C., Ashish, N., Sheth, A.: The Internet of Things: A Survey from the Data-Centric Perspective. In: Aggarwal, C.C. (ed.) Managing and Mining Sensor Data, pp. 383–428. Springer, ??? (2013). https://doi.org/10.1007/978-1-4614-6309-2_12

[3] Kalamaras, I., Kaklanis, N., Votis, K., Tzovaras, D.: Towards Big Data Analytics in Large-Scale Federations of Semantically Heterogeneous IoT Platforms. In: Iliadis, L., Maglogiannis, I., Plagianakos, V. (eds.) Artificial Intelligence Applications and Innovations: Proceedings of AIAI 2018 IFIP 12.5 International

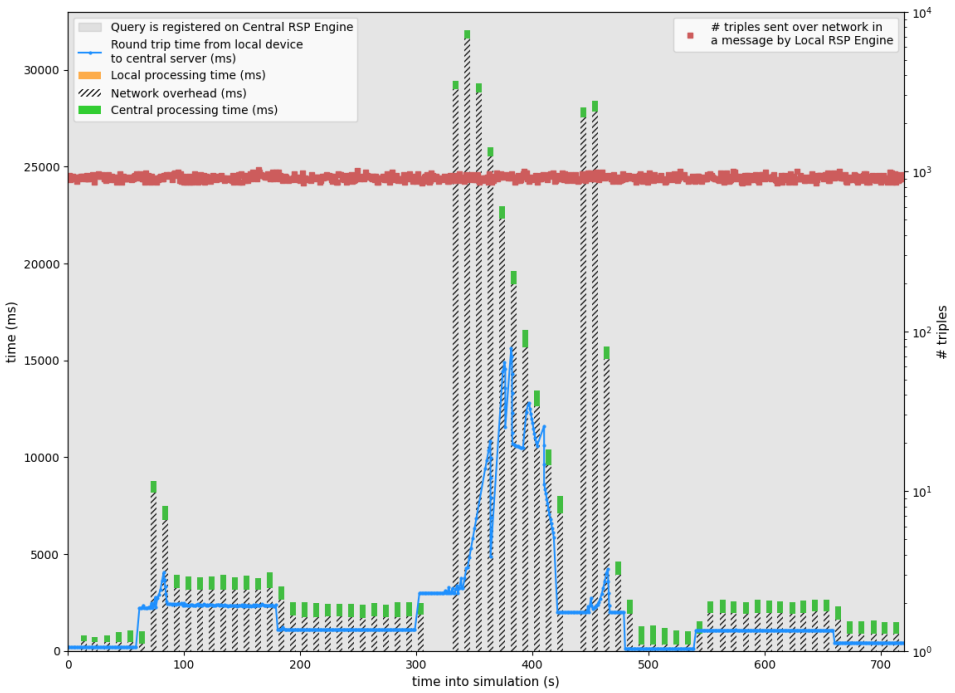


Figure 12: Results of a single evaluation run of evaluation scenario 2 that updates the RSP query location based on the monitored network RTT, for the DIVIDE Central set-up. This run is included in the aggregated results shown in Figure 6c.

Workshops, pp. 13–23. Springer, Cham, Switzerland (2018). https://doi.org/10.1007/978-3-319-92016-0_2

[4] Barnaghi, P., Wang, W., Henson, C., Taylor, K.: Semantics for the Internet of Things: Early Progress and Back to the Future. International Journal on Semantic Web and Information Systems (IJSWIS) **8**(1), 1–21 (2012) https://doi.org/10.4018/jswis.2012010101

[5] Dell'Aglio, D., Della Valle, E., Harmelen, F., Bernstein, A.: Stream reasoning: A survey and outlook. Data Science **1**(1-2), 59–83 (2017) https://doi.org/10.3233/DS-170006

[6] De Brouwer, M., Steenwinckel, B., Fang, Z., Stojchevska, M., Bonte, P., De Turck, F., Van Hoecke, S., Ongenae, F.: Context-aware query derivation for IoT data streams with DIVIDE enabling privacy by design. Semantic Web **14**(5), 893–941 (2023) https://doi.org/10.3233/SW-223281

[7] De Brouwer, M., Arndt, D., Bonte, P., De Turck, F., Ongenae, F.: DIVIDE:

Adaptive Context-Aware Query Derivation for IoT Data Streams. In: Joint Proceedings of the International Workshops on Sensors and Actuators on the Web, and Semantic Statistics, Co-located with the 18th International Semantic Web Conference (ISWC 2019), vol. 2549, pp. 1–16. CEUR Workshop Proceedings, Aachen (2019). https://ceur-ws.org/Vol-2549/article-01.pdf

[8] Cyganiak, R., Wood, D., Lanthaler, M., Klyne, G., Carroll, J.J., McBride, B.: RDF 1.1 concepts and abstract syntax. W3C Recommendation, World Wide Web Consortium (W3C) (2014). https://www.w3.org/TR/rdf11-concepts/

[9] W3C OWL Working Group: OWL 2 Web Ontology Language. W3C Recommendation, World Wide Web Consortium (W3C) (2012). https://www.w3.org/TR/owl2-overview/

[10] Harris, S., Seaborne, A.: SPARQL 1.1 Query Language. W3C Recommendation, World Wide Web Consortium (W3C) (2013). https://www.w3.org/TR/sparql11-query/

[11] Motik, B., Grau, B.C., Horrocks, I., Wu, Z., Fokoue, A., Lutz, C., et al.: OWL 2 Web Ontology Language Profiles (Second Edition). W3C Recommendation, World Wide Web Consortium (W3C) (2012). https://www.w3.org/TR/owl2-profiles/

[12] Su, X., Gilman, E., Wetz, P., Riekki, J., Zuo, Y., Leppänen, T.: Stream reasoning for the Internet of Things: Challenges and gap analysis. In: Proceedings of the 6th International Conference on Web Intelligence, Mining and Semantics (WIMS 2016), pp. 1–10. Association for Computing Machinery (ACM), New York, NY, USA (2016). https://doi.org/10.1145/2912845.2912853

[13] Barbieri, D.F., Braga, D., Ceri, S., Della Valle, E., Grossniklaus, M.: C-SPARQL: a continuous query language for RDF data streams. International Journal of Semantic Computing **4**(1), 3–25 (2010) https://doi.org/10.1142/S1793351X10000936

[14] Dell'Aglio, D., Della Valle, E., Calbimonte, J.-P., Corcho, O.: RSP-QL Semantics: A Unifying Query Model to Explain Heterogeneity of RDF Stream Processing Systems. International Journal on Semantic Web and Information Systems (IJSWIS) **10**(4), 17–44 (2014)

[15] Stuckenschmidt, H., Ceri, S., Della Valle, E., Van Harmelen, F.: Towards expressive stream reasoning. In: Semantic Challenges in Sensor Networks, Dagstuhl Seminar Proceedings, Dagstuhl, Germany (2010). https://doi.org/10.4230/DagSemProc.10042.4 . Schloss Dagstuhl-Leibniz-Zentrum für Informatik

[16] De Brouwer, M., Ongenae, F., Bonte, P., De Turck, F.: Towards a Cascading Reasoning Framework to Support Responsive Ambient-Intelligent Healthcare Interventions. Sensors **18**(10), 3514 (2018) https://doi.org/10.3390/s18103514

[17] Bonte, P., Tommasini, R., Della Valle, E., De Turck, F., Ongenae, F.: Streaming MASSIF: cascading reasoning for efficient processing of iot data streams. Sensors **18**(11), 3832 (2018) https://doi.org/10.3390/s18113832

[18] Berners-Lee, T., Connolly, D., Kagal, L., Scharf, Y., Hendler, J.: N3Logic: A logical framework for the World Wide Web. Theory and Practice of Logic Programming **8**(3), 249–269 (2008) https://doi.org/10.1017/S1471068407003213

[19] Daniele, L., Hartog, F., Roes, J.: Created in Close Interaction with the Industry: The Smart Appliances REFerence (SAREF) Ontology. In: Formal Ontologies Meet Industry, pp. 100–112. Springer, Cham, Switzerland (2015). https://doi.org/10.1007/978-3-319-21545-7_9

[20] Rijgersberg, H., Van Assem, M., Top, J.: Ontology of units of measure and related concepts. Semantic Web **4**(1), 3–13 (2013) https://doi.org/10.3233/SW-2012-0069

[21] Rodola, G.: psutil. Accessed: 2023-03-29 (2022). https://github.com/giampaolo/psutil

[22] RSP Service Interface for C-SPARQL. Accessed: 2018-10-18. https://github.com/streamreasoning/rsp-services-csparql/

[23] The Apache Software Foundation: Apache Jena. Accessed: 2022-02-01 (2021). https://jena.apache.org/

[24] Hitzler, P., Krotzsch, M., Rudolph, S.: Foundations of Semantic Web Technologies. CRC press, ??? (2009)

[25] Francesconi, E., Montemagni, S., Peters, W., Tiscornia, D.: Integrating a Bottom–Up and Top–Down Methodology for Building Semantic Resources for the Multilingual Legal Domain, pp. 95–121. Springer, ??? (2010). https://doi.org/10.1007/978-3-642-12837-0_6

[26] Vet, P.E., Mars, N.J.I.: Bottom-up construction of ontologies. IEEE Transactions on Knowledge and Data Engineering **10**(4), 513–526 (1998) https://doi.org/10.1109/69.706054

[27] Atanasova, T.: Methods for Processing of Heterogeneous Data in IoT Based Systems. In: DCCN 2019: Distributed Computer and Communication Networks, pp. 524–535. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-36625-4_42

[28] Antunes, M., Gomes, D., Aguiar, R.: Semantic-Based Publish/Subscribe for M2M. In: 2014 International Conference on Cyber-Enabled Distributed Computing and Knowledge Discovery, pp. 256–263 (2014). https://doi.org/10.1109/CyberC.2014.53

[29] Corral-Plaza, D., Medina-Bulo, I., Ortiz, G., Boubeta-Puig, J.: A stream processing architecture for heterogeneous data sources in the Internet of Things. Computer Standards & Interfaces **70** (2020) https://doi.org/10.1016/j.csi.2020.103426

[30] Cimmino, A., Oravec, V., Serena, F., Kostelnik, P., Poveda-Villalón, M., Tryferidis, A., García-Castro, R., Vanya, S., Tzovaras, D., Grimm, C.: VICINITY: IoT semantic interoperability based on the web of things. In: 15th International Conference on Distributed Computing in Sensor Systems (DCOSS), pp. 241–247. IEEE, New York, NY, USA (2019). https://doi.org/10.1109/DCOSS.2019.00061

[31] Ganzha, M., Paprzycki, M., Pawłowski, W., Szmeja, P., Wasielewska, K.: Semantic interoperability in the Internet of Things: An overview from the INTER-IoT perspective. Journal of Network and Computer Applications **81**, 111–124 (2017) https://doi.org/10.1016/j.jnca.2016.08.007

[32] Bröring, A., Schmid, S., Schindhelm, C.-K., Khelil, A., Käbisch, S., Kramer, D., Le Phuoc, D., Mitic, J., Anicic, D., Teniente, E.: Enabling IoT ecosystems through platform interoperability. IEEE Software **34**(1), 54–61 (2017) https://doi.org/10.1109/MS.2017.2

[33] Soursos, S., Žarko, I.P., Zwickl, P., Gojmerac, I., Bianchi, G., Carrozzo, G.: Towards the cross-domain interoperability of IoT platforms. In: 2016 European Conference on Networks and Communications (EuCNC), pp. 398–402. IEEE, New York, NY, USA (2016). https://doi.org/10.1109/EuCNC.2016.7561070

[34] Sofia2: Sofia2 – Technology for Innovators. Accessed: 2022-03-10 (2020). https://sofia2.com

[35] Javed, A., Kubler, S., Malhi, A., Nurminen, A., Robert, J., Främling, K.: bIoTope: Building an IoT Open Innovation Ecosystem for Smart Cities. IEEE Access **8**, 224318–224342 (2020) https://doi.org/10.1109/ACCESS.2020.3041326

[36] Cirillo, F., Solmaz, G., Berz, E.L., Bauer, M., Cheng, B., Kovacs, E.: A standard-based open source IoT platform: FIWARE. IEEE Internet of Things Magazine **2**(3), 12–18 (2019) https://doi.org/10.1109/IOTM.0001.1800022

[37] Zgheib, R., Kristiansen, S., Conchon, E., Plageman, T., Goebel, V., Bastide, R.: A scalable semantic framework for IoT healthcare applications. Journal of Ambient Intelligence and Humanized Computing (2020) https://doi.org/10.1007/s12652-020-02136-2

[38] Jabbar, S., Ullah, F., Khalid, S., Khan, M., Han, K.: Semantic interoperability in heterogeneous IoT infrastructure for healthcare. Wireless Communications and Mobile Computing **2017** (2017) https://doi.org/10.1155/2017/9731806

[39] Ullah, F., Habib, M.A., Farhan, M., Khalid, S., Durrani, M.Y., Jabbar, S.:

Semantic interoperability for big-data in heterogeneous IoT infrastructure for healthcare. Sustainable Cities and Society **34**, 90–96 (2017) https://doi.org/10.1016/j.scs.2017.06.010

[40] Jaiswal, K., Anand, V.: A Survey on IoT-Based Healthcare System: Potential Applications, Issues, and Challenges. In: Rizvanov, A.A., Singh, B.K., Ganasala, P. (eds.) Advances in Biomedical Engineering and Technology, pp. 459–471. Springer, ??? (2021). https://doi.org/10.1007/978-981-15-6329-4_38

[41] Balakrishna, S., Solanki, V.K., Gunjan, V.K., Thirumaran, M.: A Survey on Semantic Approaches for IoT Data Integration in Smart Cities. In: ICICCT 2019 – System Reliability, Quality Control, Safety, Maintenance and Management, pp. 827–835. Springer, Singapore (2020). https://doi.org/10.1007/978-981-13-8461-5_94

[42] Chamoso, P., González-Briones, A., De La Prieta, F., Venyagamoorthy, G.K., Corchado, J.M.: Smart city as a distributed platform: Toward a system for citizen-oriented management. Computer Communications **152**, 323–332 (2020) https://doi.org/10.1016/j.comcom.2020.01.059

[43] D'Aniello, G., Gaeta, M., Orciuoli, F.: An approach based on semantic stream reasoning to support decision processes in smart cities. Telematics and Informatics **35**(1), 68–81 (2018) https://doi.org/10.1016/j.tele.2017.09.019

[44] Jara, A.J., Serrano, M., Gómez, A., Fernández, D., Molina, G., Bocchi, Y., Alcarria, R.: Smart Cities Semantics and Data Models. In: Proceedings of the International Conference on Information Technology & Systems (ICITS 2018), pp. 77–85. Springer, ??? (2018). https://doi.org/10.1007/978-3-319-73450-7_8

[45] Vergara, J.E., Guerrero, A., Villagrá, V.A., Berrocal, J.: Ontology-based network management: study cases and lessons learned. Journal of Network and Systems Management **17**(3), 234–254 (2009) https://doi.org/10.1007/s10922-009-9129-1

[46] Moraes, P.S., Sampaio, L.N., Monteiro, J.A.S., Portnoi, M.: MonONTO: A Domain Ontology for Network Monitoring and Recommendation for Advanced Internet Applications Users. In: NOMS Workshops 2008 – IEEE Network Operations and Management Symposium Workshops, pp. 116–123. IEEE, New York, NY, USA (2008). https://doi.org/10.1109/NOMSW.2007.21

[47] Salvador, A., Vergara Méndez, J., Tropea, G., Blefari-Melazzi, G., Ferreiro, A.: Ontology design and implementation for IP networks monitoring. In: International Workshop on Web & Semantic Technology (WeST-2009) (2009)

[48] Adianto, W., Laat, C., Grosso, P.: Future Internet Ontologies: The NOVI Experience. Preprint submitted to Semantic Web Journal (2009)

[49] Silva, R.F., Carvalho, P., Rito Lima, S., Álvarez Sabucedo, L., Santos Gago, J.M.,

Silva, J.M.C.: An Ontology-Based Recommendation System for Context-Aware Network Monitoring. In: Rocha, Á., Adeli, H., Reis, L.P., Costanzo, S. (eds.) New Knowledge in Information Systems and Technologies, pp. 373–384. Springer, Cham, Switzerland (2019). https://doi.org/10.1007/978-3-030-16184-2_36

[50] Krinkin, K., Vodyaho, A., Kulikov, I., Zhukova, N.: Models of Telecommunications Network Monitoring Based on Knowledge Graphs. In: 9th Mediterranean Conference on Embedded Computing (MECO 2020), pp. 1–7 (2020). https://doi.org/10.1109/MECO49872.2020.9134148

[51] Kulikov, I., Vodyaho, A., Stankova, E., Zhukova, N.: Ontology for Knowledge Graphs of Telecommunication Network Monitoring Systems. In: Computational Science and Its Applications – ICCSA 2021, pp. 432–446. Springer, Cham, Switzerland (2021). https://doi.org/10.1007/978-3-030-87010-2_32

[52] Fallon, L., Keeney, J., O'Sullivan, D.: Applying Semantics to Optimize End-User Services in Telecommunication Networks. In: Meersman, R., Panetto, H., Dillon, T., Missikoff, M., Liu, L., Pastor, O., Cuzzocrea, A., Sellis, T. (eds.) On the Move to Meaningful Internet Systems: OTM 2014 Conferences, pp. 718–726. Springer, Berlin, Heidelberg (2014). https://doi.org/10.1007/978-3-662-45563-0_44

[53] Funika, W., Janczykowski, M., Jopek, K., Grzegorczyk, M.: An Ontology-based Approach to Performance Monitoring of MUSCLE-bound Multi-scale Applications. Procedia Computer Science **18**, 1126–1135 (2013) https://doi.org/10.1016/j.procs.2013.05.278 . 2013 International Conference on Computational Science

[54] Ryabinin, K., Chuprina, S.: Ontology-Driven Edge Computing. In: Computational Science – ICCS 2020, pp. 312–325. Springer, Cham, Switzerland (2020). https://doi.org/10.1007/978-3-030-50436-6_23

[55] Tayur, V.M., Suchithra, R.: A Comprehensive Ontology for Internet of Things (COIoT). In: 2019 Second International Conference on Advanced Computational and Communication Paradigms (ICACCP), pp. 1–6. IEEE, New York, NY, USA (2019). https://doi.org/10.1109/ICACCP.2019.8882936

[56] Strassner, J.: DEN-ng: achieving business-driven network management. In: NOMS 2002. IEEE/IFIP Network Operations and Management Symposium.'Management Solutions for the New Communications World'(Cat. No. 02CH37327), pp. 753–766. IEEE, New York, NY, USA (2002). https://doi.org/10.1109/NOMS.2002.1015622

[57] Famaey, J., Latré, S., Strassner, J., De Turck, F.: An Ontology-Driven Semantic Bus for Autonomic Communication Elements. In: Modelling Autonomic Communication Environments, pp. 37–50. Springer, Berlin, Heidelberg (2010). https://doi.org/10.1007/978-3-642-16836-9_4

[58] Latré, S., Famaey, J., Strassner, J., De Turck, F.: Automated context dissemination for autonomic collaborative networks through semantic subscription filter generation. Journal of Network and Computer Applications **36**(6), 1405–1417 (2013) https://doi.org/10.1016/j.jnca.2013.01.011

[59] Hwaitat, A.K.A., Shaheen, A., Adhim, K., Arkebat, E.N., Hwiatat, A.A.A.: Computer Hardware Components Ontology. Modern Applied Science **12**(3), 35–40 (2018) https://doi.org/10.5539/mas.v12n3p35

[60] Keeney, J., Fallon, L., Tai, W., O'Sullivan, D.: Towards composite semantic reasoning for realtime network management data enrichment. In: 11th International Conference on Network and Service Management (CNSM 2015), pp. 246–250 (2015). https://doi.org/10.1109/CNSM.2015.7367365

[61] Fallon, L., O'Sullivan, D.: The Aesop Approach for Semantic-Based End-User Service Optimization. IEEE Transactions on Network and Service Management **11**(2), 220–234 (2014) https://doi.org/10.1109/TNSM.2014.2321784

[62] Keeney, J., Lewis, D., O'Sullivan, D.: Ontological semantics for distributing contextual knowledge in highly distributed autonomic systems. Journal of Network and Systems Management **15**(1), 75–86 (2007) https://doi.org/10.1007/s10922-006-9054-5

[63] Verbelen, T., Simoens, P., De Turck, F., Dhoedt, B.: AIOLOS: Middleware for improving mobile application performance through cyber foraging. Journal of Systems and Software **85**(11), 2629–2639 (2012) https://doi.org/10.1016/j.jss.2012.06.011

[64] Sebrechts, M., Volckaert, B., De Turck, F., Yangy, K., AL-Naday, M.: Fog Native Architecture: Intent-Based Workflows to Take Cloud Native Towards the Edge. IEEE Communications Magazine **60**(8), 44–50 (2022) https://doi.org/10.1109/MCOM.003.2101075

[65] Santos, J., Wauters, T., Volckaert, B., De Turck, F.: Towards Network-Aware Resource Provisioning in Kubernetes for Fog Computing Applications. In: 2019 IEEE Conference on Network Softwarization (NetSoft), pp. 351–359 (2019). https://doi.org/10.1109/NETSOFT.2019.8806671

[66] Idrees, A.K., Al-Qurabat, A.K.M.: Energy-efficient data transmission and aggregation protocol in periodic sensor networks based fog computing. Journal of Network and Systems Management **29**(1) (2021) https://doi.org/10.1007/s10922-020-09567-4

[67] Verma, S., Bala, A.: Auto-scaling techniques for IoT-based cloud applications: a review. Cluster Computing **24**(3), 2425–2459 (2021) https://doi.org/10.1007/s10586-021-03265-9

[68] Tran-Dang, H., Bhardwaj, S., Rahim, T., Musaddiq, A., Kim, D.-S.: Reinforcement learning based resource management for fog computing environment: Literature review, challenges, and open issues. Journal of Communications and Networks **24**(1), 83–98 (2022) https://doi.org/10.23919/JCN.2021.000041

[69] Bai, J., Di, C., Xiao, L., Evenson, K.R., LaCroix, A.Z., Crainiceanu, C.M., Buchner, D.M.: An activity index for raw accelerometry data and its comparison with other activity metrics. PLoS ONE **11**(8), 0160644 (2016) https://doi.org/10.1371/journal.pone.0160644

[70] Steenwinckel, B., De Brouwer, M., Stojchevska, M., Van Der Donckt, J., Nelis, J., Ruyssinck, J., Herten, J., Casier, K., Van Ooteghem, J., Crombez, P., De Turck, F., Van Hoecke, S., Ongenae, F.: Data Analytics For Health and Connected Care: Ontology, Knowledge Graph and Applications. In: Proceedings of the 16th EAI International Conference on Pervasive Computing Technologies for Healthcare (EAI PervasiveHealth 2022) (2022). https://dahcc.idlab.ugent.be

[71] Girod-Genet, M., Ismail, L.N., Lefrançois, M., Moreira, J.: ETSI TS 103 410-8 V1.1.1 (2020-07): SmartM2M; Extension to SAREF; Part 8: eHealth/Ageing-well Domain. Technical report, ETSI Technical Committee Smart Machine-to-Machine communications (SmartM2M) (2020). https://www.etsi.org/deliver/etsi_ts/103400_103499/10341008/01.01.01_60/ts_10341008v010101p.pdf

[72] Esnaola-Gonzalez, I., Bermúdez, J., Fernández, I., Arnaiz, A.: Two Ontology Design Patterns toward Energy Efficiency in Buildings. In: Proceedings of the 9th Workshop on Ontology Design and Patterns (WOP 2018), Co-located with 17th International Semantic Web Conference (ISWC 2018), pp. 14–28. CEUR Workshop Proceedings, ??? (2018). https://ceur-ws.org/Vol-2195/pattern_paper_2.pdf

[73] Empatica: E4 wristband. Accessed: 2020-10-23 (2020). https://www.empatica.com/research/e4

[74] Ghent University – imec: iLab.t Virtual Wall. Accessed: 2023-04-09 (2023). https://doc.ilabt.imec.be/ilabt/virtualwall