



UNIVERSITÀ DEGLI STUDI DI MILANO

UNIVERSITÀ DEGLI STUDI DI MILANO

PHD SCHOOL ON COMPUTER SCIENCE

DEPARTMENT OF COMPUTER SCIENCE "GIOVANNI DEGLI ANTONI"

PHD IN COMPUTER SCIENCE - XXX CYCLE

DOCTORAL THESIS

Ontology-based Consistent Specification and Scalable Execution of Sensor Data Acquisition Plans in Cross-Domain IoT Platforms

INF/01

Author:

Luca FERRARI

Advisor:

Prof. Marco MESITI

Co-Advisor:

Prof. Stefano VALTOLINA

School Director: Prof. Paolo BOLDI

Academic Year 2016/17

UNIVERSITÀ DEGLI STUDI DI MILANO

Abstract

Computer Science

Doctor of Philosophy

Ontology-based Consistent Specification and Scalable Execution of Sensor Data Acquisition Plans in Cross-Domain IoT Platforms

by Luca FERRARI

Nowadays there is an increased number of *vertical* Internet of Things (IoT) applications that have been developed within IoT Platforms that often do not interact with each other because of the adoption of different standards and formats. Several efforts are devoted to the construction of software infrastructures that facilitate the interoperability among heterogeneous cross-domain IoT platforms for the realization of *horizontal* applications. Even if their realization poses different challenges across all layers of the network stack, in this thesis we focus on the interoperability issues that arise at the data management layer. Starting from a flexible multi-granular Spatio-Temporal-Thematic data model according to which events generated by different kinds of sensors can be represented, we propose a Semantic Virtualization approach according to which the sensors belonging to different IoT platforms and the schema of the produced event streams are described in a *Domain Ontology*, obtained through the extension of the well-known ontologies (SSN and IoT-Lite ontologies) to the needs of a specific domain. Then, these sensors can be exploited for the creation of Data Acquisition Plans (DAPs) by means of which the streams of events can be filtered, merged, and aggregated in a meaningful way. Notions of soundness and consistency are introduced to bind the output streams of the services contained in the DAP with the Domain Ontology for providing a semantic description of its final output. The facilities of the StreamLoader prototype are finally presented for supporting the domain experts in the Semantic Virtualization of the sensors and for the construction of meaningful DAPs. Different graphical facilities have been developed for supporting domain experts in the development of complex DAPs. The system provides also facilities for their syntax-based translations in the Apache Spark Streaming language and execution in real time in a distributed cluster of machines.

Acknowledgements

I would really want to thank my supervisor Marco. In the last three years we encountered considerable difficulties, but thanks to him, to his perseverance, we were able to complete this course of study. We spent also a lot of funny moments and I think that now I am not only a Doctor in Philosophy, but I am a better person. I'd like to thank my co-supervisor Stefano that continuously supported me and stimulated me in order to achieve the target goals.

I need to thank Professor Bruno Apolloni and Professor Simone Bassis that convinced me to make the decision to start a doctorate and helped me in the first phases of my PhD.

During the last three years I worked and I collaborated with a lot of researchers and I have to thank them all. In particular Paolo Perlasca and Barbara Rita Barricelli, to whom I have continuously asked for help and advices and have always listened to me and always helped me. I need to thank also Professor Koji Zettsu for the opportunity he gave me to spend 3 months at the National Institute of Information and Communications Technology (NICT) in Kyoto and Doctor Minh-Son Dao and Doctor Sulayman K. Sowe for the help and support they gave me during that period.

I have to thank Doctor Luisa Ferrario and Doctor Anna Paola Bocciarelli who gave me the opportunity to finish my PhD when I started my work at the Divisione Sistemi Informativi and all the DivSI colleagues.

I can not forget to thank Lara for the huge amount of time we spent at the phone during my "crises". She always convinced me that giving up was not the right choice.

Another important person to thank is Matteo. In the last few years I spent more time with him (and Davide playing together for the group *La Melissa*) than with my family and he always supported me and tolerated all my strange and, sometimes, stupid ideas.

Finally I have to thank Serena. The most amazing person I have met so far and thanks to her, to the moments we spent together, has made these last months happier and carefree.

Contents

Declaration of Authorship	iii
Abstract	v
Acknowledgements	vii
Introduction	1
1 IoT Platforms. Current Features and Future Trends	13
1.1 IoT Architecture Building Blocks	15
1.1.1 The Device Layer	15
1.1.2 The Gateway Layer	17
1.1.3 The Integration Layer	18
1.1.4 The Application Layer	20
1.2 IoT Platforms	24
1.2.1 Vertical IoT Platforms	24
1.2.2 Cross-Domain IoT Platforms	31
1.3 Comparison of IoT Platforms	36
1.4 Concluding Remarks	38
2 Semantic Interoperability in the IoT Context	41
2.1 The Issues of Interoperability	42
2.1.1 IoT Interoperability	43
2.1.2 Semantic Interoperability	45
2.2 Ontologies for IoT	46
2.2.1 Km4City	47
2.2.2 Stream Annotation Ontology (SAO)	50
2.2.3 W3C Semantic Sensor Network Ontology (SSN)	51
2.2.4 IoT-Lite	53
2.3 Semantic Description Techniques	55
2.4 Mapping Techniques	57
3 Big IoT Data Processing	61
3.1 Characteristics of Big Data	62
3.2 Communication Protocols	64
3.2.1 Request-Reply Interaction	64

3.2.2	Push-Based Data Propagation	65
3.2.3	Publish-Subscribe Interaction Model	66
3.3	Batch Processing Systems	69
3.4	Stream Processing Systems	71
3.5	Hybrid Processing Systems	74
3.6	Comparison of IoT Streaming Systems	81
4	Syntactic Data Model and Domain Ontology	85
4.1	The STT Syntactic Data Model	86
4.1.1	Spatial and Temporal Granularities and Thematic Dimensions	86
4.1.2	Temporal and Spatial Types and Values	87
4.1.3	STT Events and Stream Data Model	88
4.2	The Domain Ontology	90
4.2.1	Spatial Dimension	92
4.2.2	Temporal Dimension	94
4.2.3	Thematic Dimension	95
4.3	Concluding Remarks	97
5	Semantic Virtualization of Sensors	99
5.1	Semantic Discovery of Sensors	100
5.1.1	Dealing with Different Formats	102
5.1.2	The Sensor Discovery Algorithm	105
5.1.3	Semantic Labeling	106
5.2	Evaluation of Sensor Consistency	109
5.2.1	Semantic Characterization of Sensors	110
5.2.2	Consistency of Sensor Schema w.r.t. the Domain Ontology	112
5.3	Automatic Transformation of Sensor Events	114
6	Sound and Consistent Data Acquisition Plans	117
6.1	Data Acquisition Services	118
6.1.1	Non-Blocking Services	118
6.1.2	Blocking Services	120
6.2	Data Acquisition Plan	122
6.3	Sound/Consistent Specification of Data Acquisition Plan	126
6.4	Verification of Consistency in a Data Acquisition Plans	126
6.4.1	Auxiliary Sensors in the Domain Ontology	127
6.4.2	A Consistent Data Acquisition Plan	130
7	The StreamLoader Prototype	133
7.1	The Overall StreamLoader Environment	134
7.2	Semantic Virtualization Graphical Specification	135
7.3	Data Acquisition Plan Graphical Specification	140
7.4	Data Acquisition Plan Translation	147

7.4.1	JSON Representation of a DAP	149
7.4.2	Configuration	153
7.4.3	Translation of Sources and Destination	154
7.4.4	Translation of Services	156
7.4.5	The Overall Translation Algorithm	163
8	Experimental Evaluation	167
8.1	Environment and Tests Configuration	167
8.1.1	Local and Cluster Configuration	168
8.1.2	Data Acquisition Plan Configuration and Metrics	169
8.2	Data Acquisition Plan Execution Experiments	170
8.2.1	Results	171
8.3	Further Data Acquisition Plan Execution Experiments	172
8.3.1	Results	173
8.4	Semantic Virtualization Experiments	174
8.4.1	Type of Experiments	176
8.4.2	Results	178
8.5	Concluding Remarks	182
	Conclusion	183
	Bibliography	187

List of Figures

1	The LHD factor, the zones of the city of Milano, and the thresholds for HD	4
2	Formats of the events generated by the sensors in the zones of Milano	5
3	StreamLoader General Architecture	7
1.1	Building blocks of an IoT architecture	16
1.2	Block diagram of a smart home system	21
1.3	Block diagram of a smart city system	22
2.1	Different levels of interoperability	43
2.2	"Knowledge Hierarchy" in the context of IoT	44
2.3	Ontology Macro-Classes and their connections	48
2.4	Overview of the SAO Ontology modules	50
2.5	Overview of the Semantic Sensor Network ontology modules	52
2.6	Overview of the Semantic Sensor Network ontology classes and properties	53
2.7	Dependencies of the SOSA ontology	54
2.8	IoT-Lite concepts and the main relationships between them	55
2.9	The three possible ways for using ontologies for content explication	56
2.10	Classification of schema matching approaches	58
3.1	Anatomy of a topic	69
3.2	Structure of a Storm topology	72
3.3	The iterative operations on Spark RDD	75
3.4	The Apache Spark Ecosystem	76
3.5	Spark Streaming working flow	76
3.6	The Apache Flink Ecosystem	78
3.7	The Apache NiFi GUI	80
3.8	The Apache NiFi Architecture	81
4.1	Graphical representation of temporal granularity	88
4.2	Graphical representation of spatial granularity	89
4.3	Relationships between the ontologies that compose our Domain Ontology	92
4.4	Twitter sensor of type T	93
4.5	Temporal dimension of sensor of type T	95

4.6	Thematic dimension of sensor of type T	96
4.7	Thematic dimension of sensor of type TW	97
5.1	Formats of the events generated by the sensors in the zones of Milano	103
6.1	BNF predicate of basic conditions	118
6.2	Graph representation of the Data Acquisition Plan of our running example	124
6.3	Evaluation of consistency of the running example's Data Acquisition Plan	130
7.1	The overall architecture	134
7.2	Specification of the STT granularities	136
7.3	Spatial Labeling	137
7.4	Temporal Labeling	138
7.5	Thematic Labeling	139
7.6	Main screen of the Web Application	140
7.7	Identification of an error during the DAP creation	142
7.8	Source tab menu	142
7.9	Filter tab menu	143
7.10	Enrich tab menu	144
7.11	Virtual Property tab menu	144
7.12	Transform tab menu	145
7.13	Aggregation tab menu	145
7.14	Union tab menu	146
7.15	Join tab menu	146
7.16	Trigger Event tab menu	147
7.17	Convert Temporal and Spatial menus	148
7.18	Data Acquisition Plan for the running example realized with the provided GUI	148
7.19	Excerpt of the DAP to be translated	149
8.1	Local execution and cluster execution environments	168
8.2	Batch information provided by the Spark UI interface	169
8.3	Composition of blocking and non-blocking services	171
8.4	Composition of 10 aggregate services in cascade	171
8.5	Input rate, Processing time, Scheduling delay and Total Delay of the blocking and non-blocking experiment	172
8.6	Input rate, Processing time, Scheduling delay and Total Delay of the 10 aggregation experiment	173
8.7	Humidex Factor calculation DAP	173
8.8	Processing time and batch size of different DAPs	175
8.9	Architectures for the application of the transformation rules	176
8.10	Type of experiments	177

8.11 Average generated tuples per second for the three experiments	179
8.12 Processing Times of the considered Data Acquisition Plans	180
8.13 Scheduling Delay of the considered Data Acquisition Plans	181
8.14 Total Delay for every experiment and for the different approaches . . .	182

List of Tables

1.1	Examples of IoT Platforms	25
1.2	Main IoT Platforms	26
1.3	Measures of Interoperability of IoT Platforms	32
1.4	Comparison of IoT Platforms	37
1.5	Comparison of IoT Platforms	39
3.1	Comparison of IoT application layer messaging protocols	67
3.2	Comparison of NiFi and FDP components and their description	80
3.3	Comparison of IoT Processing Frameworks	83
5.1	Notations	105
5.2	Primitives for the modification of ontology instances	109
6.1	Formal Sensors, Services, Destination nodes and edges representation	123
6.2	Primitives for the modification of Ontology instances	128
6.3	Instructions for modifying the Ontology Instances according to the service	129
7.1	Graphical representation of source, services and destination	141

*Dedicated to Davide, because without his help I would never
have managed to reach this goal.*

Introduction

According to Gartner Inc.¹ in 2020 more than 20.8 billion connected things will be in use worldwide. These devices can be either physical or social sensors which produce large, complex, heterogeneous, structured or unstructured data that can be profitably used for generating a wide plethora of services. Dealing with the vast amount of data produced by the things, their varying capabilities, and an exploding number of services that can be realized, are around the biggest conceptual and technological challenges of our time [54]. This challenge is further exaggerated by the typical ills of early-stage technology that lead to the production of different solutions and standard formats (e.g. oneM2M [3], OMA NGSI 9/10 [105], ETSI M2M [40]) in different context of use that rarely are able to collaborate each other. This has led to the creation of more than 600 IoT platforms (according to a recent survey [5]).

By exploiting these platforms, many "vertical" applications can be developed in different fields (e.g. environment monitoring [106], healthcare service [18], transportation and logistics [69], Smart Cities [113], Smart Homes [73]). However, these solutions are characterized by the use of hardware and software of a specific industry and the interoperability with other applications is rarely supported. This means that if we have an application for monitoring the energy consumption in a Smart Home platform and one for environment monitoring in a Smart City platform there is no easy way to combine them in the scope of a new added-value application, like for the example determining the levels of the internal heater relying on the weather forecast for the next hours. This lacks of interoperability prevents the emergence of broadly accepted IoT ecosystems [24]. A recent McKinsey study [93] estimates that a 40% share of the potential economic value of the IoT directly depends on interoperability gaps among IoT platforms.

As a result, the development of "horizontal" applications that exploit sensors, actuators and infrastructures made available by different platforms is complex with negative effects on the development of vibrant IoT ecosystems. These technological barriers also have negative impact on the business opportunities, especially for small innovative enterprises, which cannot afford to offer their solutions across multiple platforms. This lack of interoperability results in the lost of business opportunities and prevents innovative business ideas.

¹<https://www.gartner.com/newsroom/id/3165317> (2016)

To overcome these limitations, a number of European Projects are currently under development with the purpose of generating infrastructures among heterogeneous platforms that facilitate the development of "horizontal" applications. Even if the realization of these horizontal applications poses different challenges across all layers of the network stack, we wish to focus on those that arise at the data management layer. First, data can be represented in different formats (e.g. CSV, XML, JSON) and present different structures for representing the same kind of information. Second, data can be represented at different spatial and/or temporal granularities (e.g. temperatures per hour in a room versus temperatures per day in a geographical area), and according to different thematics (data about traffic jams vs data about pollutions). Moreover, as remarked in [16, 77], data can be incomplete and need to be enhanced by considering contextual information that can be acquired by knowledge bases and other information sources. Last but not least, data might have diverse semantics, including units of measurement (temperatures in Celsius or Fahrenheit degrees), accuracy, mathematical constructs, sensor types and properties and more.

All these issues, that are common in any situation in which we wish to integrate heterogeneous information systems, are further exacerbated from the variability and frequency with which sensors appear and disappear from the network, and the lack of a standard ontology for the description of the sensor data that is widely recognized. Only few approaches (like OpenIoT [136] and an extension of Orion Context Broker [155]) exploit ontologies (like the *Semantic Sensor Network* ontology [126] and the *IoT-Lite* ontology [15]) for characterizing the semantics of the data produced by sensors. However, they can guarantee on the semantics of the events generated by sensors but no support is provided for the characterization of the services that are applied on the sensor data for their manipulation and integration. Even if many approaches have been proposed for the matching of heterogeneous ontologies, they are difficult to apply in the context of sensor data where the creation of an ontology for the description of the sensors and its mapping can require much more time than the life of the sensors.

In this thesis we have developed a system, named StreamLoader, that supports the user in the creation of *Data Acquisition Plans (DAPs)* from physical and social sensors that belong to cross-domain platforms. A DAP is the composition of different services for filtering, aggregating, joining, enhancing the data generated from sensors in order to make them suitable for conducting experiments or the application of machine learning algorithms for the actuation of a given behavior. StreamLoader adopts a very flexible Syntactic Data Model in which the formats adopted in different platforms for the representation of sensor data can be easily mapped. As largely adopted in the context of Smart Cities and smart mobility, we organize the structure of the events generated by sensors according to the Spatio-Temporal-Thematic (STT) dimensions. Key point of the model is that the three dimensions are optional as well

as the properties of the thematic in order to handle sensors that only produce single-simple measurements (e.g. temperatures) or sensors that produce structured events along with their timestamps and locations. The model can be used in different contexts in which the sensors are not equipped for associating to the observations the spatio-temporal coordinates and the thematic. However, this information and other metadata can be added to the values produced by the sensor during the acquisition process. For example, a gateway in charge of a set of temperature sensors in a given zone of a city can calculate the average temperature and assign its location and current time as spatio-temporal dimensions.

The Syntactic Data Model is anyway independent from semantic associated with the information in a certain context of use. StreamLoader is also equipped with a *Domain Ontology* in which a specific meaning is given to the used STT dimensions. This ontology is developed by the domain experts that starting from standard IoT ontologies (e.g. IoT-Lite ontology [15] and the SSN ontology [31, 126]), include concepts and relationships that are usually adopted in the specific IoT domain. Moreover, the ontology can be extended with concepts and relationships of a specific application domain. Each time new sensors are made available from the underlying platforms, a Semantic Virtualization process is undertaken in which the attributes belonging to the sensor schema are semantically labeled with the concepts of the Domain Ontology. This process can be only partial because the ontology might not represent the information produced by any kind of sensors. This semantic labeling is then exploited for the automatic characterization of the sensor in the Domain Ontology and for the production of transformation rules for automatically translating the event produced by a sensor into our internal data model eventually labeled with concepts of the Domain Ontology. When the semantic characterization is full, we say that the sensor is consistent with the Domain Ontology, that is the semantics of the data produced by the sensor is well described in the adopted Domain Ontology.

Once sensors are registered in our environment, StreamLoader offers a graphical environment by means of which a DAP can be graphically specified by composing a set of different ETL (Extraction, Transform and Load) services that can be exploited for filtering, integrating, enhancing, and aggregating the streams of sensor events. The system offers facilities for checking the soundness of the produced DAP and also for the characterization, at the level of the Domain Ontology, of the schema of the data produced by each single service and by the entire DAP. In this way it is possible to check the consistency of the output stream produced by the services. Since the application of the services can alter the schema of the incoming streams (e.g. by introducing new properties or modifying the STT dimensions or transforming its properties) it is possible that non consistent streams become consistent. When no mismatching are identified between the schema of the final output stream and the ontology populated with instances for the description of the Data Acquisition Plan, we argue that the plan is "*consistent*" w.r.t. the Domain Ontology, that is, it is well

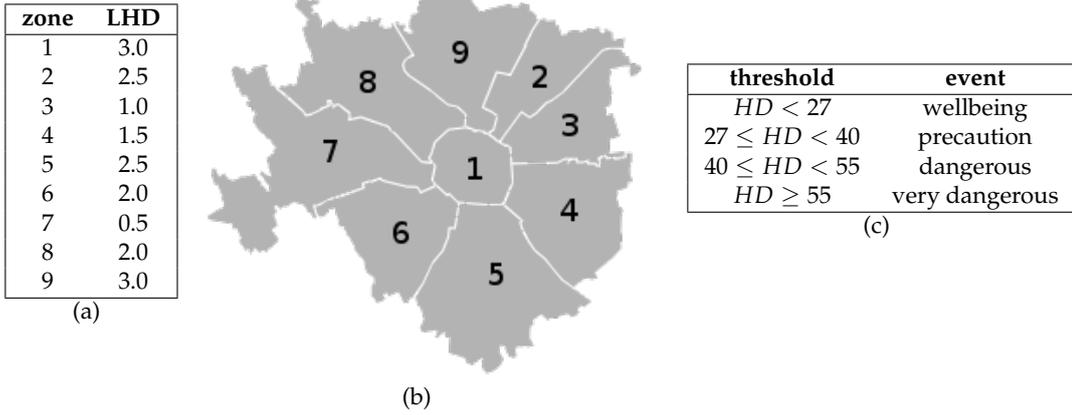


FIGURE 1: The LHD factor, the zones of the city of Milano, and the thresholds for HD

described at the semantic level. However, we also give the chance to work with Data Acquisition Plans that are not consistent. Indeed, the semantics of the final output stream can be fixed afterwards (when events are stored in a Datawarehouse or in the Cloud) or asserted by the experts that develop the plan.

The generated DAP is finally automatically translated in a language of new generation for processing big data streams and executed in a distributed environment in order to easily scale to the number of sensors and events generated by the sensors. At the current stage the DAP is translated in Apache Spark Streaming [128] because it allows to handle both stream and stored data. However, the DAP can be easily translated also in other frameworks. The events generated by the sensors are made available to the Spark Streaming script by means of the communication protocol Apache Kafka [140].

In the remainder of the Introduction we introduce a motivating example that will be exploited throughout the thesis and the general architecture on which StreamLoader is based. Finally, the overall structure of the thesis is described.

Motivating Scenario

Suppose that the mayor of Milano wishes to evaluate the *Human Discomfort* (HD) in the different zones of the city (depicted in Figure 1(b)) due to excessive heat and humidity. His/her meteorologists started from the Humidex factor [94] and proposed a formula that takes into account also the mood of people gathered by considering the tweets exchanged in the city zones. The formula is:

$$HD = [T + A + (0.555 \cdot (H - 10))] \cdot STweet + LHD$$

Where, A is the accuracy of the identified temperature T (both expressed in $^{\circ}C$), H is the humidity degree (expressed in hPa), LHD is a local correction factor that takes

<p>A</p> <pre>{ "metadata": { "thematic": "temperature", "location": { "latitude": "45.464161", "longitude": "9.190336", "data": [{ "parameter": "timestamp", "dataType": "date", "value": "2016-05-15 12:10", { "parameter": "measurement", "dataType": "real", "value": 21.4}] } } } }</pre>	<p>B</p> <pre><event> <temperature>71.24</temperature> <time>2016/05/15 12:10</time> </event> <event> <temperature>71.96</temperature> <time>2016/05/15 12:12</time> </event></pre>
<p>C</p> <pre>15-05-2016 13:08, {45.441524, 9.085241}, 'humidity', 32.3 15-05-2016 13:09, {45.441524, 9.085241}, 'humidity', 32.1 15-05-2016 13:10, {45.441524, 9.085241}, 'humidity', 31.9</pre>	<p>D</p> <pre>2016-05-15 12:00, z1, ['today is very hot', 'In my home there is much heat'], 2 2016-05-15 12:01, z1, ['a nice day for me', 'hot, hot, hot', 'I am walking'], 3 2016-05-15 12:20, z2, ['I am sweat', 'My, It's hot in here'], 2</pre>

FIGURE 2: Formats of the events generated by the sensors in the zones of Milano

into account the position of each zone within the area of Milano (the *LHD* factors are depicted in Figure 1(a)), and *STweet* is the percentage of tweets containing the words *hot*, *heat*, and *sweat* among those exchanged in a given zone. This formula should be evaluated every hour when the maximal temperature in the area of Milano is greater than 20°C and the obtained values are used to evaluate the level of alert (Figure 1(c)).

Suppose now that in the area of Milano there are many sensors that can be exploited for the computation of *HD*, but these sensors belong to different IoT platforms and return the events using different formats, and with different spatio-temporal granularities. A first platform contains sensors of type T_1 that allow to gather the temperatures every 10 minutes in Celsius degree with an accuracy of the retrieved value of more or less 3 degrees (their JSON format is reported in Figure 2(a)). A second platform contains: *i*) sensors of type T_2 for gathering the temperatures every 20 minutes in Fahrenheit degree with the XML format in Figure 2(b) (but no geo-spatial location is provided and they act only during spring 2016); and, *ii*) sensors of type H_1 that allow to generate the events about humidity every 30 minutes with the CSV format in Figure 2(c). Finally, a third platform is equipped with sensors of type TW_1 that generate every minutes the list and the total number of tweets that are exchanged in a given zone of the city in the CSV format in Figure 2(d).

The information needed for computing *HD* is thus present in the three platforms

(and in the metadata associated with their sensors) but need to be acquired, normalized, integrated and elaborated before being ready for the computation. For example the events of sensors of type T_1 presented in Figure 2 has to be enriched with information about the accuracy of the gathered temperature, as well as the events of other sensors need to be integrated with other information about their spatial, temporal and thematic dimensions. With this goal, a Data Acquisition Plan should be defined that can be applied to the streams of events that are generated by the different sensors in order to produce the information required for the computation of *HD*. First, the schema of the sensors needs to be mapped to an internal data model in which, when available, the temporal, spatial and thematic dimensions are pointed out. This guarantees the adoption of a common model within our system (though at this level we cannot guarantee the adoption of the same semantics). Then, some services are composed for filtering, combining, aggregating and enhancing the events produced by the sensors in order to lead to the specific calculus interested in the analysis.

Requirements/Design

In order to guarantee the behavior illustrated in the motivating scenario, we will exploit the general architecture depicted in Figure 3. In our architecture we assume the presence of a cross-platform middleware that exposes the sensors made available by the different architectures. The middleware offers facilities for sensor discovery and maintenance of queues of currently available sensors with their schema structure and information about spatio-temporal granularity (when available).

Semantic Virtualization is the process through which the physical and social sensors are associated with the StreamLoader services and described by means of the adopted Domain Ontology. In this activity, the schema of the events produced by the sensors are mapped to our Spatio-Temporal-Thematic data model and the correspondences with the concepts of the Domain Ontology are pointed out. Specific wrappers are then realized for transforming the input formats (e.g. CSV, XML, JSON) of the sensors of a given type in a JSON representation conforming to our STT data model. The available/registered sensors are handled by means of a Publish-Subscribe system that is present in the cross-platform middleware. For each sensor, when available, the middleware reports the frequency at which events are generated, the spatial coordinates covered by the sensors, and the produced thematics. This information is exploited for the retrieval of the sensors that are more adequate for a given kind of analysis and for establishing the kinds of data sources that can be exploited for enriching the sensors events.

StreamLoader offers a set of services for the data acquisition process that can be exploited by the user in the definition of the Data Acquisition Plan (DAP). These services can be composed in order to work on the events made available from the

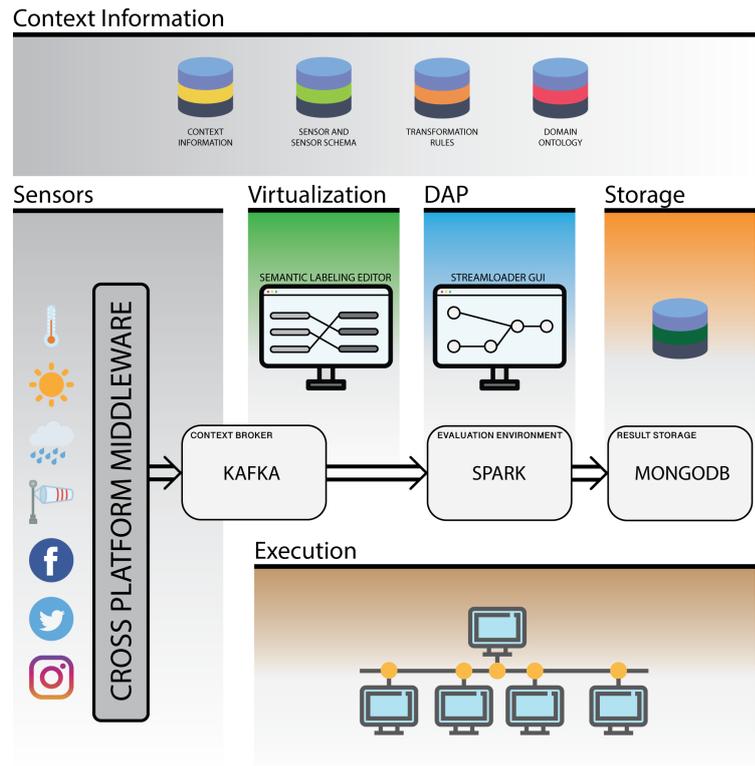


FIGURE 3: StreamLoader General Architecture

sensors through the virtualization process and lead to establish the flow of events that need to be stored or analyzed. The composition process is guided by means of the Ontological model that give feedbacks on the quality of the obtained event streams. We remark that we wish to obtain a very flexible system that is able to work also with low quality data and that only partially adheres to the STT model.

In our architecture new services can be added quite easily. The designer has to decide whether the new service will work on each single event (non-blocking service) or on windows of events (blocking service) and produce the corresponding code and applicability conditions. The new service will be made available/invokable for the data acquisition process.

The whole system is executed on a cluster of machines and results are stored on a database in order to be subsequently analyzed.

Contributions

In this thesis we developed on approaches for handling the variety and velocity of stream sensor data in the context of Internet of Things. More specifically, we have worked on the following research issues: *i*) how to deal with the heterogeneity of

the formats with which data streams are produced by sensors belonging to heterogeneous contexts; *ii*) how to easily design Data Acquisition Plans on the sensor data streams and efficiently execute them.

Due to the heterogeneity of the data produced by sensors, we initially worked on the definition of Syntactic Data Model that can be used for abstracting the schema of the data generated by the sensor and represent them in a flexible Spatio-Temporal-Thematic model that is general purpose and with no predefined semantics. The main challenge was to address in the most efficient way the issue of time granularity, as well as that for space granularity. Then, we proposed a semantic annotation of the sensor data model against a Domain Ontology (i.e. an ontology based on the SSN and IoT-Lite ontologies) in a semi-automatic way, in order to clearly provide (when possible) a well-defined semantics of the produced stream. Finally, all these facilities have been integrated in a semantic virtualization process according to which sensors made available by heterogeneous IoT platforms can be integrated and semantically annotated in an environment. In this way the events generated by the sensors can be treated according to a specific semantics. The semantic virtualization process is a valuable approach to the problem of interoperability among cross-domain IoT platforms at the data-management layer.

In the second direction, we proposed a graphical language for the specification of Data Acquisition Plan on the sensor data by composing data acquisition operations exploited for filtering, aggregating and integrating the sensor data. The problem we addressed in this direction has been to propose a tool that could help domain experts and could be easy to learn and use. Moreover, we developed the concept of soundness and consistency to be applied both on the original data produced by the sensors and on the results of the application of a Data Acquisition Plan. Finally, we defined a syntax-based translation of a Data Acquisition Plan in a Apache Spark script for its execution in a cluster of machines. The idea is to exploit the natural scalability of the Spark framework for dealing with the high input rates with which data produced by the sensors need to be processed. The execution of the Spark program is realized in a real context in which different heterogeneous sensors are made available through a Publish-Subscribe system (Apache Kafka) and the data they produce need to be processed in real (or quasi-real) time.

The overall system, named StreamLoader, has been implemented and an interactive environment has been developed that supports the user in charge of integrating sensor data belonging to cross-domain platforms to discover the sensors useful in a given situation, specify the adequate DAP for extracting, filtering, integrating, aggregating, (eventually) storing, and analyzing the events coming from the identified sensors. StreamLoader provides domain experts with a visual environment that according to their competencies can be used for configuring, manipulating, and accessing the flow of data and events that charactering their IoT domain. Since domain experts are usually not computer experts, specific attentions have been devoted to

create easy-to-use interfaces that support them in the Semantic Virtualization of heterogeneous sensors according to the Domain Ontology that easily adapt to their mental model and in the design of DAPs able to extract meaningful information and to analyze them.

Organization of the Thesis

Chapter 1 is dedicated to the presentation of IoT Platforms. First, it discusses the main building blocks at the base of any IoT platform according to a four-layer model. Then, it presents the main features of some of the most relevant IoT platforms available on the market pointing out those that can be exploited for vertical and horizontal applications. The chapter concludes with a comparison of the presented platforms and with a discussion on the strategies used to face the problem of interoperability.

Chapter 2 is devoted to present the issues of Interoperability. After presenting the main form of interoperability issues, it moves to describe the specific issues in the context of IoT. Since ontologies are the main means for addressing the semantic interoperability issues, we present some ontologies that have been proposed in the context of Smart Cities and IoT. The last part of the chapter provides an overview of the common techniques used in Ontology Mapping. These techniques are the standard approaches for guaranteeing the semantic interoperability based on the use of ontologies among different applications.

Chapter 3 presents the main characteristics of big data processing systems. The chapter first presents the main characteristics of big data and then introduce the communication protocols that can be adopted for passing the sensor data to the processing system. The chapter also discusses and compares three categories of systems (batch, stream and hybrid) developed for processing big static and stream data. For each category, examples of systems are provided and compared.

Chapter 4 presents the Syntactic Data Model that we propose to model the sensor data according to the Spatio-Temporal-Thematic (STT) dimensions. The STT dimensions are then formally described along with the spatio-temporal granularities and the event stream data model. The last part of the chapter focuses on presenting the concepts and features of our Domain Ontology and how the STT dimensions are represented through it.

Chapter 5 describes the Semantic Virtualization process. The process allows to discover a new sensor made available from the cross-domain platform, semi-automatically extracting its schema, and to semantically annotate it with concepts of the Domain Ontology. Then, the approach for the characterization of the discovered sensor in the Domain Ontology is discussed along with the concept of *consistency*. The last part of the chapter is dedicated to present the approach for the automatic creation of transformation rules to translate the sensor observations into our internal data model.

Chapter 6 gives a formal definition of every service that can be devised for processing and combining the streams produced by the sensors. Then, the concept of Data Acquisition Plan is introduced along with real-case examples. The definitions of

Soundness and Consistency of a DAP are introduced and the algorithm for evaluating the consistency of a DAP with respect to the Domain Ontology is presented.

Chapter 7 presents the overall StreamLoader system. The first part focuses on its architecture and the adopted technologies. Then, an exhaustive description on the interfaces for the Semantic Virtualization and the DAP design is provided. The last part of the chapter presents the translation of the DAP into a runnable Apache Spark Script.

Chapter 8 is focused on the presentation of the experimental activity. After presenting the working environment and the metrics used for evaluating the system, we describe the results of two kinds of experiments. The first one is devoted to evaluate the performances of the execution of the DAP locally or in a cluster of machines. The second one is devoted to evaluate where the transformation rules should be applied for transforming the sensor data in the internal format.

Conclusions on the entire work that we carried out on this thesis are reported in the last chapter. We provide a discussion of the thematics and of the issues that have been addressed and present possible future work.

Chapter 1

IoT Platforms. Current Features and Future Trends

Kevin Ashton, the British technologist [119, 60, 114, 9], introduced for the first time the term *Internet of Things (IoT)* in 1999 to describe *the system of physical objects in the world that connect to the internet via a sensor*. He also created a global standard system for *Radio-Frequency Identification (RFID)* and other sensors that *tags the physical objects to the internet for the purpose of counting and tracking of goods without any human interference* [119]. Starting from these definitions, many institutions have proposed definitions that promote particular concepts in the whole world. The Internet Architecture Board (IAB) defines IoT as the networking of smart objects, meaning a huge number of devices intelligently communicating in the presence of internet protocol that cannot be directly operated by human beings but exist as components in buildings, vehicles or the environment. The Internet Engineering Task Force (IETF) defines IoT as the networking of smart objects in which smart objects have some constraints such as limited bandwidth, power and processing accessibility for achieving the interoperability among smart objects [131]. For the IEEE Communications category magazine, the IoT is a framework of all things that have a representation in the presence of the internet in such a way that new applications and services enable the interaction in the physical and virtual world in the form of Machine-to-Machine (M2M) communication in the cloud [160].

More recently, the term IoT has been used also for representing the social sensors, that is flows of data produced by social networks like Twitter, Facebook, etc. In this direction there is the definition proposed in [53] as an "omnipresent network, consisting of physical and virtual objects/resources, equipped with sensing, computing, communication and actuating capabilities" that moves in the direction to realize *ubiquitous computing* [149, 51]. The research introduced in [97] shows that there is a tight relationship between the real world events and tweets. Starting from this concept, an interesting research has been conducted in [157] where a new method that explores Spatio-Temporal-Thematic correlations between physical and social data streams has been introduced for event detection and pattern interpretation from

heterogeneous sensors. Other works [156] focus their attention on the recognition of asthma risk factor from heterogeneous physical and social sensors.

According to Gartner Inc.¹ in 2020 more than 20.8 billion connected things will be in use worldwide. These can be either physical devices or social objects which produce large, complex, heterogeneous, structured and unstructured data. Dealing with the vast amount of data produced by the things, their varying capabilities, and an exploding number of services that can be realized, are around the biggest conceptual and technological challenges of our time [54]. This challenge is further exaggerated by the typical ills of early-stage technology that lead to the production of different solutions in different context of use that rarely are able to collaborate each other. This has led to the creation of more than 600 IoT platforms (according to a recent survey [5]). By exploiting these platforms, many "vertical" applications can be developed. However, these solutions are characterized by the use of hardware and software of a specific industry and the interoperability with other applications is rarely supported. This lack of interoperability prevents the emergence of broadly accepted IoT ecosystems [24]. A recent McKinsey study [93] estimates that a 40% share of the potential economic value of the IoT directly depends on interoperability gaps among IoT platforms.

As a result, the development of "horizontal" applications that exploit sensors, actuators and infrastructures made available by different platforms is almost complex with negative effects on the development of vibrant IoT ecosystems. These technological barriers also have negative impact on the business opportunities, especially for small innovative enterprises, which cannot afford to offer their solutions across multiple platforms. In order to face these issues a number of European Projects are currently under development with the purpose of generating infrastructures among heterogeneous platforms that facilitate the development of "horizontal" applications.

This Chapter is organized as follows. Section 1.1 introduces the building blocks of IoT platforms according to a four-layer model that moves from the acquisition of raw data from the sensors to the development of applications in a given applicative scenario. We provide a description of the main technologies adopted for the acquisition of data from sensors (*device layer*), their intermediate collection and processing by means of gateways (*gateway layer*) that can be positioned closer to sensors (the so called *edge computing*) or on the cloud offering different kinds of service, their aggregation and processing by means of scalable facilities (*integration layer*) and the applications that can be built on top of them (*application layer*). Section 1.2 provides a general description of some of the main IoT platforms available on the Market by means of which vertical applications can be easily deployed and discusses new trends in the context of cross-domain platforms that are under investigation from some European Projects for supporting the development of horizontal applications. Section 1.3 provides a comparison of these middlewares and discusses the solutions

¹<https://www.gartner.com/newsroom/id/3165317> (2016)

adopted for dealing with the barriers of interoperability. The discussion points out how, at the current stage, there are mainly syntactic strategies that are adopted for dealing with interoperability. Finally, Section 1.4 provides some remarks regarding the proposed IoT platforms.

1.1 IoT Architecture Building Blocks

As we will describe in the following section, many IoT Platforms have been proposed to deal with data produced by physical and social sensors in the last few years. These platforms rely on different architectures and there is no general consensus on the adoption of single IoT Architecture. Among the proposed architectures we mention the following ones proposed by international bodies and industry consortia.

- the *Industrial Internet Consortium* has delivered the *Industrial Internet Reference Architecture (IIRA)*², with a strong industry focus specifically on industrial IoT applications.
- The *Internet of Things IoT-A EU*³ initiative delivered a detailed architecture and model from the functional and information perspectives.
- The *Reference Architecture Model Industrie 4.0 (RAMI 4.0)*⁴ goes beyond IoT, adding manufacturing and logistics details. This is effectively a reference architecture for smart factories dedicated to IoT standards.
- The *IEEE P2413*⁵ project formed a working group for the IoT architectural framework, highlighting protection, security, privacy and safety issues.

What all these architectures have in common is a clear separation into layers providing a separation of concern on how the most important aspects of the architecture operate. Layers can exploit specific technologies and components or cross-cutting aspects such as security and management. Conceptually, device, gateway, integration and application layers can be identified as reported in Figure 1.1 These layers are the building blocks according to which applications that need to handle big flows of sensor data (eventually processed in real time) need to be realized. They are discussed in the remainder of the section.

1.1.1 The Device Layer

This layer is also named Perception or Recognition Layer and its main responsibility is to collect useful information/data from *things/sensors* or the environment (such as

²https://www.iiconsortium.org/IIC_PUB_G1_V1.80_2017-01-31.pdf

³http://www.meet-iot.eu/deliverables-IOTA/D1_5.pdf

⁴http://www.plattform-i40.de/I40/Redaktion/EN/Downloads/Publikation/rami40-an-introduction.pdf?__blob=publicationFile&v=4

⁵<http://standards.ieee.org/develop/project/2413.html>

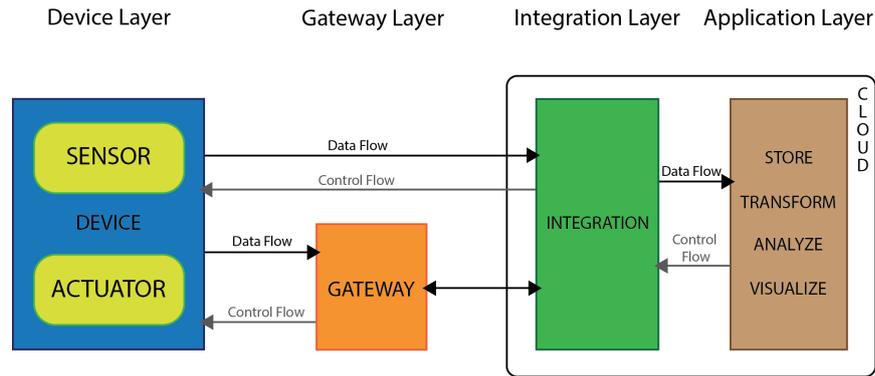


FIGURE 1.1: Building blocks of an IoT architecture

WSN, heterogeneous devices, sensor type real world objects, humidity and temperature etc.) and transform them in a digital object that can be transmitted through the Internet. These things can be either physical or social and produce large, complex, heterogeneous, structured or unstructured data.

Physical sensors nodes are hardware components that are able to measure physical entities or to actuate a given behaviour (e.g. turn on the light of a room). They can be standalone components or integrated with other kinds of sensors in a single motherboard. Sometimes they are only able to make available a single measurement (e.g. the temperature), others they (or the device in which they are integrated) are equipped with an Operating System able to perform local computation. These things can communicate each others by means of wired or wireless networks and produce data that are collected in a gateway or can transmit their data directly to the Application Layer for their management. Among the many physical sensors, we mention: *i*) devices able to detect data about physical phenomena (e.g. temperature, humidity, wind, rain, pressure, level of sea water); *ii*) phones, tablets, smart watches, fit bands and fitness trackers; *iii*) medical devices, smart machinery and a set of technical instruments used in the context of industry; *iv*) home automation and security systems.

There is also a proliferation of social sensors able to collect data from people. Internet transactions, email, video, click streams, Twitter, and Facebook are the major examples of social sensors that can be exploited for evaluating the social characteristics of Internet users. The data produced by these social sensors can be combined with data produced by physical sensors in order to respond to emergency situations. For example, temperatures and levels of rains detected in a given area can be combined with the tweets posted by people moving on the highway in order to identify situation of flooding. Moreover, the presence of a peak of tweets containing specific hashtags (e.g. #rain, #water, #storm) in a specific interval of time can trigger the analysis of the levels of water in the rivers in order to discover if certain emergency condition may occur, for example torrential or heavy rain.

1.1.2 The Gateway Layer

Sensors and actuators can be connected or embedded into devices and be connected to the Internet in two ways. Directly, with the Integration Layer over IP protocols such as REST, Message Queue Telemetry Transport (MQTT), Extensible Messaging and Presence Protocol (XMPP) or Advanced Message Queuing Protocol (AMQP) (described on Section 3.2). Devices can also be connected indirectly by means of an aggregation node that acts as an intermediary, or *gateway*, to aggregate data from devices with a small footprint (Constrained Application Protocols), short-range communication (such as Zigbee devices connected via a Zigbee gateway) or Bluetooth Low Energy devices connecting via a mobile phone. The gateway filters and intelligently reacts to data and sends and receives data or commands to the Internet.

A gateway device is used to connect previously unconnected devices, older devices, and insecure devices. It can also provide operational efficiency by allowing multiple devices to share a common connection. Moreover, they can provide functions such as data filtering, cleanup, aggregation and packet content inspection. The gateway device might be responsible for managing security on behalf of the locally connected devices as a proxy for the other devices that are connected to the outside world.

When gateways are positioned closer to the sensors they can allow a form of *edge computing* [133, 20]. That is the possibility to forward to the other tiers only useful events. Moreover, events that occur at the same time can be collected in a single structured message and contextual information can be included in the measurement in order to provide sophisticated computations that take into account the environment in which the data are collected. Sometimes, gateways are also positioned in the cloud. In this case, they offer more processing power for the management of the events generated by sensors. Finally, hierarchies of gateways can be generated when the number sensors to be handled is very high and distributed in a vast area (like in smart cities, in the context of automotive). For example, connected vehicles contain many sensors and processors that are themselves unsecured and connected only to the local controller area network (CANbus) in the vehicle. In this scenario, a gateway acts as a communication middleware between the vehicle and the outside world. The gateway aggregates data from the other vehicle subsystems to communicate to the Internet and interprets commands or data that are received from the Internet. The gateway redistributes the data and commands through the local CANbus to the other vehicle subsystems. In an industrial environment, such as a manufacturing facility, it is common to find devices that are connected by means of existing industrial protocols, such as Modbus, Profibus, or DeviceNet, to a local gateway device. The local gateway can aggregate data, filter data and perform local analytics. It can also connect to a cloud or back-end server to propagate data up to higher-level systems and analytics.

1.1.3 The Integration Layer

Typically hosted in the cloud, this layer is responsible for receiving data from the connected devices and storing them either in-memory or in databases for their further processing. Moreover, this layer hosts many components for the protection, integration and the processing of data. In this way the events generated by the underlying levels are made available to the Application Layer. This layer has thus the purpose to generate a bridge from the things that produce events and the applications by offering an Application Programming Interface (API) for communication, data management, computation, security, and privacy that allows the developers to focus on the requirements of applications rather than on interacting with the baseline hardware.

Among the services offered for the integration layer [127, 30, 117], we need to mention:

- *Interoperability and programming abstractions.* These services are meant for facilitating collaboration and information exchange between heterogeneous devices. In the context of IoT, the issue of interoperability can be identified at network, syntactic, and semantic levels. Network interoperability issues are due to the use of heterogeneous interface protocols for communication between devices that prevent applications to consider events generated by sensors connected with unknown protocols. Syntactic interoperability issues arise when events are generated using different formats, structures, and encoding of data. Even if applications use the same syntax, they might assign different meaning at the same piece of data that can generate semantic interoperability issues. A deeper analysis of the interoperability issues will be discussed in Chapter 2 with a discussion of the current efforts for their addressing.
- *Device discovery and management.* These services allow devices to be aware of the presence of other devices in the neighborhood and to deal with them. These services should be scalable and efficient in order to take into account the dynamism with which a high number of devices can be included (and excluded) from the IoT. APIs are offered to list the IoT devices, their services, capabilities and for discovery them relying on their capabilities. Finally, software components should be made available to perform load balancing, manage devices based on their levels of battery power, and report problems occurring in devices to the users.
- *Security and privacy.* Events generated by the monitored sensors can contain sensitive information about the personal life or the activity of an industry. Therefore, different mechanisms should be made available to guarantee the protection of the data from the sensor to the application in charge of its management, as well as to guarantee the privacy of single users in case of analysis. Moreover, data should be accessible only from authenticated users.

- *Context detection.* The Integration Layer can be equipped with contextual information systems that can be exploited for enriching the data generated by the sensors and for easily identify abnormal values produced by the sensors or their faulty. The contextual information associated with events can be subsequently exploited for the analysis of the data and for taking appropriate actions on the status of the devices.
- *Cloud services.* The aforementioned services are usually executed on a centralized cloud. There is therefore the need to be able to run on different types of clouds and to enable users to leverage the cloud to get better insights from the data collected by the sensors.

These services are offered from a variety of platforms proposed in the market for dealing with different kinds of sensors. Many of them offer support for interoperability and abstraction. In the next section, we will discuss in more details the characteristics of these middleware systems that can be broadly classified on the basis of the design [117] in:

- *Event-based.* In these kinds of platforms, the interact among components is guided through events. Events have a type and some properties and are exchanged through a Publish-Subscribe approach. Consumers interested to a given type of events can subscribe to such a type and be notified when those events are generated from the underlying architecture.
- *Service oriented.* These platforms rely on the Service Oriented Architectures (SOA) [25], in which independent modules provide services through accessible interfaces. A service oriented middleware views resources as service providers and abstracts them through a set of services that are used by applications. There is a service repository containing the services that providers make publicly available. Consumers can discover them from the repository and then bind with the provider to access the service. Advertising services are usually included in these platforms for supporting consumers in the discovery and use of services.
- *Database oriented.* In this case, the network of IoT devices is considered as a virtual relational database system. Accessing, discovering and manipulating devices is handled through easy-to-use interfaces that pose queries on such relational database systems. Even if this approach make the management of devices very easy, it presents scalability issues due to the use of a centralized model.
- *Application specific.* This kind of platforms is used specifically for the application domain for which it has been conceived. Its whole architecture is fine-tuned on the basis of requirements of the application and rarely can be exploited in other contexts.

1.1.4 The Application Layer

On top of the discussed platforms, many application can be realized that can visualize by means of dashboards the events monitored by the developed sensors, perform different kinds of analysis by exploiting machine learning algorithms, process and prepare the data for the generation of reports, actuate different behaviour on the monitored devices (eventually exploiting the results of machine learning algorithms on the monitored data).

The size, velocity and variability of the data generated from the sensors can require to exploit new distributed techniques for processing the data. The term "lambda architecture" has been introduced for a generic, scalable and fault-tolerant data processing architecture, with data, batch, serving and speed layers. It is possible to build advanced solutions based on such an architecture by exploiting NoSQL frameworks [74] such as Apache Kafka and Apache Spark for downstream processing. These systems will be discussed in more detail in Chapter 3.

In the remainder of the section we discuss some application contexts in which applications have been realized for managing sensors data.

Smart Homes. In the context of home automation, different sensors can be installed in a house for providing intelligent and automated services to the users. They help in automating daily tasks and help in maintaining a routine for the individuals who tend to be forgetful. For example monitoring the internal temperature, humidity, the presence of gas or smoke, the level of lightness, for checking and activating white and black equipments (air conditioners, oven, dish and blanket washers, etc.). Many challenges and issues are connected with smart home and home automation [72]: *i*) energy conservation [65], helps on disconnecting or turning off lights and electronic devices automatically when their are not used; *ii*) support for elderly and differently abled users [102, 135, 153], by monitoring their health, relatives can be immediately informed in case of emergency or in case a person falls on the floor (by installing pressure sensors on the floor); *iii*) security and privacy [118], by installing Closed Circuit Television (CCTV) cameras, motion detection sensors everything is happening inside and outside the home is being recorded. Figure 1.2 provides a diagram for the main issues on Smart Home systems.

Smart Cities. As shown on Figure 1.3 the context of Smart Cities provides a wide area of interests. Smart mobility application can help on managing traffic in cities, reducing or minimizing the traffic congestions, ensure the parking discovery, avoiding accidents, car and bike sharing and spotting drunk drivers. Most of these applications are based on GPS sensors for location, accelerometers for speed, gyroscopes for direction, RFID for vehicle identification, infrared sensors for counting people or vehicles and cameras for traffic recording and security [36]. Some of these applications give visual information and can estimate traffic conditions in a certain area. Infrared sensors, cameras and GPS sensors can help on detecting traffic congestions

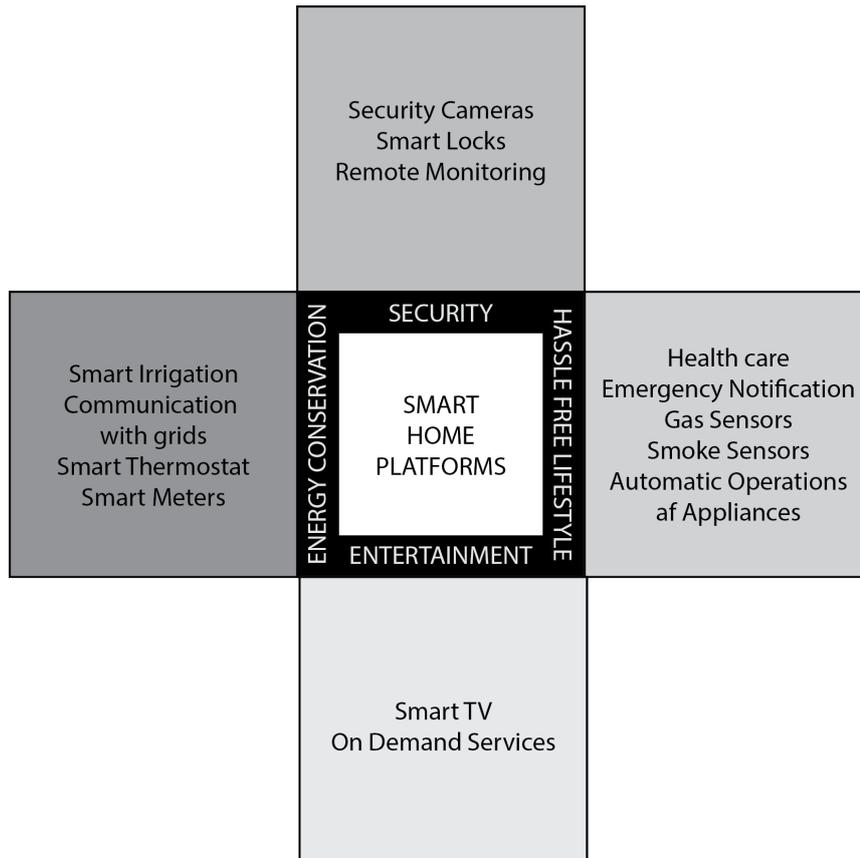


FIGURE 1.2: Block diagram of a smart home system

but some applications implement this system by using smartphone sensors, such as accelerometers and GPS sensors [91].

Many applications aim at the driver security. Sensors (e.g. eye movement detectors, face detectors, pressure detectors on the steering) help on monitoring the physical conditions of the drivers by detecting when they are tired or they are falling asleep. The creation of smart parking grids helps the drivers in discovering free parking places. Smart traffic lights equipped with sensors sense the traffic congestions at the intersections. This can modify the interval between red and green light and it can also help in emergency situation by giving way to ambulance or police. Another field of interest in the context of smart cities is energy and environment. For example, smart water systems and applications, especially in areas where water scarcity is a big issue, are used to measure the degree of inflowing and outflowing water with the aim of discovering and identifying possible leaks [66]. If they are associated with data from weather satellites and river water sensors can also help on flooding prediction. Security is another big issue in the context of smart city. Video surveillance and Closed Circuit Television (CCTV) camera provide monitoring of all the city areas.

Social Life and Entertainment. The "opportunistic IoT" term [61], is used to refer to a set of applications that shares information among devices that reside in the same

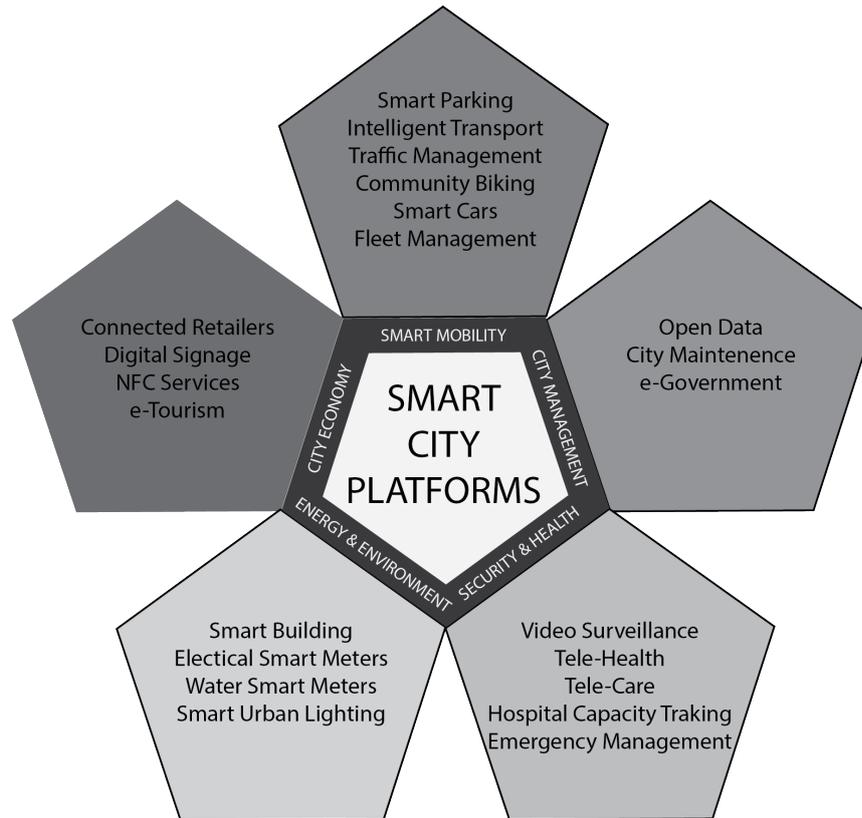


FIGURE 1.3: Block diagram of a smart city system

area and based on position and availability of contacts. The main issue of these kinds of applications is the tracking of human social life and activities. Circle Sense [88], Camy [120] and LogMusic [86] are applications that have been developed with the aim of finding people with common interests or purposes and helping them to interact. While Circle Sense is an application for supporting users in detecting their social activities and habits through the analysis of patterns of social activities and the people movements around them, the other two belongs to the affective computing technology [108] that leverages and responds to the emotions of the user. With the analysis of body gesture, facial expressions, sleep patterns, it is possible to understand the feeling of a person. Camy is a virtual/artificial pet dog that interacts with the human and her emotions and feelings. It provides emotional support, encourages playful and active behaviour, also with other people, and increases the love for himself/herself. Logmusic is an applications that proposes and recommends songs on the basis of location, temperature, weather and time.

Health and Fitness. Thanks to the advent of wearable devices, such as smart watches, fit bands and fitness tracker and due to the introduction and improvement of accelerometers, gyroscopes, GPS, many health applications have been developed. Most of them are focused on monitoring fitness activities (e.g. daily number of steps taken, the distance, the amount of calories burned, real time heart-rate) and let the

users connecting with gym apparatus. Some applications help and make independent living possible for elderly and patients with critical or serious health problems, by continuously monitor health conditions and transit warning in case of abnormal indicators or values. This kind of applications can be linked to Electronic Health Records (EHR), which is a record of all medical details of a person which may include a range of data, including demographics, medical history, medication and allergies, immunization status, laboratory test results, radiology images, vital signs, personal statistics like age and weight, billing information, but also primary care information (such as General Practitioner) and secondary care information (such as hospital). Other applications aim at recognizing stress level [49] by monitoring the movements during the whole day, the amount of physical activity, amount of rest and sleep, and, through audio data and calls, they can also monitor interactions and relationship with other people [148].

Smart Environment and Agriculture. Agriculture can be significantly improved through the use of smart applications. Sensors measuring temperature, humidity, soil information and other parameters can be useful used for a better and more efficient production, or to improve crop quality and yield [159]. For example, automated irrigation according to weather conditions can sensibly reduce the waste of water, the use of acetylcholinesterase biosensors [158] give information about size, time, location and amount of pesticide residues. The introduction of QR code can help on speed up the process of ordering products and to check the amount of products available online before buying. Regarding environment, air pollution is a great concern today. The possibility to measure the quality of the air, to identify via RFID tags polluting vehicles and take action against them is a great challenge that some applications try to overcome [92].

Supply Chain and Logistics. IoT tries to simplify real world processes in business and information systems [44]. RFID and NFC can help on easily tracking goods in the supply chain, from the place of manufacture to the final places of distribution. RFID tags uniquely identify a product automatically and provide information about the product in real time along with location information. This system helps in automatic collection and analysis of all the information related to supply chain management, which may help examine past demand and come up with a forecast of future demand. The possibility to access in real time these data by the supply chain components, give them the possibility to analyze, extract useful information and insights and improve, in the long run, the performance of supply chain systems [151].

Energy Conservation. Smart grid is the information and communication technology that enable modern electricity generation, transmission, distribution, and consumption system [71]. This technology introduces the concepts of scalable energy and bidirectional flow of power (back from the consumer to the supplier) and can be used to make smart electric power generation, transmission, and distribution. In a real application a smart grid is a set of different microgrids [42] that generate power

in order to provide enough energy to the local sites, but in case of emergency or in case of shortfall they can also demand energy from the central grid. Two-way flow of power also benefits consumers, who are also using their own generated energy occasionally (say, solar, or wind power); the surplus power can be transmitted back so that it is not wasted and the user will also get paid for that "unused" power. Some of the IoT applications in a smart grid provides online monitoring of transmission lines for disaster prevention and efficient use of power in smart homes by having a smart meter for monitoring energy consumption [90]. Smart meters read and analyze consumption patterns of power and send this information to the server. Users can consult their consumptions and can adjust their use so as to reduce costs.

1.2 IoT Platforms

According to a recent survey [5] in the last few years has been proposed more than 600 IoT platforms that demonstrates a great interest from both the research and industrial communities in the development of infrastructures for the acquisition, integration and analysis of events produced by sensors of different types. These platforms can be divided in *vertical* and *horizontal* platforms. Vertical solutions are characterized by the use of hardware and software of a specific industry and the interoperability with other applications is rarely supported. This means that if we have an application for monitoring the energy consumption in a smart home platform and one for environment monitoring in a smart city platform there is no easy way to combine them in the scope of a new added-value application, like for example determining the levels of the internal heater relying on the weather forecast for the next hours. To overcome these limitations, a considerable amount of work is dedicated to the construction of IoT infrastructures that allow the development of *horizontal* applications, that is applications that exploit sensors, actuators and network components belonging to cross-domain IoT platforms.

In the remainder of the section, the considered platforms are briefly described pointing out their main characteristics.

1.2.1 Vertical IoT Platforms

Starting from a comparison among the most relevant and used platforms in the market [152, 143], in this section we discuss the main characteristics of a set of vertical platforms. Table 1.1 introduces the name of these platforms along with their main drivers and the url where further details can be found. Moreover, Table 1.2 reports the common domain in which they are usually adopted, the licensing models used for the distribution and use and their Technological Readiness Level (TRL)⁶. The

⁶http://ec.europa.eu/research/participants/data/ref/h2020/wp/2014_2015/annexes/h2020-wp1415-annex-g-trl_en.pdf

Name	MAIN DRIVERS	URL
BEZIRK	Bosch CR Target at Open Source	http://www.bezirk.com
Bosch SCP	Bosch	https://www.bosch-iot-suite.com/
CSI Piemonte	CSI Piemonte	http://www.smartdatanet.it/
FIWARE	EU Project	https://www.fiware.org
WordSensing	WordSensing	http://www.worldsensing.com
TIC platform	VMZ Berlin	http://viz-info.de
IFTTT	IFTTT Inc.	https://ifttt.com
Wubby	Econais	http://wubby.io
Xively	LogMeln	http://xively.com
Kaa	Kaa Project	https://www.kaaproject.org/
Carriots	Altair	https://www.carriots.com
macchina.io	Netidee	https://macchina.io/

TABLE 1.1: Examples of IoT Platforms

TRL, as defined by the European Commission, is used to enable comparison of the maturity of the analyzed technologies.

BEZIRK. It is an open platform by the Robert Bosch Start-up GmbH. It is a peer-to-peer IoT middleware for both communication and service execution on local devices following a service-oriented paradigm. It is developed with the idea of facilitating asynchronous interactions among various components or services of an application that are distributed across different devices in a network. It is implemented on top of UDP over Wi-Fi within local networks, and on top of TCP sockets for internet communications. BEZIRK was primarily developed for consumer IoT domains (e.g. smart home) but can also be adopted in other domains where IoT devices communicate and collaborate in a local environment, or where users desire to manage and control their privacy. Main features are: *i*) handling reliable delivery of messages among devices or services; *ii*) hiding service distribution; *iii*) enforcing security and privacy policies specified by the user; *iv*) encrypted communications; and *v*) facilitating dynamic service discovery and semantic addressing. Interoperability is facilitated by the adoption of Open API, Open Protocol and Open Source. The decentralized design requires no infrastructure that facilitates local integration without any dependencies on back-end infrastructure or providers. This platform also enables easy middleware integration with new IoT platforms by adopting standard communication protocols (e.g. TCP/IP for data transport, JSON for data encoding) and available Java implementations for different operating systems (Linux, OSX, Windows, Android).

Bosch's Smart City Platform (SCP). This tool has been developed for composing heterogeneous solutions in the Smart City environment (i.e., governance, mobility, energy, environment, industry life, tourism, etc.) and adopts a SaaS business model.

Name	COMMON DOMAIN	LICENSING MODEL	TRL
BEZIRK	IoT domains	Not fixed	5-6
Bosch SCP	Smart City	SaaS - commercially	7
CSI Piemonte	Smart data	Common licenses CCO or CC BY	9
FIWARE	Smart Citiy, agriculture and food safety, health/AAL, etc	Difference depending on functionality	5-6
WordSensing	Smart City, smart mobility	Not applicable, not for public	8
TIC platform	Smart Mobility	SaaS (commercially available)	9
IFTTT	Business, commerce, connected car, connected home, fitness and wearable, Web etc.	The platform is not open source, It provides a service free of charge	9
Wubby	Low power 32bit devices	Depending on components	5-6
Xively	Smart Home and Smart Lab	Commercial access through PaaS	9
Kaa	Agriculture, automotive, health-care, IoT industry, Smart City and Smart Home, wearable and telecom	Open source: Licensed under Apache Software License 2.0	7-8
Carriots	Smart Home and Smart City	PaaS - free up to 2 devices	7
macchina.io	IoT domains, hardware domains	Open source: Licensed under Apache Software License 2.0	7

TABLE 1.2: Main IoT Platforms

It offers tools and methods to develop, operate and maintain such systems without sacrificing data security and privacy. The key aspects of the Bosch SCP platform are: secure data routing/fanout, third party and developer tooling, data as a service/data tooling integration, operational support. The platform has been specifically developed for the Smart City domain, including Smart Mobility and environment domains. Smart Home and Smart Factory are currently not addressed but may be required for an IoT platform. This platform has been designed to interoperate with other solution providers and to contribute and rollout artifacts in a quality controlled multi-staged software provisioning concept. The platform is designed to be flexible enough to support a wide span of technologies and/or programming languages and to adapt to new requirements without disrupting installations and rebuilding them from scratch. Specifically, the following measures for interoperability have been stressed in the design: easy deployment on premise (e.g. notebook) or on varied cloud providers, flexible integration of diverse sensors and devices (with different protocols/APIs/standards), and ability to provide Data-as-a-Service.

CSI Piemonte Smartdata Platform (SDP). The Piemonte Region (Italy) developed this platform and made it available to public and private entities of its territories. SDP is based on project Yucca, a cloud self-service platform enabling to develop applications based on IoT and big data. SDP works as a Platform-as-a-Service as an IoT cloud-based platform and supports features such as multi-tenancy. It receives events from "things", "people" (twitter) and "applications", exposes APIs that realize a Publish-Subscribe paradigm in near real time and APIs with Request-Response

paradigm to read historical data. It uses the *ODATA protocol*, different types of visibility (opendata, public, private and shared APIs), and makes use of OAUTH2 security to access non-public APIs. Beyond that it enables Complex Event Processing in near real time to enrich or filter events. SDP can serve different domains, but is mainly oriented to data for the Piemonte region in Italy. SDP has proven itself in operational environment. The interoperability approach is to define Open APIs and send events to the platform for the data, information and service levels. It uses HTTP(s)/MQTT(s) for data transport, JSON/ODATA for data encoding, and OAuth2 for authorization. SDP can interact with other platforms by sending events over MQTT in JSON format, reading historical events through an ODATA REST API, or subscribe to events through MQTTs or Web sockets.

FIWARE. This middleware platform has been developed for the future Internet (FI⁷) by the EU commission. The FIWARE platform consists of a number of General Enablers (GEs) that have been developed by different chapters within the FIWARE and FI-CORE projects [155]. The FIWARE Internet of Things Generic Enablers allows "things" to become available, searchable, accessible, and usable resources fostering FIWARE-based Apps interaction with real life objects. The success behind FIWARE IoT is that all things and IoT resources are exposed to FIWARE App developers just as other Next Generation Service Interface (NGSI) Context Entities. Therefore, developers do not have to deal at all with today's complexity and high fragmentation of IoT technologies and deployment scenarios. On the contrary, app developers just need to learn and use the same NGSI Context-Broker API, used in FIWARE to represent all context information. FIWARE has been defined as a Future Internet core platform and used in a variety of domains such as Smart Cities, agriculture and food safety, eHealth/AAL etc⁸. The interoperability of FIWARE is achieved through a standard interface, namely NGSI-10 based sources. Core interfaces for the IoT and Context GEs are the NGSI-9/10 Context Interfaces that were originally developed by OMA and for which FIWARE defined a REST-like interface with a few extensions. In addition, the IDAS/IoT Data Edge provides interoperability with a number of existing technologies, e.g., UL2.0/HTTP, MQTT, OMA LWM2M/CoAP, ThinkingThings Protocol and SIGFOX protocol. Interoperability takes place at the interface level.

Worldsensing. This platform provides a unique traffic management portfolio for Smart Cities that includes Bitcarrier, a real time intelligent traffic management and information solution designed for both road and urban environments, Fastprk, an intelligent parking system commercially operational since November 2012 and Sensefields, an innovative system for detecting and monitoring vehicles and traffic flow.

⁷<https://ec.europa.eu/programmes/horizon2020/en/h2020-section/future-internet>

⁸<https://www.fiware.org/tag/applications/>

Worldsensing business relies on providing useful data analysis of their own information sources. Worldsensing platform is proprietary code and is not released. World-sensing transfer data through HTTP(s) in the JSON data format and through a REST API. Interoperability to other platforms/solutions can be achieved by adopting the RESTful APIs and an already integrated pull service. No raw access to the "actual" things/sensors but a bunch of functions/functionality is provided, based on the collected data.

TIC mobility platform. The traffic Information Center (TIC) of the City of Berlin has been extended in the TIC mobility platform to provide comprehensive information on all mobility options available in the city. The platform includes real time data from the traffic information center, mobility operators and infrastructure providers and offers a multimodal routing platform using the modal router offered by third parties. The system comprises of three components: *i*) a data platform integrating real time data from mobility providers; *ii*) a routing platform; and *iii*) an online monitoring system for air pollution and noise in the arterial street network of Berlin. Applications developed on top of this platform, such as mobility websites and mobility displays, are implemented and running in various environment, end user apps based on the mobility platform are qualified and are ready to be launched. The limiting factor for the platform is the provision of data from mobility providers and low standardization of interfaces. Interoperability on the platform level is enabled via Open API (charging stations), data level (detector data), service level (modal routers). The TIC Platform has been connected with the VBB platform (Public transportation data and routing services in the region Berlin/Brandenburg).

IFTTT (If This Then That). IFTTT is a website offering facilities for the definition of simple IoT applications, i.e., connections between pairs of IoT devices and services made available from third-party providers (e.g. Instagram, Dropbox, Google, Facebook). Such applications are typically expressed as trigger-action rules, where an action is automatically executed when an event (the trigger) is detected. It enables users to create chains of simple conditional statements, called "recipes", which are exposed as web services, between more than 400 supported IoT objects (named services). The supported objects range from commercial devices (e.g. the Nest thermostat), to web or mobile services (e.g., Facebook). Recipes, at least in the free version, can include a single trigger and a unique action, and can be composed by using a wizard-based procedure. Its simple service-based paradigm allows the interoperability of applications among different platforms even if some limitations are present. Indeed, as pointed out in [32], IFTTT adopts an highly technology-dependent representation models according to which devices and services are represented by manufacturers or brands. This implies that, if we have to specify the rule *"if the bedroom motion sensor detects a movement, then turn the table lamps in the*

bedroom on", for a set of movement and lamp sensors in our building, we have to create different rules for each of them. With this lack of discovery and adaptation features, the expressive power of the definable rules is very poor. Despite that, it allows to monitor the behaviour of the hundreds of connected APIs from other service providers (e.g., information about the API requests, metrics such as response time and HTTP status codes etc.) and uses elastic search and its components "Internal Monitoring and Alerting" and "Developer Dashboard". Manual integration with the hundreds of APIs is supported by IFTTT.

Wubby. It was created in 2015 as a spin-off of a company (Econais) specialized of Wi-Fi modules with very unique characteristics (smallest size, lowest power consumption, complete software, etc.). Wubby is an ecosystem of software components and services for the rapid development of physical objects embedded with electronics, software, sensors and network connectivity to collect and exchange data. The Wubby VM runs in the microcontroller of these objects and provides a hardware agnostic environment for the creation of interoperable applications. Wubby is a solution to fuel fast development of low cost, low complexity, low power and high volume IoT products made on Embedded Microcontroller Unit (MCU) regardless of the domain for which they will be used. Wubby provides interoperability at the device level (device-service discovery, initial setup) and new protocols can be added by installing new python libraries/modules in the device. Wubby allows new platforms to be integrated by allowing syntactic interoperability whenever possible (e.g. Wi-Fi Direct service discovery) and enables integration of new protocols by adding new applications in the Wubby VM. Wubby encourages interoperability on different levels: it is protocol agnostic (where users can add support for their own protocols), it supports standard wireless technologies (e.g. Wi-Fi, BLE), multiple security schemes can be integrated, and it supports device and service discovery using MQTT.

Xively. Xively is an enterprise grade IoT cloud-based platform and offers easy-to-use APIs developed as "microservices". The central goal of Xively is to enable building of a "Connected Product" and a customer shall be enabled to realize a "Connected Business". Xively is designed to be a centralized PaaS. Thus, its scalability is global. Xively is generally designed for any domains, but it has mainly been applied in domains such as Smart Home and Smart Lab. Since Xively is closed source (the internal implementation details are not disclosed) and uses Open APIs to promote interoperability. The Open API is defined at the data, information and service levels. It provides four key functionalities: *i) blueprint service*, to describe the metadata and relations between device, user and organization; *ii) messaging service*, to set up data messaging streams from devices to services via MQTT protocol; *iii) time series service*, to query historical data measured by connected devices via HTTP GET request; and

iv) identity management. Interoperability on the platform level (as well as its offered services) is enabled through the public and well documented APIs.

Kaa. It is a highly flexible, multi-purpose, open-source middleware platform for implementing complete end-to-end IoT solutions, connected applications, and smart products. Kaa offers a set of out-of-the-box enterprise-grade IoT features that can be easily plugged in and used to implement a large majority of the IoT use cases (i.e. agriculture, automotive, healthcare, IoT industry, Smart City and Smart Home, wearable and telecom). The platform features include device management, data collection, configuration management, messaging, and more. The platform is also designed for broad compatibility of connectivity protocols and data management solutions. Kaa provides a Kaa Sandbox, a preconfigured virtual environment that lets the user to create a Kaa-based application. By uploading the schema definition for the device the system allows the automatic generation of a SDK for an application. Kaa is hardware-agnostic because most of the platforms are already pre-integrated, but even if a platform is not listed, through the Kaa SDKs (Java, C++, C and Objective-C) it is easy its integration. It is also transport-agnostic because it allows building applications that work over any type of network connection, either persistent or intermittent. Kaa is distributed pre-integrated with popular data processing system (i.e. Apache Spark, Hadoop, CDAP, Apache Flume). Main features are: *i)* mechanisms for delivery of configurable event messages across connected devices; *ii)* endpoints that perform temporary storage of logs of any predefined structure and implement triggers that initiate periodic logs upload from the endpoint to the server; *iii)* client-side endpoint profile, exposed for the access by Kaa applications; *iv)* server-side endpoint profile controlled by Kaa server users via Administration UI or by other server applications via REST API; *v)* notification system, based on topics, for the delivering of messages from server to subscribed endpoints; *vi)* updates of operational data, such as configuration data, from the Kaa server to endpoints; and, *vii)* support of multi-tenancy and multi-application configuration.

Carriots. It is a scalable Platform-as-a-Service designed for IoT and M2M applications. It gives the possibility to connect any type of devices and hardware, like sensors, gateways, machines, with a web connectivity (e.g. Arduino, Raspberry Pi, Nanode). Any kind of data, in XML and JSON format, is sent through Carriots HTTP RESTful API and stored in their NoSQL storage. Data transfer is secured through Apikeys, checksums and HTTPS. It gives the possibility to build a complete control panel over Carriots REST API in any language. In order to create rules, Carriots uses the if-then-else approach. More complex rules could be developed through the Carriots SDK that relies on the Groovy technology ⁹. Key features of this platform

⁹Groovy is a "dynamic language for the Java Virtual Machine that builds upon the strengths of Java but has additional power features inspired by languages like Python, Ruby or Smalltalk"

are: *i*) remote control, maintenance and interaction with devices; *ii*) interaction with data and devices is actuated through simple if-then-else structures or through more complex rules by using Groovy scripts; *iii*) listeners and rules are both securely and efficiently executed by Carriots SDK; *iv*) integration with other IT systems, creation of export files, pushing data to another database or just using REST API to manage outbound data; *v*) custom alarms and debug logs provide information about the system status and devices interaction; and *vi*) a custom control panel able to manage all Carriots entities and also a user management tool to help managing project and profiling.

macchina.io. macchina.io is an open source software toolkit, created by Guenter Obiltschni, Founder and Lead Developer of the POCO C++ Libraries, for embedded IoT edge and fog computing applications that connect sensors, devices and cloud services. This platform implements a web-enabled, modular and extensible C++ and JavaScript runtime environment and provides easy to use building blocks. Applications can thus talk to various sensors, devices and cloud services, and to process, analyze and filter sensor data locally, at the edge device or within the local network ("fog"). macchina.io combines the power of V8 JavaScript for rapid application development with the power and performance of native C++ code. A unique bridging system and a code generator make it easy to consume C++ services from JavaScript, without the need to manually write glue code. It is distributed with a powerful embedded web application server, providing a flexible module system that makes it easy to build dynamically extensible applications providing rich web-based user interfaces. It is equipped with rich APIs for accessing various sensors and devices, which can be used from both JavaScript and native C++ code and includes HTTP(S) and MQTT clients for connecting to cloud services and other IoT devices. Security is guaranteed using my-devices.net for secure remote management and via Web, SSH and VNC for remote access.

1.2.2 Cross-Domain IoT Platforms

The platforms presented in previous section allow the development of multiple IoT applications that are deployed independently. In most cases there is no easy way to combine data and services from diverse IoT platforms, even when these platforms have been conceived for the same application domain. These difficulties are due from one side to the heterogeneity of the data formats and protocols adopted by the IoT platforms and from the other to the adoption of different semantic representations for the IoT resources, including units of measurement, mathematical constructs, sensor types and properties and more. This makes the so far proposed

Name	SOLUTIONS FOR INTEROPERABILITY	LAYER OF INTEROPERABILITY	BARRIERS FOR INTEROPERABILITY
BEZIRK	Open API, Open Protocol, Open Source	Protocol-level, Interface-level, Device-level	Interoperability only with systems in the local network
Bosch SCP	Data-as-a-Service Data-level	FlyBits platform	Not-known
CSI Piemonte	Open API	Data-level, Interface-level, Service-level	The impossibility of manage lifecycle of smart objects represents a barrier for a complete integration with other platforms
FIWARE	OMA BGSi-9/10, IDAS/IoT Data Edge	Interface-level	Lack of easy-to-use SDKs for IoT Platform and service developers. Lack of semantic expressiveness (to describe/discover data and functions). Lack of maturity (TRL of IoT Discovery around 5)
WordSensing	HTTP(s) (data transport), REST API	Interface-level	No row access to the "actual" things/sensors. Access to the platform will be limited to the Barcelona pilot, for specific clients just during the project.
TIC platform	Open API	Data-level, Service-level	The platform integrates data via open API, but also to a great part data provided by 3rd parties. For these data we have no consent to use the data within interoperable solutions
IFTTT	Open API	Data-level	For creating IFTTT rules, a developer needs to know a service/platform that is connected to IFTTT and IFTTT itself.
Wubby	Adding new applications in Wubby VM	Device-level	Not-known
Xively	Open APIs	Device-level	Not-known
Kaa	REST API	Data-level	Not-known
Carriots	HTTP RESTful API	Device-level	Not-known
macchina.io	Open API	Device-level	Works only with platforms that supports the POCO C++ libraries. The build system currently supports only Linux and OS X

TABLE 1.3: Measures of Interoperability of IoT Platforms (enhanced version of [68])

IoT platforms adapt for the development of vertical applications and provide limited support for more integrated horizontal applications, that are able to combine IoT data and services from multiple IoT platforms.

The first platform that has been conceived for creating a bridge among heterogeneous vertical IoT platform is the OpenIoT Platform that provides basic facilities for the development of cross-domain platform. OpenIoT incorporates an enhanced version of the popular Global Sensor Network (GSN) middleware (<https://lsir.epfl.ch/research/current/gsn/>) and the Linked Sensor Middleware (LSM) projects [85] which enable the collection of data streams from different IoT sensors and devices along with their semantics annotation according to the W3C Semantic Sensor Networks (SSN) ontology (discussed in next chapter) and extension over it. Moreover, In the context of the Internet of Things European platforms Initiatives (IoT-EPI: <http://iot-epi.eu>), eight projects have been financed by the European Commission with the purpose to develop infrastructures among heterogeneous platforms in order to address the interoperability issues at different layers.

In the reminder of the section we will discuss the main features of the OpenIoT (Open Source cloud solution for the Internet of Things) platform and then present the characteristics of the technological-oriented projects among those that received funding: BIG-IoT project (Bridging the Interoperability Gap of the Internet of Things), Biotope (building an IoT Open Innovation Ecosystem for Connected Smart Objects), the INTER-IoT project (Interoperability of Heterogeneous IoT Platforms), and SymBioTe project (Symbiosis of Smart Objects Across IoT Environments).

OpenIoT. This middleware infrastructure is used for implementing and integrating IoT solutions. The OpenIoT infrastructure provides the following: *i*) collecting and processing data from virtually any sensor in the world, including physical devices, sensor processing algorithms, social media processing algorithms and more; *ii*) semantically annotating sensor data, according to the W3C Semantic Sensor Networks (SSN) specifications [126]; *iii*) streaming various sensors data to a cloud computing infrastructure; *iv*) dynamically discovering/querying sensors and their data; *v*) composing and delivering IoT services that comprise data from multiple sensors; *vi*) visualizing IoT data based on mashups (charts, graphs, maps etc.); and, *vii*) optimizing resources within the OpenIoT middleware and cloud infrastructure.

The term "sensor" in OpenIoT refers to any components that can provide observations. OpenIoT facilitates the integration of the above sensors with only minimal effort (i.e. few man days effort) for implementing an appropriate access driver. OpenIoT combines and enhances results from leading edge middleware projects, such as the Global Sensor Networks (GSN) [2] and the Linked Sensor Middleware (LSM) projects [84]. Moreover, it is designed to satisfy different domains, in particular focusing on providing efficient ways to use and manage cloud environments for IoT

entities and resources such as sensors, actuators and smart devices. OpenIoT transports data by using XML and RDF with the standard HTTP protocol. Interoperability of the platform level is enabled through public and well documented APIs.

BIG-IOT. BIG-IOT project will develop an architecture as a foundation for building IoT ecosystems that address the issues of interoperability by adopting: *i*) a common API, *ii*) semantic descriptions of resources and services, as well as *iii*) a marketplace as a nucleus of the ecosystem. This will allow new services by combining data from multiple platforms (e.g., parking information from various smart city platforms). In addition, platforms from multiple domains (e.g. home and city) and regions will be combined, such that applications can utilize all relevant information and work seamlessly across regions (e.g. the same smart parking application works on top of a smart city platform in Berlin, in Barcelona and in London).

Biotope. This project aims at addressing the following objectives: *i*) provide the necessary standardised Open APIs to enable interoperability between today's vertical IoT silos; *ii*) enable new forms of co-creation of services ranging from simple data collection and processing, to intelligent, situation aware and self-adaptive support of everyday work and life; *iii*) establish a robust IoT framework for security, privacy & trust that facilitates the responsible access and ownership of data; *iv*) develop large-scale pilots in smart cities to provide proofs-of-concept of bIoTope enabled SoS ecosystems; *v*) maintain, grow and sustain the socio-technical and business models of bIoTope ecosystems by establishing a governance roadmap for ecosystem evolution. bIoTope technologies enable the publication, consumption and composition of heterogeneous information sources and services from across multiple systems (OpenIoT, FIWARE, city dashboards...). Full advantage is taken of recent IoT standards, notably the O-MI (Open Messaging Interface) and O-DF (Open Data Format) standards, while an "Everything as a Service" design enables rapid development of new IoT systems and reduced development costs. The bIoTope platform enables IoT product and service providers to quickly develop and deploy IoT solutions utilizing diverse information sources, which are easily integrated to compose more advanced and higher value solutions without substantial development costs.

symbIoTe. It aims at providing a simplified IoT application and service development process over interworking IoT platforms. This will be accomplished by: *i*) providing the means to create and manage virtual IoT environments across various IoT platforms; *ii*) implementing high-level APIs -enablers- leveraging such virtual environments to offer specialized services (e.g., localization in indoor spaces or unified access to environmental data gathered from various sources), tailored to the needs of symbIoTe-specific use cases; *iii*) offering the means for creating dynamic and self-configurable smart spaces; and *iv*) implementing a secure interworking protocol

between the platforms in accordance with recommendations from standardization bodies. This will support SMEs and new entrants in the IoT domain to build innovative IoT services within short development life cycles. symbIoTe is built around the concept of virtual IoT environments provisioned over various cloud-based IoT platforms. Virtual IoT environments are an abstraction composed of virtual representations of actual sensors and actuators being exposed by their host platforms to third parties. Of course, a single virtual sensor may emit raw, aggregated or filtered data produced by many sensors residing within a host platform. It needs to be noted that the host platform defines the policies for exposing its virtual sensors to third parties. symbIoTe envisions dynamic and adaptive virtual environments since resource offerings across symbIoTe-enabled IoT platforms are also continuously changing. Its architecture is built around a hierarchical IoT stack (motivated by the OneM2M approach and OpenIoT's VDK) and spans over different IoT platforms. Smart objects are expected to be connected to IoT gateways within the smart spaces which also host various computing and storage resources. The local infrastructure shares the available local resources (connectivity, computing and storage) and is connected to platform services (e.g. resource discovery and management, data analytics) running in the cloud. symbIoTe aims at implementing an Open Source middleware prototype, following an agile-like approach. Developers from all consortium partners will join forces in the implementation of the software components in the aforementioned domains. Regarding licensing, the consortium is discussing on the selection of the appropriate licensing scheme. Initial discussions have indicated that for the licensing of the Application/Cloud domain SW components (i.e., the symbIoTe high level APIs), a "copyleft" license will be selected (most probably the GNU Library or 'Lesser' General Public License version 3.0 (LGPL-3.0)), so that updates, bug fixes and new features are always given back to the Open Source Community. For the middleware components residing at the platforms' side (i.e., at the Cloud or Smart Space domains), the licensing will follow the "non-copyleft" approach (e.g., the BSD 3-Clause "New" or "Revised" License (BSD-3-Clause)).

Inter-IoT. This project is aiming at the design and implementation of an experimentation with, an open cross-layer framework to provide voluntary interoperability among heterogeneous Internet of Things (IoT) platforms. The project is driven by use cases from two domains: (e/m)Health and transportation and logistics in a port environment. The proposal will allow effective and efficient development of adaptive, smart IoT applications and services on top of different heterogeneous IoT platforms, spanning single and/or multiple application domains. The solution adopted by INTER-IoT will include three main products/outcomes:

1. *INTER-LAYER*: methods and tools for providing interoperability among and across each layer (virtual gateways/devices, network, middleware, application services, data and semantics) of IoT platforms. Specifically, they will

propose real/virtual gateways [48] for device-to-device communication, virtual switchers based on SDN for network-to-network interconnection, super middleware for middleware-to-middleware integration, service broker for the orchestration of the service layer and the semantics mediator for data and semantics interoperability [111].

2. *INTER-FW*: a global framework (based on an interoperable meta-architecture and meta-data model) for programming and managing interoperable IoT platforms, including an API to access INTER-LAYER components and allow the creation of an ecosystem of IoT applications and services.
3. *INTER-METH*: an engineering methodology based on CASE (Computer Aided Software Engineering) tool for systematically driving the integration/interconnection of heterogeneous non-interoperable IoT platforms.

1.3 Comparison of IoT Platforms

In the previous section we presented different IoT platforms that are currently available on the market or developed in the context of projects funded by Public Institutions. Some of them are open-source, others are commercial and the code is not publicly available, some of them have been realized in the context of European Projects and sometimes are no-longer maintained. In this section we wish to compare them from different perspectives in order to better understand the current status of middleware softwares in the IoT domain.

The issue of interoperability is mainly addressed at the syntactic level by offering public protocols, interfaces and description of device capabilities. Some of them exploit semantic web concepts even if mainly as research extensions of the systems. For each platform, Table 1.3 shows the approach/solution used for interoperability (e.g. OpenAPI, Open Source, etc.), the level of interoperability (e.g. at data-level, device-level, interface-level, protocol-level, service-level), and the barriers of interoperability that need to be removed. A deeper analysis of the issues of interoperability in the IoT domain will be discussed in Chapter 2 where research approaches for the semantic interoperability are discussed.

Relying on the work presented in [152], we have outlined the main differences among the presented platforms for what concern: persistency, communication, interoperability, identity management, marketplace, semantic descriptions, service discovery and orchestration, deployment, and data streaming and processing. Among the Horizontal Platforms we are considering only OpenIoT because it is the only one currently available. According to the presented analysis, eleven of the IoT platforms provide information communication by XML and/or JSON data model, three IoT platforms support special purpose JSON data format, BEZIRK supports JSON-LD

Name	COMMUNICATION	OPENNESS	SECURITY	PERSISTENCY
BEZIRK	JSON and JSON-LD, TCP/IP (ZeroMQ)	Open API, Open Protocols and Open Source	Encryption	On device storage
Bosch SCP	XML, JSON, HTTP, REST, MQTT	Data-as-a-Service	Spring XD security	MySQL, MongoDB, and DynamoDB
CSI Piemonte	JSON/ODATA, HTTP(s)/MQTT(s)	Open API	OAuth2	Yucca Storage
FIWARE	JSON/HTTP, Orion Context Broker	OMA NGSI-9/10 IDAS/IoT Data Edge	OAuth 2.0	Different depending on GEs
OpenIoT	XML, JSON, RDF	HTTP	Omuth2	OpenIoT RDF store (LSM light)
WordSensing	JSON	Not known	JWT	Not supported
TIC platform	JSON	Open API	Apkey	NoSQL (only internal use)
IFTTT	Not known, depending on supported services	Open API	Two-step verification using user password and phone	Amazon Simple Storage Service (a cloud storage)
Wubby	Anything	Protocol agnostic: MQTT	Not known	MySQL
Xively	JSON+SenML, XML +SenML, CSV (proprietary)	HTTP REST (GET)	Not known	Not known
Kaa	Structured and unstructured data	HTTP REST API	Not known	MongoDB, Couchbase, Cassandra, Hadoop Filesystem
Carriots	XML and JSON	HTTP RESTful API	Apikeys, checksums and HTTPS	NoSQL Data Base
macchina.io	Anything supported by the POCO C++ Libraries	Open API, MQTT	my-devices.net, SSH, HTTPS	SQLite

TABLE 1.4: Comparison of IoT Platforms: Communication, Openness, Security and Persistency (enhanced version of [68])

and CSI Piemonte supports JSON/ODATA. IoT platforms support their functionalities mostly by using HTTP protocol. IoT platforms, which offer message-oriented communication such as Bosch's SCP, CSI Piemonte, Wubby and macchina.io use mostly MQTT for communication. To improve interoperability, the IoT platforms provide programming interfaces, such as BEZIRK, CSI Piemonte, OpenIoT, TIC Platform, IFTTT, Kaa, and macchina.io. Other IoT platforms try to achieve interoperability through an open source approach, such as BEZIRK and OpenIoT. Instead of opening their functionality, Bosch SCP exchanges their information with other systems through a data as service approach. Xively and Carriots allow only the delivery of data over their platform as a platform as a service approach. WorldSensing is currently a closed system and does not offer specific interoperability measures.

The IoT platforms BEZIRK and WorldSensing improve system security by encryption. WorldSensing uses JSON Web Token (JWT)¹⁰ while BEZIRK uses its sphere-based model¹¹, utilizing standard encryption technologies. Moreover, CSI Piemonte, FIWARE, and OpenIoT provide the AAA (Authentication, Authorization, and Accounting) concept by using OAuth2 protocol¹². Bosch SCP supports their security by using the framework Spring XD security whereas Carriots uses Apikeys, Checksums and HTTPS. To store data, Bosch SCP provides MySQL, MongoDB and DynamoDB. Wubby uses MySQL. CSI Piemonte uses the Yucca Storage to save their data. Amazon Simple Storage Service (AWS S3) provides the IFTTT service platform with a cloud storage. WorldSensing manages the real time traffic information and supports no storage. In contrast, OpenIoT uses an RDF store to persist information according to the used semantic models. Kaa uses MongoDB, Couchbase, Cassandra and Hadoop HDFS to store data. Carriots uses a generic NoSQL database while macchina.io stores data through a SQLite database.

Support for semantic descriptions is not wide. Three of the thirteen surveyed IoT platforms support semantic descriptions of their data or services. OpenIoT and FIWARE both support the semantic description of their data by using RDF/OWL [11]. Thereby, OpenIoT uses the SSN ontology and OpenIoT Ontology. BEZIRK also uses semantic descriptions of the information, assuming the use of application or domain specific ontologies. Bosch SCP uses semantics for service discovery, but it does not support semantic description of smart objects. Regarding Service Discovery, ten IoT platforms support service discovery. Three IoT platforms discovery its services based on semantic description. Seven IoT platforms support service discovery through syntactic description of services, whereas the others have no discovery functionality. None of the analyzed IoT platforms support service orchestration.

Cloud based IoT platforms, Bosch SCP, FIWARE, and the TIC Platform use the Docker Container. One benefit of using Docker is the automatic synchronization of the development environment and runtime environment. Four IoT of the thirteen IoT platforms have provided a marketplace. However, the FIWARE marketplace is platform dependent. It allows transactions of the data/services that are developed within FIWARE. IFTTT also has a marketplace, but the details are unknown. Acting as general conclusion, the overview Tables 1.4 and 1.5 show the concepts as columns and the platforms as rows.

1.4 Concluding Remarks

The presented systems do not provide an easy communication with other platforms. The attempts of guarantee the semantic interoperability by exploiting Ontologies are

¹⁰<https://jwt.io/>

¹¹<http://www.sphereproject.org/>

¹²<https://oauth.net/2/>

Name	SEMANTIC CONCEPT	SERVICE DISCOVERY AND ORCHESTRATION	DEPLOYMENT	MARKETPLACE
BEZIRK	Yes, based on domain-specific semantic protocols and ontologies	Semantic based	Not supported	Not supported
Bosch SCP	Semantic is supported. No ontologies	Syntactic level	Docker, provisioner	Not supported
CSI Piemonte	Not supported	Only through user interaction	Not known	Using for free
FIWARE	RDF/OWL	Through different GEs: Syntax based: NGSI-9 Server Semantic based: Sense2Web	Docker GE in FiWare: Sagitta. GE in FiWare: Repository RI	Different GEs: WMarket and WStore
OpenIoT	SSN ontology, OpenIoT ontology	Semantic based: depend on ontology	Not known	Not supported
WordSensing	Not supported	Not supported	Not supported	Not supported
TIC platform	Not known	No Exist	Docker	Not exist
IFTTT	Syntax based: keyword-based	Not known	Not supported	Yes, details unknown
Wubby	Syntax based: ID-based, service based, product based	Not supported	A custom buildbot approach together with some logic in the devices for automatic updates	it exists
Xively	Not known	Syntax based: very limited search criteria	Not known	Not known
Kaa	Not supported	Through Kaa Sandbox	Custom Applications	Not exist
Carriots	Not known	Only through user interaction	Custom Application	Yes, details not known
macchina.io	Not known	Not Supported	Custom Application	in the pro version

TABLE 1.5: Comparison of IoT Platforms: Semantic Concept, Service Discovery and Orchestration, Deployment and Marketplace (enhanced version of [68])

quite limited and there is the need to make available (eventually under specific constraints) the events generated by their sensors and to allow the reception of commands to be executed on their actuators.

In order to face the last requirement, a common approach that can be exploited is the use of context brokers, that is systems able to expose the events generated by the sensors to external platforms by mean of a publish/subscribe communication protocol. In this way, applications interested in receiving events from a platform can subscribe to the channel associated with a sensor and be notified of the occurrence of a new event. Even if this approach does not solve the issue related to the semantics associated to the events that are published on a given channel, it allows to avoid to

know the way in which the events are generated and transmitted within the IoT platforms. External applications can simply consume the generated events. Analogous considerations can be done for sending commands to actuators that are managed by an IoT platform. We remark that both in the case of reading events from sensors and sending commands to actuators, the need arises to guarantee these services only to authorized applications and thus to exploit security and privacy techniques associated with the context brokers.

In the thesis we consider context brokers that make available to our system the events generated by cross-domain IoT sensors. The sensors and the schema of the events generated by the sensors are semantically annotated according to the domain ontology adopted in a given context of use in order to guarantee their semantics and correct processing. As context broker we will adopt Apache Kafka. However, also other context brokers can be easily integrated in our architecture.

Chapter 2

Semantic Interoperability in the IoT Context

This chapter describes recent projects and tools in sensor network research area that have focused their efforts on how to integrate data generated in raw and heterogeneous formats by means of a common semantics able to describe their meanings, the lack of which imposes barriers to interoperability among heterogeneous sensors. In the IoT domain, users are primarily interested in understanding the meaning of combined streams that can lead to the detection of significant events instead of raw data flows. Nevertheless, sensors provide raw data that do not contain any additional description or metadata and require specialized knowledge and manual effort for their meaningful combination. This chapter aims at tackling this problem by describing several solutions able to integrate sensor data through Semantic Web technologies in order to publish data streams in an enriched and standardized way, so that they can be accessed and consumed by external applications.

One of the key factors of these strategies is the possibility to describe the semantics of the sensor data taking into account a specific Domain Ontology, that is an ontology developed by the domain experts that starting from standard IoT ontologies include concepts and relationships that are usually adopted in the IoT domain. In this way, the semantics adopted in a given domain is clear and well specified. The use of a domain-based vocabulary in an ontology-driven approach for the explication of implicit and hidden knowledge is a possible approach to overcome the problem of semantic heterogeneity. It allows information exchange such that the data meaning can be automatically interpreted and useful elaborated by the receiver.

The chapter is structured as follows. Section 2.1 provides a description of the interoperability problem specifically oriented toward semantic interoperability in IoT field, then Section 2.2 describes some of the most used ontologies in this domain. Section 2.3 provides an overview of semantic description techniques used for the explicit presentation of the semantics of the data sources and Section 2.4 focuses on arguing how to use semantic models for describing data sources in terms of the concepts and relationships defined by a Domain Ontology.

2.1 The Issues of Interoperability

There is no single definition of the word *interoperability* because it has different meanings in different contexts. In some cases, interoperability is defined as the ability to describe the extent to which systems and devices can exchange data and interpret the shared data. For two systems to be interoperable, they must be able to exchange data and subsequently present them in a way that can be understood by a user¹. Other sources² define interoperability as the ability of a computer system to run application programs from different vendors and to interact with other computers across local or wide-area networks regardless of their physical architecture and operating systems. Interoperability is feasible through hardware and software components that conform to open standards such as those used for internet.

The National Alliance for Health Information Technology (Alliance) introduces the levels of interoperability and adopts the use of four categories as proposed by the Center for Information Technology Leadership [67]. At the most basic level is the exchange of data in non-electronic formats - pieces of paper and phone calls, for example. The second level are data that can be transmitted electronically, such as via fax or e-mail. The third level represents another leap: data that machines can organize, such as labeled documents and images. The fourth level, the highest level of interoperability can be achieved when machines can interpret data and perform automatic functions, for example, in the context of Health, integrating lab results from one facility into the electronic health record (EHR), or electronic medical record (EMR) system of another facility. Achieving machine interpretable data requires standards for the acquisition, storage and transmission of data, and involves careful attention to data integrity, privacy, and security. ETSI divides interoperability in different categories (Figure 2.1) [142]:

- *Technical interoperability*: it refers with hardware/software components, systems and platforms that communicate through machine-to-machine protocols.
- *Syntactical interoperability*: it is usually associated with data format. The communication protocols need to transmit messages with a well-defined syntax and encoding. Contents can be represented using high-level formats such as HTML, XML or JSON.
- *Semantic interoperability*: it is everything that concern the meaning of the content and its interpretation by the machines.
- *Organizational interoperability*: it is the ability of organizations to effectively communicate and transfer (meaningful) data (information) even though they may use a variety of information systems over different infrastructures.

¹<http://www.himss.org/library/interoperability-standards/what-is-interoperability>

²<http://www.businessdictionary.com/definition/interoperability.html>

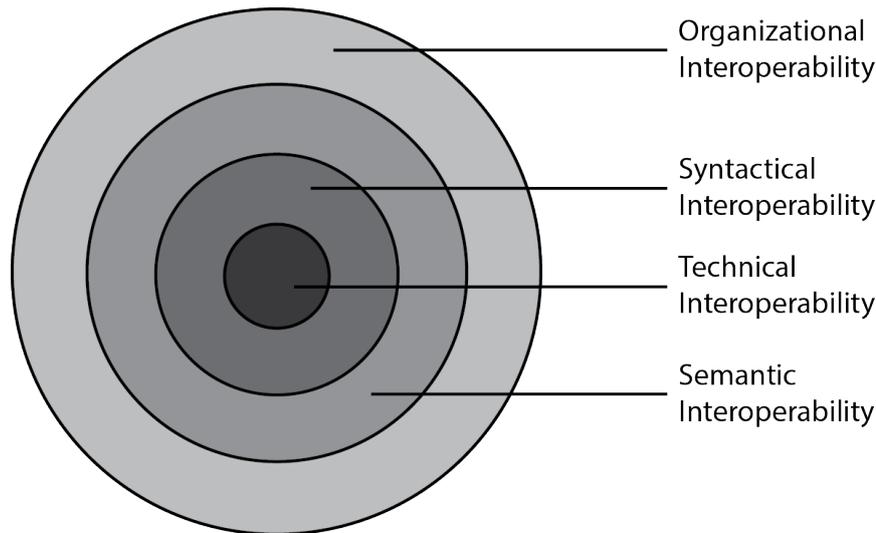


FIGURE 2.1: Different levels of interoperability

Interoperability has been promoted through different projects and approaches [70]. Starting from the LISI (Levels of Information System Interoperability) approach in 1997, a project developed by the C4ISR architecture working group (AWG) in order to provide maturity level to US Department of Defense (DoD) [87], the IoT Forums are working to develop a common model that can ensure interoperability among smart objects.

2.1.1 IoT Interoperability

Interoperability in the Internet of Things is now a hot topic and it is critical for emerging services and applications. Devices on the Internet of Things (IoT) are connected to the Internet, are equipped with identifying, sensing and processing capabilities, they are constantly connected to each other, also to services, and they collaborate or interoperate in order to discover their context or to create some values. To reach the full potential of the IoT, however, it is not sufficient for things to be connected to the Internet; they also need to be found, accessed, managed and potentially linked to other things. To enable this interaction, a higher degree of interoperability is necessary that goes beyond simple protocol interoperability as provided by the Internet [19]. This means that IoT requires standards to enable the development of horizontal platforms on top of cross-platforms that are communicable, operable and programmable across devices, regardless of make, model, manufacturer or industry. The hope is that connectivity between people, processes and things works no matter what screen type, browser or hardware is used. The reality, however, is that the IoT is fragmented and lacks interoperability. According to [37], the fragmentation is mainly due to the following reasons:

- different Original Equipment Manufacturers (OEMs): devices or equipment that are not made by the same manufacturer cannot integrate.

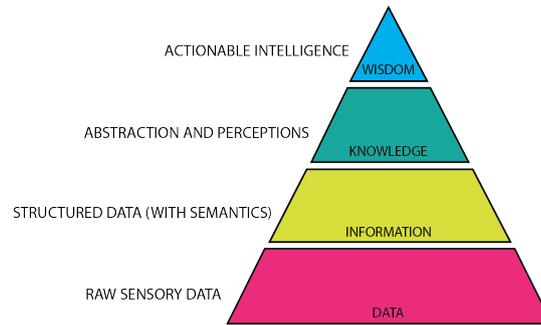


FIGURE 2.2: "Knowledge Hierarchy" in the context of IoT. Taken from [121]

- Different Operating Systems (OSs): inability to run on the same OS.
- Different versions or times of purchase: devices that were not made or purchased at the same time.
- Different/incommunicable types of connectors or connectivity frameworks (e.g. devices).
- Different/inconsistent communication protocol standards (i.e. rules).
- Lack of programmability needed to connect in the first place.

The understanding of a situation (such as the comprehension of a context) enables services and applications to make intelligent decisions and to respond to the dynamics of their environment. As discussed in the previous chapter, sensors data are usually heterogeneous (temperature, light, sound, video, etc.) and different devices can produce data with different quality that depends on the time and location where data is acquired [11]. This data can be analyzed, transformed, enriched in order to give us better understanding about our physical world and in order to create valuable and meaningful information. This data transformation process can be better illustrated using an adapted version (for IoT) of the "knowledge hierarchy" [121].

As show in Figure 2.2 the lower layer refers to the huge amount of data produced by sensors and devices. This data is characterized by its low quality, the presence of different outliers and with values that need to be cleaned, integrated and contextualized in order to be processed. The layer above helps creating structured and machine readable information from the raw data of various forms to enhance interoperability. However, applications and services require high-level abstractions and perceptions that provide human and machine-understandable meanings and insights of the underlying data. The high-level abstractions and perceptions then can be transformed to actionable intelligence (wisdom) with domain and background knowledge to exploit the full potential of IoT and create end-to-end solutions [11].

Open Internet Consortium (OIC) is currently focusing on the IoT interoperability to define specifications, integration of billions of smart objects, and scalability issues [79]. Low cost interoperability among smart objects is an important factor for

Smart Cities. For example, smart mesh backbone gateways (WRT54GL by Linksys³ and net5501 by Soekris Engineering Inc.⁴) are IoT gateways that try to solve the communication gap between field control/sensor nodes and customer's cloud, enabling field data to be harnessed for manufacturing process optimization, remote management, and preventive maintenance. They are developed for smart cities which provide low cost interoperability. The Grid-Wise Architecture Council (GWAC) mission is to enable interoperability among the objects that interact with the electric power system. The GWAC introduced a context setting framework which identifies interoperability issues [64].

2.1.2 Semantic Interoperability

The overall challenge in interoperability is first to ensure technical interoperability from technologies to deliver a mass of information and then complementary challenges are for the information to be understood and processed. Interoperability can be solved if communicating smart objects are semantically interoperable [53]. IoT device's semantics details, interpretation, and exchange of information must be developed in order to remove semantics conflicts in interoperability.

Semantic interoperability is the exchange of information with meaningful and understandable meaning [4, 150]. It is achieved when interacting systems attribute the same meaning to data exchanged, ensuring consistency of the data through the systems regardless of individual data format. Semantic conflicts occur whenever two contexts do not use the same interpretation of the information. Goh identifies three main causes for semantic heterogeneity [56]:

- Confounding conflicts occur when information items seem to have the same meaning, but differ in reality, e.g. owing to different temporal contexts.
- Scaling conflicts occur when different reference systems are used to measure a value. Examples are different currencies.
- Naming conflicts occur when naming schemes of information differ significantly. A frequent phenomenon is the presence of homonyms and synonyms.

The use of shared vocabularies either in a schema form and/or in an ontology-driven approach for the explication of implicit and hidden knowledge is a possible approach to overcome the problem of semantic heterogeneity. Interoperability can be seen as a key application of ontologies, and many ontology-based approaches to information integration to achieve interoperability have been developed [141, 145].

³<http://www.linksys.com/us/support-product?pid=01t80000003KOkNAAW>

⁴<http://soekris.com/products/eol-products/net5501.html>

2.2 Ontologies for IoT

As reported in [137] and [53] it is reasonable to believe that semantic technologies, based on the use of ontologies, have the best chance to facilitate interoperability among the *things*, as well as across the IoT platforms. Ontologies can be used for semantic annotation, access management and for discovering resources in the IoT. As a result, the use of shared ontologies for common interpretation of information and data is the best pathway to achieve semantic interoperability. It allows information exchange such that the data meaning can be automatically interpreted and useful elaborated by the receiver.

The introduction of metadata to describe the contents and context of data to facilitate its discovery, understanding and (re)usability is a key solution. Metadata is about reducing the separation between semantics and values by ensuring that data is provided with context and description. Taxonomies are often build on metadata using parent-child relationship, and are used to describe the organization of terms within a specific domain. Ontologies further extend this concept to capture relationship capable of supporting richer operations and more advanced levels of reasoning. Practical use of semantic methods and tools requires formulation/existence of explicitly expressed ontologies, represented using one of ontology languages (currently RDF(S) or OWL).

Many ontologies for the IoT has been developed. Most of them focus on representing specific properties and characteristics. For example PROV-O provenance ontology⁵, LinkedGeoData⁶, WGS84 geo-ontologies⁷, LSM linked sensor middleware ontology⁸, W3C Time Ontology⁹, TimeLine Ontology¹⁰. Most existing ontologies, capturing the IoT, were developed within individual research projects and, as a consequence, they typically are in prototype stage, often incomplete and sometimes abandoned. The lack of a clear and widely adopted ontology in different application contexts is one of the reason for which it is really difficult to create horizontal applications. As explained later, notable exception due to their usage in real contexts of use grounded on Smart Cities projects are *Km4City* an the *SAO ontology*. Some other ontologies aim at capturing further and specific information about sensor capabilities, performance, usage conditions and enable contextual data discovery. The most notable ontology is the *Semantic Sensor Network ontology* [126], that is one of the most used in the IoT context and could be eligible to become a standard. On October 2017 a new extension of this ontology, called *SOSA* (Sensor, Observation, Sample, and Actuator) [63], has been introduced. A lightweight instantiation of the SSN ontology

⁵<https://www.w3.org/TR/prov-o/>

⁶<http://linkedgeodata.org/OSM>

⁷<https://www.w3.org/2003/01/geo/9>

⁸<http://open-platforms.eu/library/deri-lsm/>

⁹<https://www.w3.org/TR/owl-time/>

¹⁰<http://motools.sourceforge.net/timeline/timeline.html>

has been proposed, *IoT-Lite* ontology [15], that provides a general IoT knowledge model intended to limit processing time of ontologically demarcated resources.

In the reminder of the section we discuss in more details the *Km4City* and the *SAO* ontologies as general purpose ontologies that allows to model IoT devices and resources. Then, we present the *SSN* and *IoT-Lite* ontologies that have been specifically tailored for the IoT context.

2.2.1 Km4City

In the field of open data for Smart Cities a large work has been done by Public Administrations (PAs) on producing data. Open data coming from PA contains typically statistic information about the city (such as data on the population, accidents, flooding, votes, administrations, energy consumption, presences on museums, etc.), location of point of interests on the territory (including, museums, tourism attractions, restaurants, shops, hotels, etc.), major GOV services, ambient data, weather status and forecast, changes in traffic rules for maintenance interventions, etc. A relevant role is covered in city also by private data coming from mobility and transport such as those created by Intelligent Transportation Systems, ITS, for bus management, and solutions for managing and controlling parking areas, car and bike sharing, car flow, good delivering services, accesses on Restricted Traffic Zone (RTZ) etc. Both open and private data may include real time data such as the traffic flow measure, position of vehicles (buses, car/bike sharing, taxi, garbage collectors, delivering services, etc.), railway and train status, park areas status, and Bluetooth tracking systems for monitoring movements of cellular phones, ambient sensors, and TV cameras streams for security [12]. This information is typically not semantically interoperable and a Smart City ontology is not yet standardized. Many research efforts are needed to identify models that can easily support the data reconciliation, the management of their complexity and the capability of reasoning on them.

In order to solve the above described problems and provide a unique point of access for interoperable data of a city metropolitan area, the DISIT Lab (Distributed Systems and Internet Technologies Lab) of the Department of Information Engineering of the University of Florence realized an ontological model called **Km4City**¹¹ and well formalized and open grounded on ontology standards. This project gathers information and aims at interconnecting with many different sources, such as various portals of the Tuscan region (*MIIC*, *Muoversi in Toscana*, *Osservatorio dei Trasporti*), open data provided by individual municipalities (mainly Florence). The *km4City* ontology reuses:

- *dcterms*,¹² a set of properties and classes maintained by the Dublin Core Metadata Initiative;

¹¹<http://www.disit.org/km4city>

¹²<http://dublincore.org/documents/dcmi-terms/>

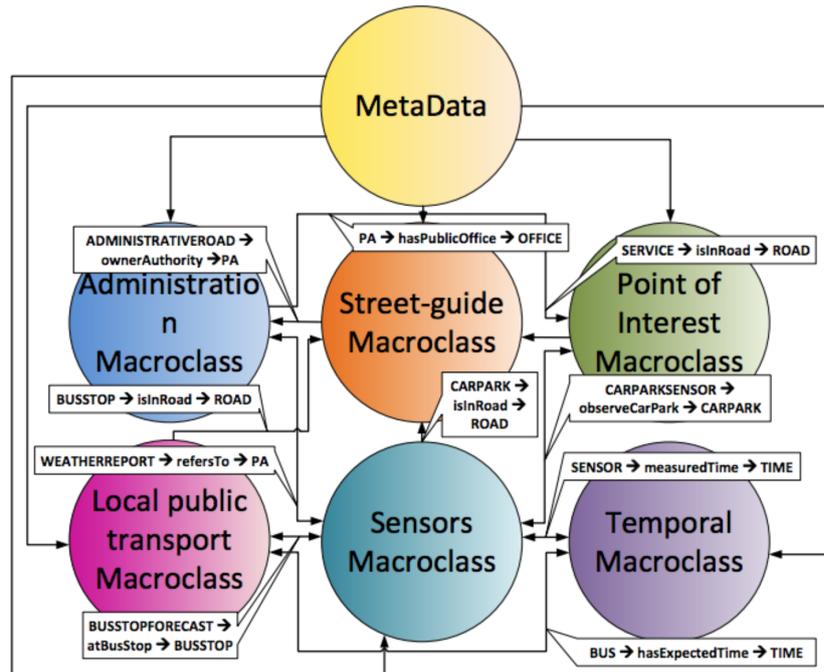


FIGURE 2.3: Ontology Macro-Classes and their connections. Taken from [12]

- *foaf*,¹³ a machine-readable ontology describing persons, their activities and their relations to other people and objects;
- *schema.org*,¹⁴ is a collaborative vocabulary with a mission to create, maintain, and promote schemas for structured data on the Internet; and
- *wgs84_pos*,¹⁵ a vocabulary for representing latitude and longitude coordinates.

The integrated ontological model proposed by Km4City presents seven main areas of macroclasses, as shown in Figure 2.3.

- *Administration macroclass*: it is structured in order to represent the Italian public administration hierarchy in which each region is divided into several provinces, within which the territory is divided into municipalities. To represent this situation the km4City ontology introduces, as main class of administration macroclass, the class PA, which is defined as subclass of *foaf:Organization*. Restriction on some ObjectProperties bring to the definition of three subclasses: Region, Province and Municipality. For example, Region is defined as a restriction of PA on ObjectProperty *hasProvince*, so that only the PA that holds provinces can be classified as regions.
- *Street-guide macroclass*: it is used to represent the entire road system of Tuscany and is formed by entities as Road, Node, RoadElement, AdministrativeRoad,

¹³<http://www.foaf-project.org/>

¹⁴<http://schema.org/>

¹⁵<https://www.w3.org/2003/01/geo/>

Milestone, streetNumber, RoadLink, Junction, Entry, and EntryRule Manoeuvre. These entities have been modelled into the Km4City ontology by choosing, as the main class, the RoadElement class.

- *Point of interest macroclass*: it allows to represent services to the citizens, points of interest, business activities, tourist attractions and is represented by a pair of coordinates and by a category (Accommodation, GovernmentOffice, TourismService, TransferService, CulturalActivity, FinancialService, Shopping, Healthcare, Education, Entertainment, Emergency and WineAndFood).
- *Public transport macroclass*: it includes information relating to public road and railway transports. For what concern road public transports it is organized in lots, each composed by a bus/tram line. Each line includes at least one ascendant direction ride and one descendant direction ride, both identified by a code. Every ride is scheduled to drive along a specific path, called route. A route is a series of road segments delimited by subsequent bus stops. Relating to rail transports, a railway line is composed by a number of railway elements, a railway direction and a railway section. At the end or the beginning of rail elements there are train stations or cargo terminals.
- *Sensor macroclass*: it consists of four parts related to car parks sensors, weather sensors, traffic sensors installed along roads/rails and to AVM/kit systems installed on buses, cars and/or bikes. The first part is focused on the real time data related to parking status. In each status report, there are information about the number of free and occupied parking spaces, for the main car parks. The weather sensors produce real time data concerning the weather forecast once or twice a day. Every report contains forecast for five days. The traffic sensors, are placed along the roads of the region, and produce real-time data, different measures and assessment related to traffic situation. The AVM (Automatic Vehicle Monitoring) systems part concerns the sensors systems installed on most of buses, which, at intervals of few minutes, send a report to the management center. They provide information about: the last stop performed, current GPS coordinates of the vehicle, the identifiers of vehicle and of the line, a list of upcoming stops with the planned passage time.
- *Temporal macroclass*: it is based on the Time ontology [43] as it has been used into OSIM ontology [101]. It requires the integration of the concept of time as it is really relevant the capability to compute intervals between timestamps.
- *Metadata macroclass*: it is used to keep track of the status and descriptors associated with the various ingested dataset. To the various graph identified within the ontology is assigned a name (i.e., an identifier) by means of Sesame¹⁶.

¹⁶www.openrdf.org

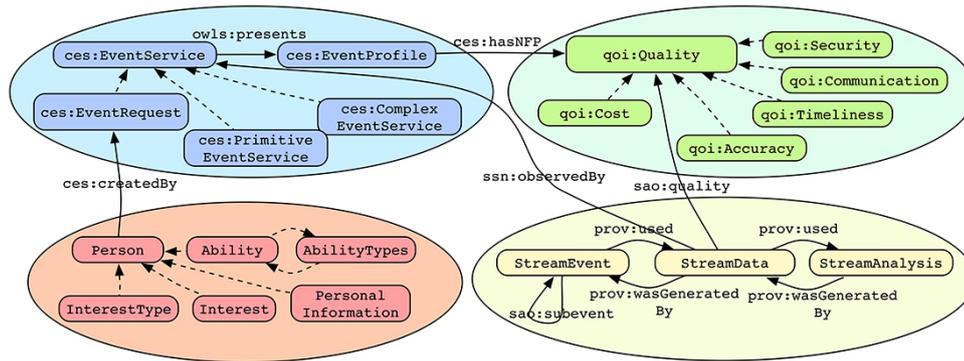


FIGURE 2.4: Overview of the SAO Ontology modules. Taken from <http://iot.ee.surrey.ac.uk/citypulse/ontologies/sao/sao>

2.2.2 Stream Annotation Ontology (SAO)

The Stream Annotation Ontology (SAO)¹⁷ is a lightweight semantic model, which is built on top of well-known models to represent IoT data streams. It has been developed within the realm of CityPulse project¹⁸ that contains four main modules, namely Stream Annotation Ontology (SAO), Quality, Complex Event Ontology, and User Profiles information models [77]. Figure 2.4 shows an overview of the proposed information model.

The SAO module can be used to express the features of stream data. It allows publishing content-derived data about IoT streams and provides concepts such as `sao:StreamData`, `sao:Segment`, `sao:SegmentAnalysis` on top of the TimeLine¹⁹ and IoTest models. Using the SAO module, it is possible to describe a data stream and a timeline instance to link the segment description with the time extent of a temporal entity representing the data stream. Thus, it allows to express a stream data as a time interval on the universal timeline, and also to relate such an interval with the corresponding interval on the discrete timeline along with its discrete sampling rate.

The Quality module is used to represent the quality of information for data streams in smart cities. The Quality itself has five categories with subcategories to describe the attributes of the annotated data stream regarding its quality. In addition, it provides a concept of trustworthiness for data sources. This concept is realized by two additional object properties `hasProvenance` and `hasReputation` to link the StreamData with an Agent as the owner and its Reputation.

The Complex Event Ontology module is used to define, detect and react to complex events. It defines a Complex Event Service (CES) that makes part of the overall event-driven service oriented architecture. The CES is strictly event driven: it

¹⁷<http://iot.ee.surrey.ac.uk/citypulse/ontologies/sao/sao>

¹⁸<http://www.ict-citypulse.eu/>

¹⁹Timeline Ontology extends OWL-Time with various timelines (e.g. universal or discrete), temporal concepts, such as instants and intervals, and interval relationships. Available at: <http://motools.sourceforge.net/timeline/timeline.html>

catches events and triggers CES when it detects them based on their definition and member events occurrences.

The User Profile module is used to represent users' information for applications in Smart Cities. It describes the users with three main concepts: personal information, interest and ability. Interest and Ability are clustered in InterestType and AbilityType respectively.

2.2.3 W3C Semantic Sensor Network Ontology (SSN)

W3C Semantic Sensor Network ontology (SSN) [31, 126] was developed as a joint effort of several research organizations and it is one of the most used in the IoT context and for the semantic description of sensors. Some previously developed ontologies contributed to development of the W3C SSN ontology:

- CSIRO Sensor Ontology²⁰: it was an early attempt for the development of a generic ontology for describing functional, physical and measurement aspects of sensors. It was created at the Commonwealth Scientific and Industrial Research Organization (CSIRO), Australia. Its main classes include sensors, features, operations, results, processes, inputs and outputs, accuracy, resolution, abstract and physical properties, and metadata links.
- SWAMO Ontology²¹: the aim of the SWAMO project was to use collaborative, distributed set of intelligent agents for supervising and conducting autonomous mission operations. SWAMO ontology enables automated decision making and responses to the sensor Web environment. One of its advantages was compatibility with the Open Geospatial Consortium (OGC) standards, enabling geo-data consumption and exchange.
- MMI Device Ontology²²: an extensible ontology of marine devices (hence, an ontology that is slightly more "domain-specific" than others) that integrates with models of sensor descriptions. Its main classes include component, system, process, platform, device, sensor, and sampler.
- SEEK Extensible Observation Ontology (OBOE²³): it is a suite of ontologies for modeling and representing scientific observations. It can express a wide range of measurement types, includes a mechanism for specifying measurement context, and has the ability to specify the type of entity being measured. In this way it is focused more on the results produced by sensors than sensors themselves.

²⁰<http://www.w3.org/2005/Incubator/ssn/wiki/SensorOntology2009>

²¹http://www.w3.org/2005/Incubator/ssn/wiki/Review_of_Sensor_and_Observations_Ontologies#SWAMO

²²<https://marinemetadata.org/community/teams/ontdevices>

²³<https://semtools.ecoinformatics.org/oboe>

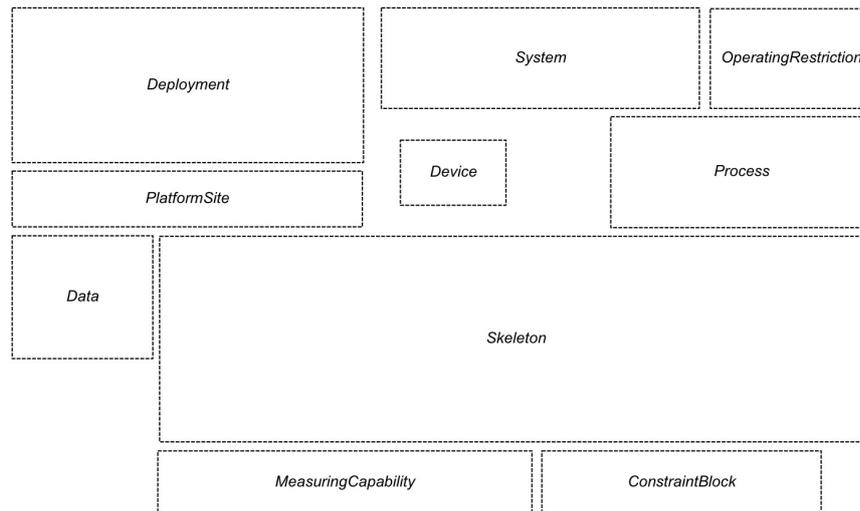


FIGURE 2.5: Overview of the Semantic Sensor Network ontology modules. Taken from [83]

- OGC SensorML standard: it provides a robust and semantically-tied means of defining processes and processing components associated with the measurement and post-measurement transformation of observations.

The Semantic Sensor Web (SSW) proposes the description of sensor data with semantic metadata [132] able to specify the capabilities of sensors, the measurement processes and the resultant observations. This approach uses the current OGC (Open Geospatial Consortium) and SWE (Sensor Web Enablement) [22] specifications and attempts to extend them with semantic web technologies to provide enhanced descriptions to facilitate access to sensor data. W3C Semantic Sensor Networks (SSN) Incubator Group [126] worked on developing an ontology for describing sensors. The core concepts and relations of the SSN ontology concern the description of sensors, features, properties, observations, and systems. Then, measuring capabilities, operating and survival restrictions, and deployments were added in turn. More specifically, the SSN consists of 10 conceptual modules (Deployment, System, OperatingRestriction, PlatformSite, Device, Process, Data, SSOPlatform, MeasuringCapability, ConstraintBlock) (Figure 2.5) which contains 41 concepts and 39 object properties (Figure 2.6). It directly inherits 11 concepts and 14 object properties from the top-level DOLCE-UltraLite ontology²⁴ and the core of the Stimulus-Sensor-Observation (SSO) ontology design pattern [126]. The W3C SSN ontology has been widely used, and both extended and specialized. Among the notable extensions are the Wireless Sensor Networks ontology (WSSN) [13], and Sensor Cloud Ontology (SCO) [100]. The specializations include the AEMET meteorological ontology [7], atmosphere observation ontology SWROAO [147], flood prediction ontology SemSor-Grid4Env [57], and Stream Annotation Ontology SAO²⁵ [77]. In order to improve the

²⁴http://www.ontologydesignpatterns.org/wiki/Ontology:DOLCE+DnS_Ultralite

²⁵<http://iot.ee.surrey.ac.uk/citypulse/ontologies/sao/sao>

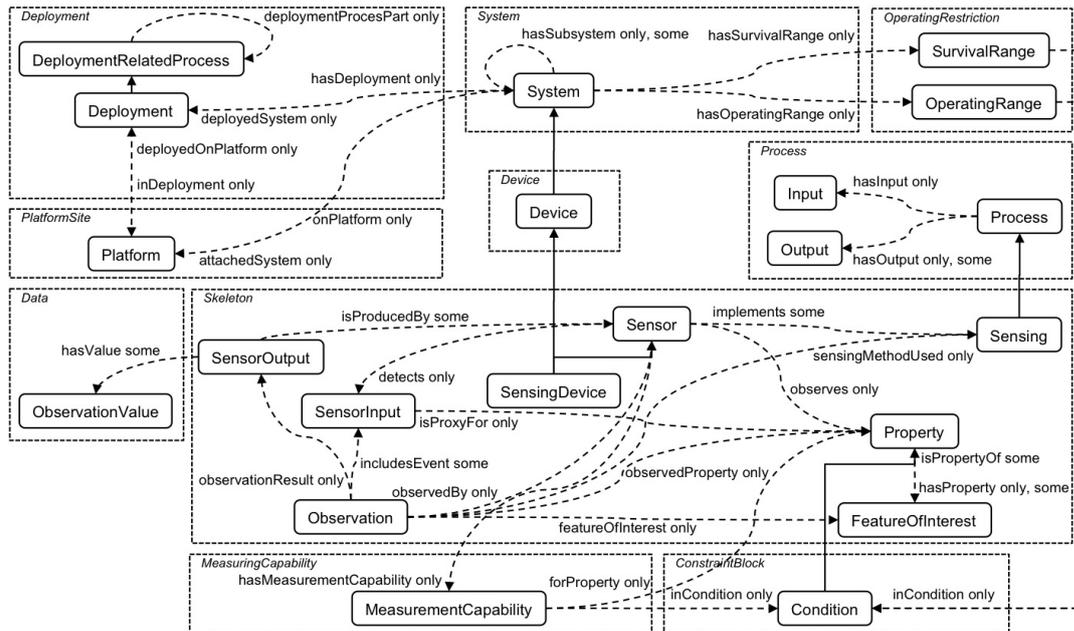


FIGURE 2.6: Overview of the Semantic Sensor Network ontology classes and properties. Taken from [31]

capability of the SSN ontology, especially for what concern the new aspects of sensing (such as actuators), a new extension has been introduced, on October 2017. The new SSN ontology has been designed to provide a flexible but coherent perspective for representing the entities, relations, and activities involved in sensing, sampling, and actuation. The main innovation of this generation of SSN has been the introduction of the Sensor, Observation, Sample, and Actuator (SOSA) ontology, which provides a lightweight core for SSN (see Figure 2.7). SOSA aims at broadening the target audience and application areas that can make use of Semantic Web ontologies. Other SSN modules add additional elements, additional ontological commitments, and/or clarify the alignment of SSN with other ontologies [63].

With their different scope and different degrees of axiomatization, SSN and SOSA are able to support a wide range of applications and use cases, including satellite imagery, large-scale scientific monitoring, industrial and household infrastructures, social sensing, citizen science, observation-driven ontology engineering, and the Web of Things. An exhaustive *Mapping Table* for a comparison of SOSA with the precedent models and ontologies can be found at [144].

2.2.4 IoT-Lite

While SSN ontology describes, in more detail, sensors and observations of data streams by merging sensor-focused, observation-focused and system-focused views,

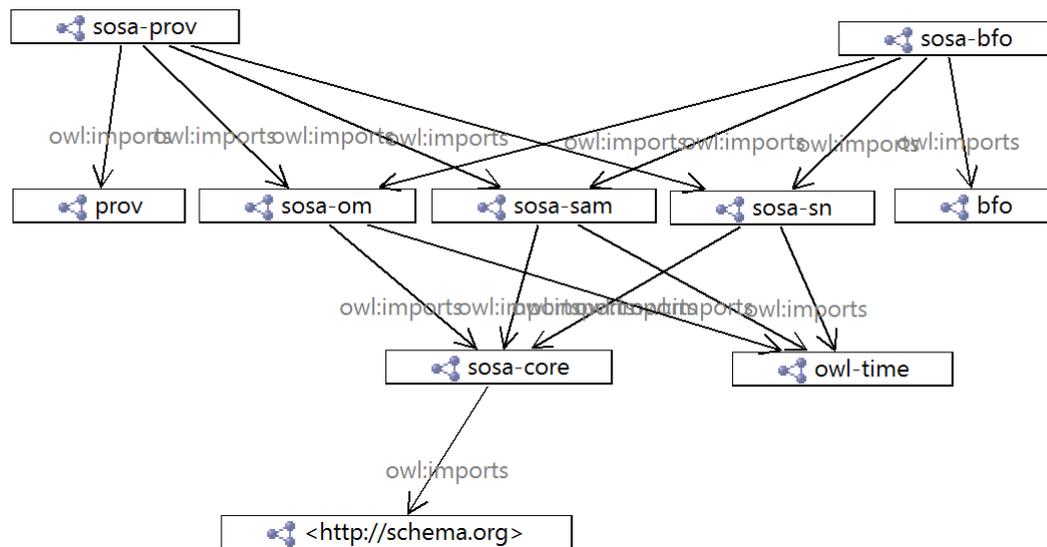


FIGURE 2.7: Dependencies of the SOSA ontology. Taken from https://www.w3.org/2015/spatial/wiki/SOSA_Ontology

IoT-Lite ontology²⁶ aims at defining a lightweight ontology to represent IoT resources, entities and services. The lightweight allows the representation and use of IoT platforms without consuming excessive processing time when querying the ontology. However it is also a meta ontology that can be extended in order to represent IoT concepts in a more detailed way in different domains.

As shown in Figure 2.8 the main purpose of the IoT-Lite ontology is to define only the most-used terms when searching for IoT concept. Concepts are described in three classes: *entity or objects*, *device or resources* and *services*, namely `iot-lite:Object`, `ssn:Device` and `iot-lite:Service`. The interrelations between these three concepts are also well-known relationships, that is, an object (or entity) `iot-lite:Object` has an attribute `iot-lite:Attribute` which is associated with a device (or resource) `iot-lite:Device`, which is exposed by a service `iot-lite:Service`. In order to provide responses to the standard queries, the rest of the ontology has been built around these three concepts, adding the necessary concepts and relationships [15]. IoT devices are classified into, although not restricted to, three classes: *sensing devices*, *actuating devices* and *tag devices* (`ssn:Sensor`, `iot-lite:Actuator`, `iot-lite:Tag`). IoT-Lite is focused on sensing, although it has a high level concept on actuation that allows any future extension on this area. Services are described with a coverage. This coverage represents the 2D-spatial covered by the IoT device and it can be a point or an area (such as circle, rectangle or polygon). IoT-Lite ontology is created to be used with a common quantity taxonomy, *qu-taxo*, to describe the *Units* and *QuantityKind* that IoT devices can measure. This taxonomy is represented by individuals in the ontology and is based on well-known taxonomies such as *qu* and *qudt*.

²⁶<http://www.w3.org/Submission/iot-lite/>

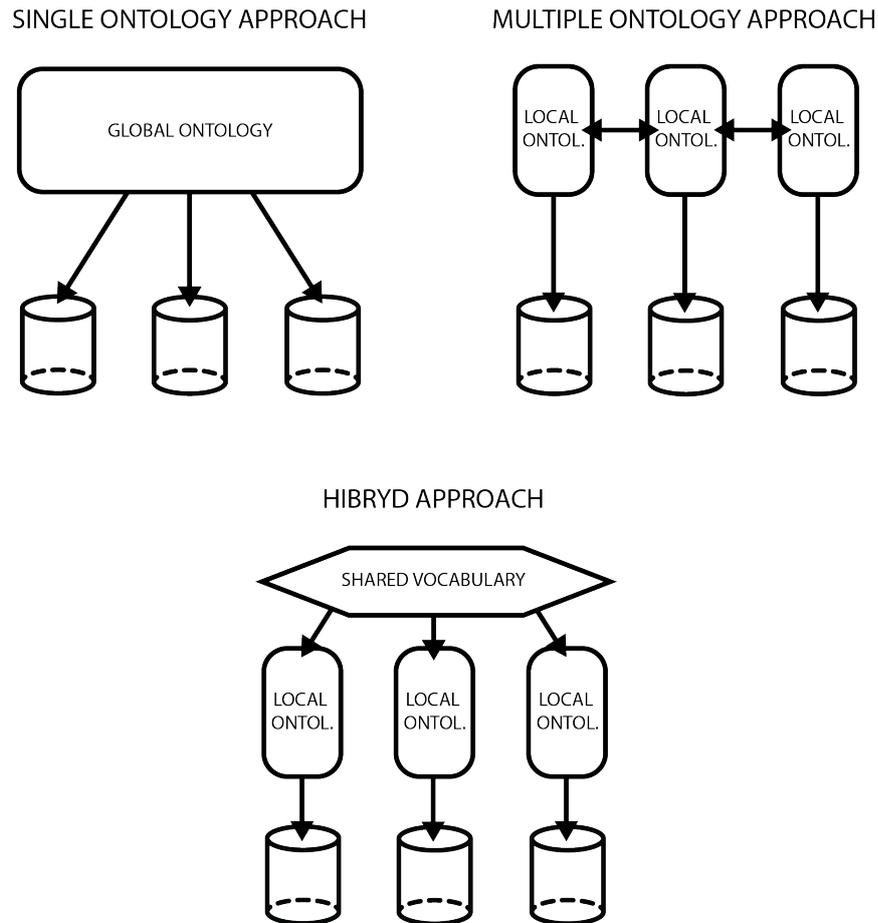


FIGURE 2.9: The three possible ways for using ontologies for content explication

Multiple Ontology approach. In this approach each information source is described by its own ontology. It could be represented as a single ontology approach for each source. OBSERVER [95] is an example of this approach. The difference is that we cannot assume that the different source ontology share the same vocabulary. The advantage of this approach is that no common and minimal ontology commitment about one global ontology is needed [59]. This simplifies the change, the modification, the integration in one information source or the adding and removing of sources. However the lack of a common vocabulary needs to introduce a formalism to define the inter-ontology mapping. The inter-ontology mapping identifies semantically corresponding terms of different source ontologies, e.g. which terms are semantically equal or similar. However, the mapping also has to consider different views on a domain e.g. different aggregation and granularity of the ontology concepts. We are aware that in practice the inter-ontology mapping is very difficult to define, because of the many semantic heterogeneity problems which may occur.

Hybrid approach. In the Hybrid approach every sources is described by its own ontology but, in order to compare source ontologies to each other, a global shared

vocabulary is introduced between the source ontologies [56, 124]. The shared vocabulary contains basic terms (the primitives) of a domain. In order to build complex terms of a source ontologies the primitives are combined by some operators. The interesting point, introduced by this approach, is how the terms of the source ontology are described by the primitives of the shared vocabulary. It could be simply an attribute value vector, or combining the primitive terms from a shared vocabulary and annotated by a label indicating the semantics of the information. In some cases, the shared vocabulary is an ontology, which covers all possible refinements. E.g. the general ontology defines the attribute value ranges of its concepts. A source ontology is one (partial) refinement of the general ontology, e.g. restricts the value range of some attributes. Since the source ontologies only use the vocabulary of the general ontology, they remain comparable. The advantage of this solution is that new sources can be easily added without the modification of mappings in the shared vocabulary. The drawback of hybrid approaches however is that existing ontologies cannot be reused easily, but they have to be re-developed from scratch, because all source ontologies have to refer to the shared vocabulary. Examples of this approach are COIN [56], MECOTA [124] and BUSTER [138].

2.4 Mapping Techniques

In the literature we can find two different approaches to map concepts belonging to different ontologies: *schema matching* and *ontology matching*. Both problems have been widely investigated in the literature and a number of approaches and tools have been proposed both in the area of data and knowledge management [28]. Regarding the first problem, an exhaustive survey has been proposed by [112] and Figure 2.10 shows part of the proposed classification scheme together with some sample approaches.

The implementation of a schema matching may require the use of match algorithms or matchers that depend on the application domain and schema types. The realization can be further categorized in: *i*) individual matchers, each of which computes a mapping based on a single matching criterion; and *ii*) combined matchers of individual matchers, either by using multiple matching criteria (e.g., name and type equality) within an integrated hybrid matcher or by combining multiple match results produced by different match algorithms within a composite matcher. For individual matchers, we consider the following classification criteria:

- *Instance vs schema*: matching approaches can consider instance data or only schema-level information.
- *Element vs structure matching*: match can be realized for individual schema elements (e.g. attributes) or combinations of elements (e.g. complex structures).

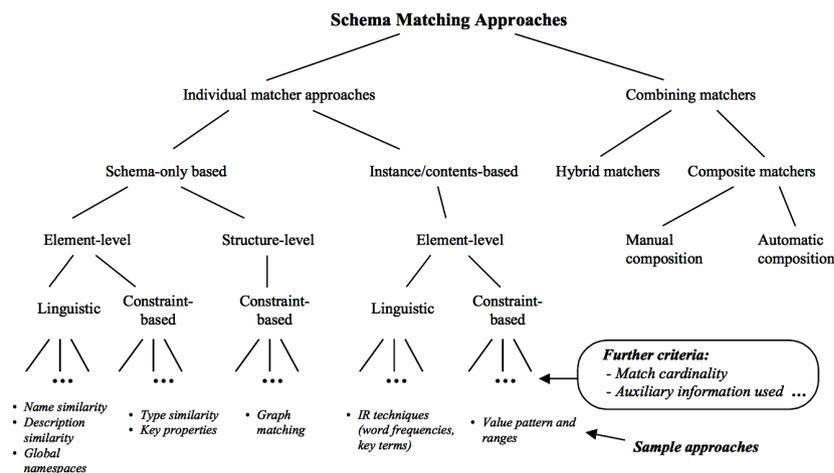


FIGURE 2.10: Classification of schema matching approaches. Taken from [112]

- *Language vs constraint*: a matcher can use a linguistic based approach or a constraint-based approach.
- *Matching cardinality*: the overall match result may relate more elements of one schema to more elements of the other (1:1, 1:n, n:1, and n:m cases).
- *Auxiliary information*: most matchers rely not only on the input schemas S1 and S2 but also on auxiliary information, such as dictionaries, global schemas, previous matching decisions, and user input.

For what concern combined matchers we can categorize them in two ways:

- *Hybrid matchers*, they directly combine several matching approaches to determine match candidates based on multiple criteria or information sources (e.g., by using name matching with namespaces and thesauri combined with data type compatibility).
- *Composite matchers*, they combines the results of several independently executed matchers, including hybrid matchers.

Regarding the problem of *ontology matching* in [103] two major architectures for mapping discovery between ontologies have been identified. For the first approach, recall that the goal of ontologies is to facilitate knowledge sharing. As a result, ontologies are often developed with the explicit goal of providing the basis for future semantic integration. The second set of approaches comprises heuristics-based or machine learning techniques that use various characteristics of ontologies, such as their structure, definitions of concepts, instances of classes, to find mappings. In the field of ontology mapping in the context of IoT we can consider the related work that has been conducted and reported within the Ontology Alignment Evaluation Initiative (OAEI) contest [41]. In this context a number of tools have been evaluated

and reported [41] and most importantly, a number of challenges have been identified [134]. Another interesting work has been the one proposed by Kotis, Katanonov and Leino [78] that reports on a recent approach towards implementing a configurable, multilingual and synthesis-based ontology alignment tool.

In the context of IoT research field, for favoring data integration and exchange, we need to reconcile the semantic heterogeneity among sources by defining logical mappings between source schemes and a common target schema. In the IoT field, data integration problem is usually decomposed in a schema matching phase followed by schema mapping phase. Schema matching finds correspondences between elements of the source and target schemes. Instead, schema mapping defines an appropriate transformation that populates the target schema with data from the sources. An interesting approach based on this data integration procedure is presented in [139]. In this approach the semantic labeling step allows the learning of candidate semantic types for each source attribute and the use of the selected semantic type for mapping an attribute to an element in the domain ontology (a class or property in the domain ontology). Instead, the schema mapping technique is based on a learning semantic model that proposes a mapping able to capture more closely the semantics of the target source in ways that schema constraints could not disambiguate. In details, the semantic model allows the description of a data source in terms of the concepts and relationships defined by the domain ontology.

The first step in building this semantic model for a data source is semantic labeling that relies on determining the semantic types of its data fields, or source attributes. That is, each source attribute is labeled with a class and/or a data property of the domain ontology. However, simply annotating the attributes is not sufficient. Unless the relationships between the columns are explicitly specified, a precise model of the data will be lacking. To build a semantic model that fully recovers the semantics of the data, a second step, that determines the relationships between the source attributes in terms of the properties in the ontology, is required. This kind of approach has been proposed within the *Karma Modeling and Integration Framework*²⁷ [76]. This framework allows users importing data from a variety of sources including relational databases, spreadsheet, XML files and JSON files and also allows importing Domain Ontologies they want to use for modeling the data. The system then automatically suggests a semantic model for the loaded source.

²⁷<http://usc-isi-i2.github.io/karma/>

Chapter 3

Big IoT Data Processing

Big data is a blanket term for the non-traditional strategies and technologies needed to organize, process, and gather insights from large datasets [146]. While the problem of working with data that exceeds the computing power or storage of a single computer is not new, the pervasiveness, scale, and value of this type of computing has greatly expanded in recent years. Even if giving an exact definition of big data is not an easy task, we can surely associate this term to: *i*) large datasets, or datasets that are too large to be processed or stored with traditional tools or on a single computer; *ii*) the types and sets of strategies and technologies that are used to handle large datasets. Most of the information produced comes from different devices and sensors and must be processed and analyzed in order to clean the data for further analysis, to be stored and to extract meaningful information, by some specific tools named *Data Processing Frameworks*.

Depending on the nature of data, the computation can be performed in *batch mode*, in the case of persistent large datasets, or in *real time/streaming mode*, when data requires to be processed in the same instant when it is produced. Another mode is a composition of the two previous approach and it is called *hybrid mode*. Data Processing Frameworks are systems developed for processing data collected in stored databases or generated on the fly by sensors like the one discussed in Chapter 1. These systems have the goal to extract knowledge from the data and also make them actionable, that is being able to actuate a given behavior relying on the verification of events in the analysed data. Processing Frameworks and Processing Engines are responsible for computing over data in a data system. While there is no authoritative definition setting apart "engines" from "frameworks", it is sometimes useful to define the former as the actual component responsible for operating on data and the latter as a set of components designed to do the same. Many of these frameworks have been designed with the aim of processing data with respect of their nature: *batch* or *stream*. For instance, Apache Hadoop¹ can be considered a processing framework with MapReduce [35, 109] as its default processing engine that has been specifically

¹<http://hadoop.apache.org/>

developed for batch processing. Apache Storm², on the other hand, has been designed with the idea of elaborating streams of data. Some other frameworks, for example Apache Spark³, are hybrid systems that can manage the elaboration of data in batch and in stream mode. Data, in order to be analyzed need to be transmitted from the physical device to the processing frameworks. There is the need to have a common language or common communication rules and this is the task of the *Communication Protocols*.

The chapter is organized as follows. Section 3.1 provides a description of the main characteristics of Big Data. Section 3.2 describes some of the most used Communication Protocols. Batch Processing Systems are described in Section 3.3, whereas Section 3.4 presents Stream Processing Systems. Hybrid approaches are discussed in Section 3.5. Finally, Section 3.6 provides a comparison among the different systems.

3.1 Characteristics of Big Data

Big data has been initially associated with the "three Vs" description that, during the last years, has evolved in "five Vs" and recently "six Vs" [39].

- *Volume*: big data means large datasets with an order of magnitude larger than traditional ones. Hundreds of Gigabytes (GB), Terabytes (TB) and also Petabytes (PB) of data can be produced and analyzed.
- *Velocity*: data flows at an unprecedented speed. Data, frequently flowing from different sources like RFID tags, sensors, smart metering devices, increase the request to handle and process them in real time.
- *Variety*: the variety of devices that produce data increases the variety of formats and types of media. Big data systems should handle structured and unstructured data, numerical data, text files, emails, videos, audios, ecc.
- *Veracity*: the variety of sources and the complexity of the processing can lead to challenges in evaluating the quality of the data (and consequently, the quality of the resulting analysis).
- *Variability*: variation in the data leads to wide variation in quality. Additional resources may be needed to identify, process, or filter low quality data to make them more useful.
- *Value*: the ultimate challenge of big data is delivering value. Sometimes, the systems and processes in place are complex enough that using the data and extracting the actual value can become difficult.

²<http://storm.apache.org/>

³<http://spark.apache.org/>

In general we can define four categories of activities involved with data processing:

- *Ingesting raw data into the system.* In this phase some level of analysis, sorting and labelling usually take place. Typical operations might include modifying the incoming data to format them, categorizing and labelling data, filtering out unneeded or bad data, or potentially validating that it adheres to certain requirements. Technologies such as *Apache Sqoop*⁴, *Apache Flume*⁵ and *Apache Kafka*⁶ have been specifically designed for this activity.
- *Persisting the data in storage.* This activity can be thought as the most simple one, but the volume of incoming data, the requirements for availability, and the distributed computing layer make more complex storage systems. Many solutions have been proposed, from distributed file systems like *Apache Hadoop's HDFS*⁷ filesystem, where large quantities of data are written across multiple nodes in a cluster of machines, to distributed databases, especially NoSQL databases [29]. They are often designed with the aim of handling heterogeneous, unstructured or semi-structured, large, complex, diverse and distributed data. Examples of NoSQL databases are *MongoDB*⁸, *Cassandra*⁹, *HBase*¹⁰ and *Neo4J*¹¹.
- *Analysis.* Once the data is available, computing and analyzing data are not easy tasks as the requirements and the best approach can vary significantly depending on what type of insights is desired. Sometimes the goal of this phase is to predict the class or value of new instances in the data stream given some knowledge about the class membership or values of previous instances in the data stream. Machine learning techniques can be used to learn this prediction task from labeled examples in an automated fashion.
- *Visualizing the results.* It is one of the most useful ways to spot trends and make sense of a large number of data points. Due to the type of information being processed in big data systems, recognizing trends or changes in data over time is often more important than the values themselves. Techniques for the visualization of results are: *Elastic Stack*¹², composed of Logstash for data collection, Elasticsearch for indexing data, and Kibana for visualization. These approaches can be used with big data systems to visually interface with the results of calculations or raw metrics. *Jupyter Notebook*¹³ and *Apache Zeppelin*¹⁴

⁴<http://sqoop.apache.org/>

⁵<https://flume.apache.org/>

⁶<https://kafka.apache.org/>

⁷https://hadoop.apache.org/docs/r1.2.1/hdfs_design.html

⁸<https://www.mongodb.com/>

⁹<http://cassandra.apache.org/>

¹⁰<http://hbase.apache.org/>

¹¹<https://neo4j.com/>

¹²<http://www.elastic.co/products>

¹³<http://jupyter.org/>

¹⁴<https://zeppelin.apache.org/>

provide visualization interfaces called *notebook* which allow for interactive exploration and visualization of the data in a format conducive to sharing, presenting, or collaborating.

3.2 Communication Protocols

All communications between devices require that the devices agree on the Communication Protocol, that is the rules defining the data formats. It covers *authentication*, *error detection* and *correction*, and *signaling* and can also describe the syntax, semantic, and synchronization of analog and digital communications. To reach agreement upon the parties involved, a protocol may be developed into a technical standard. Communicating systems use well-defined formats for exchanging messages, each of which has an exact meaning intended to provoke a defined response of the receiver. Protocols should therefore *specify rules governing the transmission*.

In the context of Web applications can be identified three different interaction models of communication [8]: *Request-Reply Interaction*, *Push-Based Data Propagation* and *Publish-Subscribe Interaction Model*, that are discussed in the remainder of the section.

3.2.1 Request-Reply Interaction

The Request-Reply Interaction, or also called Pull-Based Data Access, or Synchronous Data Delivery is specifically tailored for service-oriented applications. It assumes that clients issue requests or queries to a service provider for specific data and the service provider replies with appropriate data. In the following a description of the main approaches based on this model is presented.

Representational State Transfer (REST). It is an architectural style for distributed hypermedia systems such as the *World Wide Web (WWW)*. It issues the standard HTTP request, choosing one of the methods such as GET, POST, PUT, and DELETE, and a server responds with appropriate data. The REST architectural style denotes interaction between a client and a server based on resources that are accessed using the HTTP protocol and represented by XML, HTML, or JSON formats.

Standard Web Service. It is a Web service with the aim to provide communication interoperability among different software platforms where the interface is described by Web Services Description Language (WSDL) [98] and messages are exchanged through the Simple Object Access Protocol (SOAP) [96], which is based on the HTTP protocol and XML data format. However, the standard Web service can be used only from the browser's plug-ins, and not from the standard HTML-JavaScript based platforms.

Remote Object Method Call. This method is available in various Web frameworks Application Programming Interfaces (API) and is based on the standard HTTP request. A remote procedure call allows the invocation of a proxy object with certain methods saved on the server and its invocation is propagated to the remote object. The common approach is that the client just passes parameters and obtains result when the operation is completed. This approach is suitable if the client and the server have compatible programming platforms and this happens if client objects can actually be mapped to server objects and vice versa.

Constrained Application Protocol (CoAP). It is designed for machine-to-machine (M2M) applications, such as smart energy and building automation, due to the constrained nature of devices and environment [130]. CoAP can be easily translated to more resource demanding HTTP thus enabling the integration of wireless sensor networks (WSN), for example, with the Web through proxies. As an extension of standard REST style discussed above, CoAP allows clients to issue request for observing specific server's resource, which results in receiving asynchronous notifications about the resources from the server.

3.2.2 Push-Based Data Propagation

The Push-Based Data Propagation (the Asynchronous Data Delivery) is a client-server interaction model on which the server, as soon as data are available, immediately sends data to the client. In the following a description of the main approaches that relies on this model is presented.

Long-Polling. A client sends a HTTP request and waits for the data from a Web server. The connection is "kept alive" from the server until new data are available, a timeout event arises, or the client disconnects. A new HTTP request is initiated by the client when data is delivered by the server. Therefore, the server is able to send data to the users at any time because there is always a pending request. The benefit of the Long-Polling technique is the use of a standard port that is not blocked by firewalls; it is robust and works together with the proxy server. The disadvantage is the allocation of a connection per client even if the data are not transferred.

HTTP Streaming. A Web server does not terminate the response message or connection as usual, but rather keeps it open and only appends new data to the response message. This can be implemented through the HTTP content type *multipart*, which enables the server to send data in multiple pieces [62]. To prevent the large size of the response message at the client side, the connection must be terminated and a new HTTP request should be issued periodically.

Socket connections. Sockets are a TCP-based technology for providing bidirectional network communication over a single connection. HTML5 specifications introduced the WebSocket protocol [45], which enables communications over sockets from Web browsers. A WebSocket connection is initiated by upgrading an HTTP request and when the connection is established, the involved parties can start sending messages without the need of header exchange between the parties, so the control data overhead is minimal.

3.2.3 Publish-Subscribe Interaction Model

The last model is the Publish-Subscribe (Pub/Sub) Interaction Model. It enables the client (*subscriber*) to subscribe on data that are associated with a certain topic and to receive the data on that topic published in broadcast by producers (*publishers*). This approach can be seen as a filtering process in which the receiver/subscriber can define which type of messages to receive. For the delivering of messages from the publishers to the subscribers, for ensuring the quality of service, for messages persistence and similar functionalities, the Pub/Sub Model introduces the concept of *message broker*. A message broker is a program module that reside in between of two components which need to communicate and it translates a message from the formal messaging protocol of the sender to the formal messaging protocol of the receiver. This approach is widely used and many protocols has been proposed. In the following, some approach relying on this model are presented.

Message Queue Telemetry Transport (MQTT). MQTT¹⁵ was designed by IBM in 1999 for lightweight M2M communication with the goal of providing a Publish-Subscribe messaging protocol with as minimal as possible bandwidth requirements, code footprint size, power consumption and message data overhead [89]. Clients are allowed to use wildcards while subscribing to topics in order to easily match multiple topics. MQTT provides three level of Quality of Service: *i) At Most Once*, messages are delivered according to the best efforts of the underlying TCP/IP network but message lost or duplication may occur; *ii) At Least Once*, messages are assured to arrive but duplicates may occur; and *iii) Exactly Once*, messages are assured to arrive exactly once.

Advanced Message Queuing Protocol (AMQP). AMQP¹⁶ is a binary, open standard protocol for high performance messaging middleware, primarily designed for enterprise environment, but it is used in various application areas. AMQP 1.0¹⁷ is the

¹⁵<http://mqtt.org/>

¹⁶<https://www.amqp.org/>

¹⁷https://www.oasis-open.org/committees/tc_home.php?wg_abbrev=amqp

Protocol	INITIAL TARGET	STANDARD	TRANSPORT	PUB/SUB MODEL	QoS WITHOUT CONFIRMATION	QoS WITH CONFIRMATION	QoS EXACTLY-ONCE
COAP	REST on Constrained Devices	IETF RFC 7252	UDP	Observing feature	+	+	+
MQTT	Lightweight M2M	OASIS Standard	TCP	Hierarchical topics	+	+	+
AMQP	Enterprise apps	ISO and IEC	TCP	4 Exchange types	?	+	+
XMPP	Instant messaging	IETF RFC 6120, 6121	TCP	Node in pub/sub plug-in	-	-	-
DDS	High-Performance apps	Open Management Group (OMG)	UDP, TCP	Typed topics	23 QoS policies		
Kafka	Logging system	Apache Foundation	TCP	Topics	+	-	-

TABLE 3.1: Comparison of IoT application layer messaging protocols

current version and is a wire-level protocol that defines what is exchanged in the network and, as MQTT, the protocol ensures reliable communication with three-modes of message-delivery: *at-most-once*, *at-least-once*, and *exactly-once*. Message format is composed of common data types, whereby additional meta-data could be provided for data interpretation, thus achieving interoperability between different vendors. Depending on the exchange type, there are four ways of routing messages between publishers and subscribers: *i) direct*, messages are delivered to queues based on a message routing key; *ii) fan out*, routes messages to all of the queues that are bound to it; *iii) topic*, a wildcard match between the routing key and the routing pattern specified in the binding; and *iv) header*, uses the header attributes for routing.

Extensible Messaging and Presence Protocol (XMPP). XMPP¹⁸ is a set of technologies for real time messaging having in its core XML streaming technology. The protocol was developed in 1999 by Jabber open source community for instant messaging (IM) applications. The protocol contains the core specification standardized by Internet Engineering Task Force (IETF) [123] and over 300 extensions through XMPP Extension Protocols (XEPs). In XMPP, clients exchange XML messages called *stanzas* and can be of three types: *message*, *presence*, and *iq* (info/query). The *message stanza* kind can be seen as a "push" mechanism whereby one entity pushes information to another entity, similar to the communications that occur in a system such as email. The *presence stanza* can be seen as a basic broadcast or "Publish-Subscribe" mechanism, whereby multiple entities receive information about an entity to which they have subscribed (in this case, network availability information) while the *iq* is a request-response mechanism, similar in some ways to HTTP. The protocol provides also alternative lightweight implementations for constrained devices [14, 75].

¹⁸<http://xmpp.org/>

Data Distribution Service (DDS). DDS¹⁹ is an open standard middleware Communication Protocol that proposes serverless architecture for high-performance interoperable data sharing using the Data-Centric Publish-Subscribe (DCPS) model. This model assumes typed interfaces by allowing DDS *participants* to define *topics* of certain data types that correspond to data types of data objects which applications want either to publish or receive. *DataWriter* of the given data type are used to publish data objects to certain topic over *Publishers* component within applications, whereas *DataReader* of the given data type are used for receiving data objects over *Subscriber* component. Dynamic discovery of DDS *participants* is the matching of their publications and subscriptions based on topics of the same name and data type. Interface Definition Language (IDL) is used for defining data types. QoS can be defined at the level of Publishers/Subscribers as well as the level of DataWriters/DataReaders. DDS protocol specifies use of multicast UDP within LAN, and TCP transport for communication over WAN.

Apache Kafka. Kafka is a distributed streaming platform that allows building real time streaming data pipelines that reliably get data between systems or application. Streams of Kafka messages are organized into *topics*. A topic is a handle to a logical stream of data, consisting of many *partitions*. A partitions is an ordered, immutable sequence of records of the data served by the topic that reside in different physical node and that is continually appended to a structured commit log. As shown in Figure 3.1, a sequential id number called the offset is associated with each records in the partition that uniquely identifies each record within the partition.

Each Kafka node (*broker*) is responsible for receiving, storing and passing on all of the events from one or more partitions for a given topic. In this way, the processing and storage for a topic can be linearly scaled across many brokers. Similarly, an application may scale out by using many consumers for a given topic, with each pulling events from a discrete set of partitions. When a partition is replicated (for durability), many brokers might be managing the same partition. Then, one of these brokers is designated as the "leader", and the rest are "followers". Kafka is assigning each message within a partition a unique id, the so-called "message offset", which represents a unique, increasing logical timestamp within a partition. This offset allows consumers to request messages from a certain offset onwards, essentially consuming data from a given past logical time.

Table 3.1 provides a comparison of characteristics of some of the application layer messaging protocols presented. We can see that only CoAP belongs to the Request-Reply Interaction while the other are Publish-Subscribe Model. All the protocols have been developed over TCP with the exceptions of DDS, that provides communication also over UDP and CoAP, that does not provide communication over TCP but over UDP. The most important features belong to QoS management. While CoAP

¹⁹<http://portals.omg.org/dds/>

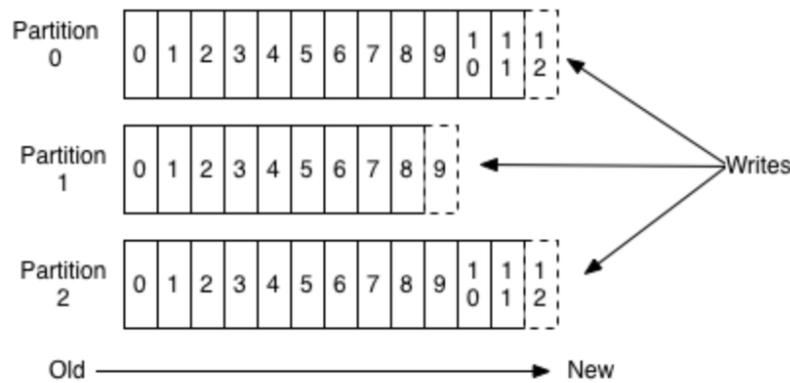


FIGURE 3.1: Anatomy of a topic. Taken from kafka.apache.org

and MQTT guarantee three different QoS, AMQP does not provide QoS without confirmation and Apache Kafka bases its protocol only over QoS without confirmation. XMPP does not guarantee any type of QoS while DDS has a different approach and provides 23 different QoS policies. Regarding the Pub/Sub Model all the approaches, except from CoAP (that is not a Publish-Subscribe Model) uses topics with some differences. MQTT uses hierarchical topics, DDS defines topics based on their type, AMQP introduces four fixed exchange types while XMPP represents topics as a node and Kafka uses a simple abstraction to define topics.

3.3 Batch Processing Systems

Batch processing has a long history within the big data world. Batch processing involves operating over a large, *static* dataset and returning the result at a later time when the computation is complete. The common characteristics of data in Batch Processing Systems are: *i*) bounded and finite dataset or collection of data; *ii*) data is persistent and almost always stored in permanent storage; and *iii*) data processed is always extremely large. Batch processing is well-suited for calculations where access to a complete set of records is required and operations require that state be maintained for the duration of the calculations. Whether the datasets are processed directly from permanent storage or loaded into memory, batch systems are built with large quantities in mind and have the resources to handle them. The trade-off for handling large quantities of data is longer computation time and for this reason, batch processing is not appropriate in contexts where processing time is significant.

Traditional systems rely on the use of *OLAP* (On-Line Analytical Processing) facilities made available by means of data warehouse. The most important and most used framework, suited especially for massive data storage of data and exclusively for batch processing, is **Apache Hadoop**. Other alternatives, like *Google's Big Query*, has similar performances but they can do also real time processing. Apache Hadoop is

a processing framework that exclusively provides batch processing. Hadoop was the first big data framework to gain significant traction in the open source community. Based on several papers and presentations by Google about how they were dealing with tremendous amounts of data at the time, Hadoop reimplemented the algorithms and the component stack to make large scale batch processing more accessible. A number of tools and layers in the Hadoop ecosystem are useful far beyond supporting the original MapReduce algorithm. Among them we mention:

- *HDFS* is a Java-based distributed file system that provides scalable and reliable data storage, on a large cluster of commodity servers. It handles storage and replication of a large number of files, even of considerable size (in the order of GB, TB and even PB) through the use of clusters that contain thousands of nodes and ensure that data remains available in spite of inevitable host failures.
- *YARN* (Yet Another Resource Negotiator), is the cluster coordinating component of the Hadoop stack. It is responsible for coordinating and managing the underlying resources and scheduling jobs to be run.

Hadoop's MapReduce processing stack relies on the following procedure: *i*) read and map the dataset from the HDFS filesystem; *ii*) divide the dataset into chunks and shuffle them among the available nodes; *iii*) each node computes on the subset of data; *iv*) intermediate results (stored in the HDFS) are redistributed to group key; *v*) the value of each key is reduced by summarizing and combining the results calculated on each individual node, and *vi*) write the result back to HDFS.

This kind of approach has some advantages and limitations. Because this methodology heavily leverages permanent storage, reading and writing multiple times per task tends to be fairly slow. On the other hand, since disk space is typically one of the most abundant server resources, it means that MapReduce can handle enormous datasets. This also means that Hadoop's MapReduce can typically run on less expensive hardware than some alternatives since it does not attempt to store everything in memory. MapReduce has incredible scalability potential and has been used in production on tens of thousands of nodes.

As a target for development, MapReduce is known for having a rather steep learning curve. Other additions to the Hadoop ecosystem can reduce the impact of this steep learning curve on varying degrees, but it can still be a factor in quickly implementing an idea on a Hadoop cluster. Hadoop has an extensive ecosystem, with the Hadoop cluster itself frequently used as a building block for other software. Many other processing frameworks and engines have Hadoop integrations to utilize HDFS and the YARN resource manager.

3.4 Stream Processing Systems

Differently from the Batch Processing Systems, these systems execute a set of continuous queries over a stream of data and produce in outputs new results on the fly. Instead of defining operations to apply to an entire dataset, stream processors define operations that will be applied to each individual data item as it passes through the system (or to windows of data). This kind of process "handles" unbounded datasets, that is datasets with the following characteristics: *i*) datasets on which it is possible to define the total size as the amount of data that has entered the system yet; *ii*) datasets limited to a single item at a time; and *iii*) datasets where processing is event-based, results are immediately available and will be continually updated as new data arrives. Stream processing systems can handle a nearly unlimited amount of data, but they only process one (true stream processing) or very few (micro-batch/window processing) items at a time, with minimal state being maintained in between records.

In the first generation of Stream Processing Systems, stand-alone prototypes or extensions of existing database engines have been proposed. They were developed with a specific use case in mind and were very limited regarding the supported operator types as well as the available functionalities (examples include Niagara²⁰, Telegraph²¹ and Aurora [1]). Advanced features were included in the second generation like fault tolerance and adaptive query processing (examples include Borealis²² [1], CEDR [10], System S [55] and CAPE [122]). The key properties of the last generation include high scalability and robustness that are conceived by means of cloud computing. Well-known systems of this generation include Apache Storm²³ and Apache Samza²⁴. A central feature of these cloud-based data Stream Processing Systems is parallelization. This feature is handled differently within the various systems, from the definition of the number of parallel tasks per operator to the creations of new key in the data stream and of the processing element, which is then executed one of the running processing node. In both approaches, the user needs to understand the data parallelism and explicitly enforce in its code the sequential ordering.

An alternative approach is taken by Hadoop Online and StreamMapReduce [23], which present methods to implement stream-based tasks through the MapReduce programming paradigm. The authors extend the notion of *map* and *reduce* with a *stateful reducer* to overcome the strict phasing and thus allowing the usage of the MapReduce paradigm for streaming use cases, which allows a custom and highly parallelized execution.

²⁰<http://datalab.cs.pdx.edu/niagaraST/>

²¹<http://telegraph.cs.berkeley.edu/telegraphcq/v0.2/>

²²<http://cs.brown.edu/research/borealis/public/>

²³<http://storm.apache.org/>

²⁴<http://samza.apache.org/>

Apache Storm. It is a Distributed Stream Processing Computation Framework that focuses on extremely low latency and is perhaps the best option for workloads that require near real-time processing. It can handle very large quantities of data and deliver results with less latency than other solutions. One of the main concepts in Storm is the *Storm Topology*. A topology is a *Direct Acyclic Graph* (DAG) of computation in which each node contains processing logic, and links between nodes indicate how data should be routed around. A topology describes the various transformations or steps that will be taken on each incoming piece of data as it enters the system. In order to better understand this concept, a closer look at the components is provided.

- *Tuple*: it is a list of ordered elements and is the Apache Storm main data structure. By default, a Tuple supports all data types. Every node in a topology must declare the output fields for the tuples it emits.
- *Stream*: it is an unordered sequence of tuples.
- *Spouts*: they are sources of stream. Generally, Storm accepts input data from raw data sources like Twitter Streaming API, Apache Kafka queue, Kestrel queue, etc. Otherwise, customized spouts can be developed to read data from datasources.
- *Bolts*: they are logical processing units. Spouts pass data to bolts and bolts process and produce a new output stream. Bolts can perform the operations of filtering, aggregation, joining, interacting with data sources and databases. Bolt receives data and emits to one or more bolts.

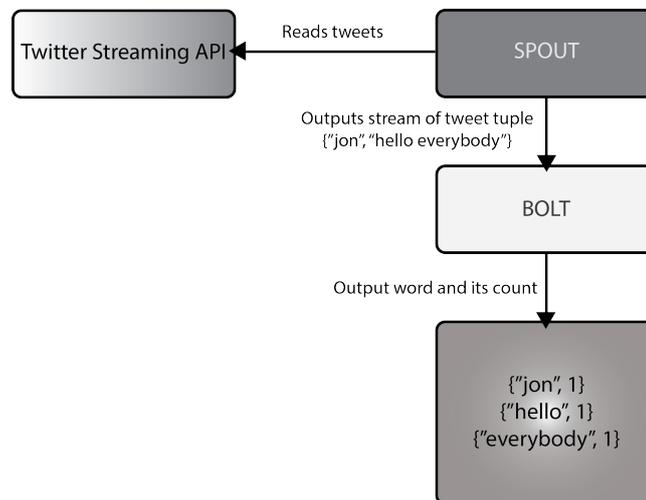


FIGURE 3.2: Structure of a Storm topology

Spouts and bolts are connected together and form a topology. Figure 3.2 depicts the structure of a simple Storm Topology. The input comes from Twitter Streaming API. Spout reads the tweets of the users using Twitter Streaming API and output as a stream of tuples. Then, this stream of tuples is forwarded to the bolt and the

bolt splits the tweets into individual word, calculate the word count, and persist the information to a configured datasource.

There are two kinds of nodes on a Storm cluster: the *master node* and the *worker nodes*. The master node runs a daemon called *Nimbus* that is responsible for distributing code around the cluster, assigning tasks (execution of a spout or a bolt) to machines, and monitoring for failures. Each worker node runs a daemon called *Supervisor* that listens for work assigned to its machine and starts and stops worker processes as necessary based on what Nimbus has assigned to it.

By default, Storm offers at-least-once processing guarantees, meaning that it can guarantee that each message is processed at least once, but there may be duplicates in some failure scenarios. Storm does not guarantee that messages will be processed in order. In order to achieve exactly-once, stateful processing, an abstraction called *Trident* is also available. To be explicit, Storm without Trident is often referred to as *Core Storm*. Trident significantly alters the processing dynamics of Storm, increasing latency, adding state to the processing, and implementing a micro-batching model instead of an item-by-item pure streaming system.

Storm, probably, is currently the best solution for near real time processing. It is able to handle data with extremely low latency for workloads that must be processed with minimal delay. The introduction of Trident allows users to use micro-batches instead of pure stream processing but, while this gives users greater flexibility to shape the tool for their needs, it also tends to negate some of the software's biggest advantages over other solutions (for example the use of micro-batch at the expense of single element). Core Storm does not offer ordering guarantees of messages, it offers at-least-once processing guarantees, so the processing of each message can be guaranteed but duplicates may occur. Trident offers exactly-once guarantees and can offer ordering between batches, but not within. Storm provides, as most of the processing framework, a wide number of APIs for different languages.

Apache Samza. It is a Stream Processing Framework that is tightly tied to the Apache Kafka messaging system. While Kafka is a distributed streaming platform, that can be used by many Stream Processing Systems, Samza is designed specifically to take advantage of Kafka's unique architecture and guarantees. It uses Kafka to provide fault tolerance, buffering, and state storage and uses YARN for resource negotiation. For this reason, a Hadoop cluster (at least HDFS and YARN), relying on the features built into YARN, is required. Samza deals with immutable streams, that represent Kafka's immutable logs. This means that transformations create new streams that are consumed by other components without affecting the initial stream. The strong connection with Kafka affords the system some unique guarantees and features like replicated storage of data that can be accessed with low

latency, easy and inexpensive multi-subscriber model to each individual data partition and writing of all output, including intermediate results, to Kafka and independently consumed by downstream stages. In many ways, this tight reliance on Kafka mirrors the way that the MapReduce engine frequently references HDFS. While referencing HDFS between each calculation leads to some serious performance issues when batch processing, it solves a number of problems when stream processing. Samza's strong relationship to Kafka allows the processing steps themselves to be very loosely tied together. An arbitrary number of subscribers can be added to the output of any step without prior coordination. This can be very useful for organizations where multiple teams might need to access similar data. Teams can all subscribe to the topic of data entering the system, or can easily subscribe to topics created by other teams that have undergone some processing. This can be done without adding additional stress on load-sensitive infrastructure like databases.

Writing straight to Kafka also eliminates the problems of *backpressure*. Backpressure is when the amount of data produced, in a certain instant, is much more bigger than the amount of data that can be processed. This can lead to processing stalls and potentially data loss. Kafka is designed to hold data for very long periods of time, which means that components can process at their convenience and can be restarted without consequence.

Samza is able to store state, using a fault-tolerant checkpointing system implemented as a local key-value store. This allows Samza to offer an at-least-once delivery guarantee, but it does not provide accurate recovery of aggregated state (like counts) in the event of a failure since data might be delivered more than once. Samza offers high level abstractions that are in many ways easier to work with than the primitives provided by systems like Storm and only supports JVM languages at this time, meaning that it does not have the same language flexibility as Storm.

3.5 Hybrid Processing Systems

Some processing frameworks have been developed with the aim of handling both batch and stream workloads. These frameworks simplify diverse processing requirements by allowing the same or related components and APIs to be used for both types of data. The way that this is achieved varies significantly between *Apache Spark*, *Apache Flink* and *Apache NiFi*, the three frameworks we will discuss. This is largely a function of how these approaches combine and bring together the two processing paradigms and what assumptions are made about the relationship between bounded and unbounded datasets. While projects focused on one processing type may have a close fit for specific use-cases, the Hybrid Processing Frameworks attempt to offer a general and wider solution for data processing. For example *Apache Spark* and *Apache Flink* do not only provide methods for processing data, they have

their own integrations, libraries, and tooling for doing things like graph analysis, machine learning, and interactive querying over structured data.

Apache Spark. It is an open-source, general-purpose, lightning fast cluster computing and Batch Processing Framework with stream processing capabilities. Spark focuses primarily on speeding up batch processing workloads by offering full in-memory computation and processing optimization and provides high-level API in Java, Scala, Python and R.

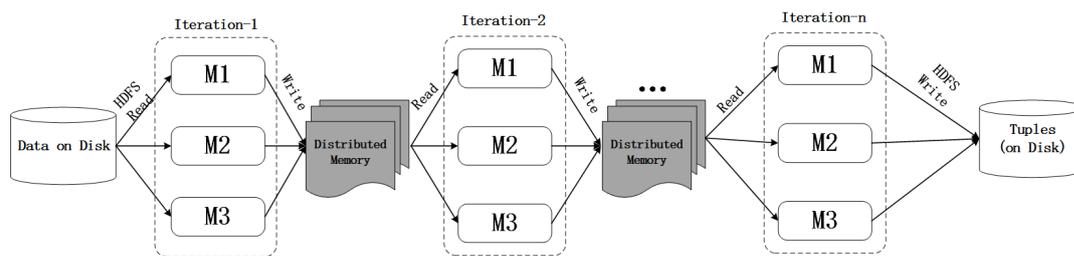


FIGURE 3.3: The iterative operations on Spark RDD. Intermediate results are stored in a distributed memory instead of a Stable storage

For what concern batch processing, Spark processes all data in-memory, only interacting with the storage layer to initially load the data into memory and at the end to persist the final results. All intermediate results are managed in memory. While in-memory processing contributes substantially to speed, Spark is also faster on disk-related tasks because of holistic optimization that can be achieved by analyzing the complete set of tasks ahead of time. It achieves this by creating Directed Acyclic Graphs, or DAGs which represent all of the operations that must be performed, the data to be operated on, as well as the relationships between them, giving the processor a greater ability to intelligently coordinate work. At the core of the Apache Spark architecture there is the *resilient distributed dataset (RDD)*, a read-only multiset of data items distributed over a cluster of machines, that is maintained in a fault-tolerant way [154]. It is an immutable distributed collection of objects. Each dataset in RDD is divided into logical partitions, which can be computed on different nodes of the cluster (Figure 3.3). Formally, RDD is a read-only, partitioned collection of records. It can be created by *parallelizing* an existing collection in a driver program, by *referencing a dataset* in an external storage system or by applying transformation operations on *existing RDDs*.

Stream processing capabilities are supplied by *Spark Streaming*, a library provided in the Spark ecosystem (see Figure 3.4), that enables powerful interactive and data analytics applications across live streaming data. Spark itself is designed with batch-oriented workloads in mind. To deal with the disparity between the engine design and the characteristics of streaming workloads, Spark implements a concept called micro-batches which are executed on top of Spark core. This strategy is designed to treat streams of data as a series of very small batches that can be handled using

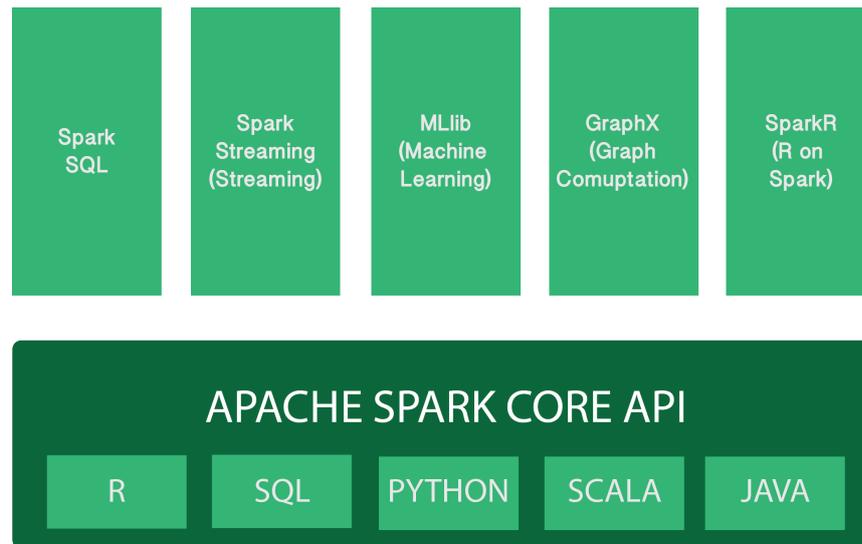


FIGURE 3.4: The Apache Spark Ecosystem

the native semantics of the batch engine. The Spark Streaming data abstraction is called *DStream* (or *Discretized Stream*) which represents a continuous stream of data. Internally, a DStream is represented as a sequence of RDDs. Figure 3.5 shows how



FIGURE 3.5: Spark Streaming working flow. Taken from <https://spark.apache.org/docs/2.0.0-preview/streaming-programming-guide.html>

Spark Streaming works internally. It receives live input data streams and divides the data into batches ready to be processed by the Spark engine. Spark Streaming works by buffering the stream in sub-second increments. These are sent as small fixed datasets for batch processing. In practice, this works fairly well, but it does lead to a different performance profile than true Stream Processing Frameworks.

The obvious reason to use Spark over Hadoop MapReduce is speed. Spark can process the same datasets significantly faster due to its in-memory computation strategy and its advanced DAG scheduling. The Spark versatility is exploit for a deploy that as a standalone cluster or integrated with an existing Hadoop cluster, to perform both batch and stream processing. Beyond the capabilities of the engine itself, Spark also has an ecosystem of libraries that can be used for machine learning (MLib), for structured data queries (Spark SQL) and for graph management (GraphX)²⁵. Spark tasks are almost universally acknowledged to be easier to write

²⁵All available from <https://spark.apache.org/>

than MapReduce, which can have significant implications for productivity. Adapting the batch methodology for stream processing involves buffering the data as it enters the system. By means of the buffer, it is possible to handle a high volume of incoming data, increasing overall throughput, but waiting to flush the buffer also leads to a significant increase in latency. This means that Spark Streaming might not be appropriate for processing where low latency is imperative.

Since RAM is generally more expensive than disk space, Spark can cost more to run than disk-based systems. However, the increased processing speed means that tasks can complete much faster, which may completely offset the costs when operating in an environment where you pay for resources hourly. One consequence of the in-memory design of Spark is that resource scarcity can be an issue when deployed on shared clusters. In comparison to Hadoop's MapReduce, Spark uses significantly more resources, which can interfere with other tasks that might be trying to use the cluster at the time.

Apache Flink. It is an efficient distributed general-purpose data analysis and data processing platform that provides API for Java, Scala, Python and SQL programming support.²⁶ It considers batches to simply be data streams with finite boundaries, and thus treats batch processing as a subset of stream processing. In this stream-first approach, called the *Kappa architecture* [80], streams are used for everything (in contrast with the Lambda architecture, where batching is used as the primary processing method) and, due to the recent growth of performances of processing engines, it is now possible to simplify the model.

The core computational element of Flink, the *Flink dataflow Runtime* on Figure 3.6, is a distributed system that accepts streaming dataflow programs and executes them in a fault-tolerant manner in a cluster of machines. This runtime can be executed taken in a cluster, as an application of YARN (Yet Another Resource Negotiator) or in a Mesos cluster (under development), or within a single machine, which is very useful for debugging Flink applications [52]. Apache Flink includes two core APIs: a *DataStream API* for bounded or unbounded streams of data and a *DataSet API* for bounded data sets. Its stream processing model handles incoming data on an event-at-a-time rather than a series of batches. This is an important distinction, as this is what enables many of its resilience and performance features. The basic components that Flink works with are:

- *streams*, e.g. immutable, unbounded datasets that flow through the system.
- *Operators*, e.g. functions operating on data streams to produce other streams.
- *Sources*, e.g. the entry point for streams entering the system.

²⁶<http://flink.apache.org/>

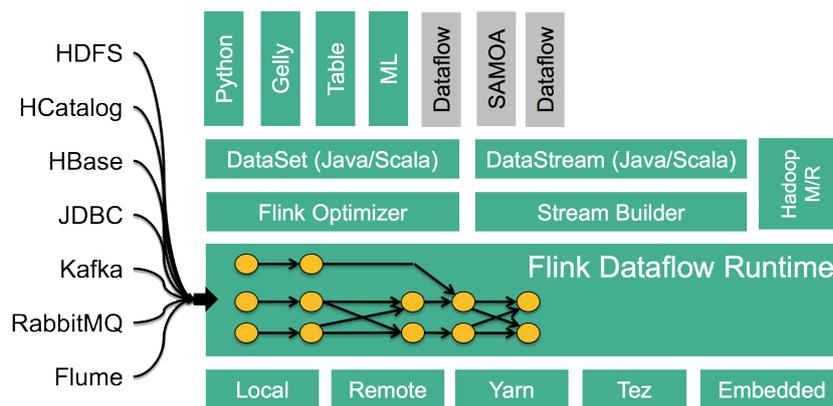


FIGURE 3.6: The Apache Flink Ecosystem. From flink.apache.org

- *Sinks*, e.g. the place where streams flow out of the Flink system. They might represent a database or a connector to another system.

Stream processing tasks take snapshots at set points during their computation to use for recovery in case of problems. For storing state, Flink can work with a number of state backends depending with varying levels of complexity and persistence. Additionally, Flink's stream processing can guarantee ordering and grouping with the introduction of the concept of "event time". It means that is able to understand the time that the event actually occurred, and can handle sessions as well.

The Flink batch processing model is an extension of the stream processing model so far discussed. Instead of reading from a continuous stream, Flink reads a bounded dataset of persistent storage as a stream and uses the exact same runtime for both of these processing models. Flink offers some optimizations for batch workloads. For instance, since batch operations are backed by persistent storage, Flink removes snapshotting from batch loads. Data is still recoverable, but normal processing completes faster. Another optimization involves breaking up batch tasks so that stages and components are only involved when needed. This helps better coordination between users of the cluster and gives Flink the ability of analyze preemptively the tasks. It can optimize the processing by seeing the entire set of operations, the size of the data set and the requirements of steps coming down the line. Flink is currently a unique option in the processing framework world. While Spark performs batch and stream processing, its streaming is not appropriate for many use cases because of its micro-batch architecture. Flink's stream-first approach offers low latency, high throughput, and real entry-by-entry processing. Somewhat unconventionally, Flink manages its own memory instead of relying on the native Java garbage collection mechanisms for performance reasons. Unlike Spark, Flink does not require manual optimization and adjustment when the characteristics of the processed data change. It handles data partitioning and caching automatically as well.

Flink analyzes its work and optimizes tasks in a number of ways. Part of this analysis is similar to what SQL query planners do within relational databases, mapping out the most effective way to implement a given task. It is able to parallelize stages that can be completed in parallel, while bringing data together for blocking tasks. For iterative tasks, Flink attempts to do computation on the nodes where the data is stored for performance reasons. It can also do "delta iteration",²⁷ or iteration on only the portions of data that have changes.

In terms of user tooling, Flink offers a web-based scheduling view to easily manage tasks and view the system. Users can also display the optimization plan for submitted tasks to see how it will actually be implemented on the cluster. As proposed by Spark, Flink offers libraries for SQL-style querying, graph processing and machine learning libraries, and in-memory computation.

Flink operates well with other components. It integrates with YARN, HDFS, and Kafka easily and can run tasks written for other processing frameworks like Hadoop and Storm with compatibility packages. One of the largest drawbacks of Flink at the moment is that it is still a very young project. Large scale deployments in the wild are still not as common as other processing frameworks and there has not been much research into Flink's scaling limitations. With the rapid development cycle and features like the compatibility packages, there may begin to be more Flink deployments as organizations get the chance to experiment with it.

Apache NiFi. Differently from the previous two described frameworks, *Apache NiFi* is a software designed to automate the flow of data between the software systems²⁸. It is a tool for the ingestion of streaming and batch data, and allows the user to enrich and modify data and store it in a database or in a file system. It stems from the need to collect data from different IoT sources and for this reason it comes equipped with different input and output connectors.

It is based on the "*NiagaraFiles*", a lightweight software previously developed by the NSA [104] able to receive data from different sources and apply a basic "preparation" of the data before storing it on different devices. Subsequently it becomes part of the Apache Software Foundation and becomes an open-source software in 2014.

The main characteristic, that let it differ from the other frameworks, is the presence of a web-based interface through which a graphical **dataflow** of services that data should follow is created (see Figure 3.7). Each node that composes the dataflow is called *processor* and they are connected to each other through *queues*. Depending on the type of processor there could be different queues based on the response of the elaboration (e.g. *failure* or *success*). Processors are placed on the canvas by dragging and dropping them.

²⁷<https://www.slideshare.net/wadkarsameer/flink-batchanditerationsv2>

²⁸<http://nifi.apache.org/>

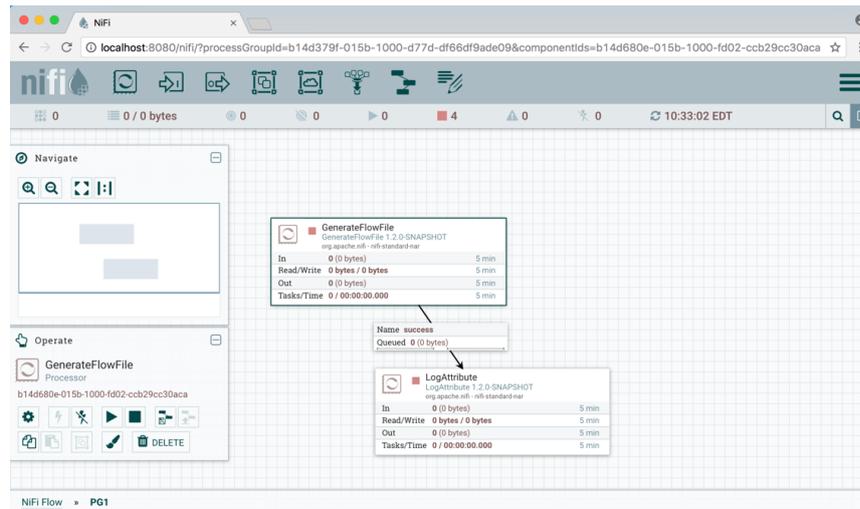


FIGURE 3.7: The Apache NiFi GUI

NiFi	FDP	DESCRIPTION
FlowFile	Information Packet	A FlowFile represents each object moving through the system and for each one, NiFi keeps track of a map of key/value pair attribute strings and its associated content of zero or more bytes.
FlowFile Processor	Black Box	Processors actually perform the work. A processor is doing some combination of data routing, transformation, or mediation between systems. Processors have access to attributes of a given FlowFile and its content stream. Processors can operate on zero or more FlowFiles in a given unit of work and either commit that work or rollback.
Connection	Bounded Buffer	Connections provide the actual linkage between processors. These act as queues and allow various processes to interact at differing rates. These queues can be prioritized dynamically and can have upper bounds on load, which enable back pressure.
Flow Controller	Scheduler	The Flow Controller maintains the knowledge of how processes connect and manages the threads and allocations thereof which all processes use. The Flow Controller acts as the broker facilitating the exchange of FlowFiles between processors.
Process Group	Subnet	A Process Group is a specific set of processes and their connections, which can receive data via input ports and send data out via output ports. In this manner, process groups allow creation of entirely new components simply by composition of other components.

TABLE 3.2: Comparison of NiFi and FDP components and their description. From <https://nifi.apache.org/docs.html>

Apache NiFi is based on the fundamental design concepts closely related to the main ideas of *Flow Based Programming* [99]. The main concepts are explained on Table 3.2.

The NiFi Architecture is presented on Figure 3.8. NiFi executes within a JVM on a host operating system. The primary components of NiFi on the JVM are as follows:

- Web Server, its purpose is to host NiFi's HTTP-based command and control API.
- Flow Controller, is the brains of the operation. It provides threads for extensions to run on, and manages the schedule of when extensions receive resources to execute.
- Extensions, they operate and execute within the JVM. There are various types of NiFi extensions.

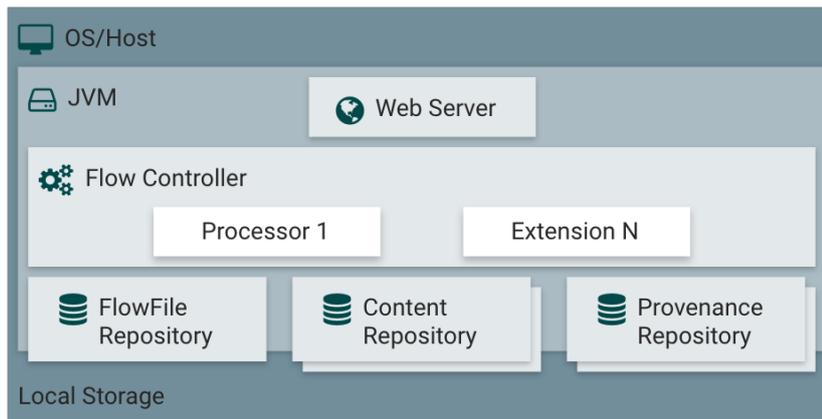


FIGURE 3.8: The Apache NiFi Architecture. From <https://nifi.apache.org/docs.html>

- FlowFile Repository, it is where NiFi keeps track of the state of what it knows about a given FlowFile that is presently active in the flow. The implementation of the repository is pluggable. The default approach is a persistent Write-Ahead Log located on a specified disk partition.
- Content Repository, it is where the actual content bytes of a given FlowFile live. The implementation of the repository is pluggable. The default approach is a fairly simple mechanism, which stores blocks of data in the file system. More than one file system storage location can be specified so as to get different physical partitions engaged to reduce contention on any single volume.
- Provenance Repository, it is where all provenance event data is stored. The repository construct is pluggable with the default implementation being to use one or more physical disk volumes. Within each location event data is indexed and searchable.

NiFi is also able to operate within a cluster.

3.6 Comparison of IoT Streaming Systems

In this chapter we provide a description of different options for big data processing. We discover three different methods and approaches to handle the different type and nature of data and also the way to extract meaningful information. Table 3.3 provides a comparison of the main characteristics of the proposed solutions. Hadoop is the only "batch-only" framework present in the table and is not possible to compare it with the other frameworks that have been developed with the aim of processing data streams. We can see that Hadoop is the oldest framework (since December 2011) while Apache Flink is the most recent. The fact that Apache Flink is still in the early stage of adoption can lead to choosing a more mature system,

like Apache Spark or Apache Storm and it can be also deduced by the number of contributors. Another feature that can support the choice to a system rather than another is latency. Hadoop provides a high latency, but if we decide to use this framework, we can work on workloads that are not time-sensitive and we cannot provide real time results. Storm offers a very low latency but the fact that it can deliver duplicates and is not able to guarantee ordering can be a problem in certain situations. The introduction of trident helps and resolves these issues but leads to an higher latency. Spark provides a medium latency due to the use of micro-batch instead of single record, while in Apache Flink latency can also be configurable. The strong point of all the solutions is the availability of different APIs for different languages and the fact that many notable users use them can confirm their effectiveness and performances. Apache Spark and Apache Flink provide wide support, integrated libraries and tools to use them flexibly in different contexts. By contrast the tight integration of Apache Samza with Apache Kafka can limit its use and make it too sectorial.

The best solution on adopting one system rather than another strongly depends upon the state of data to process, how time-bound the requirements are, what kind of results the application expects. For sure, for batch processing, Apache Hadoop is the best solution due to its strong community and the long history. Regarding stream processing systems Apache Samza is a good solution, but its strong connection with Apache Kafka makes it too sectorial while Apache Storm has been the leader in stream processing from the beginning, but the unordered sort of the processed results and duplications leads to choose other systems for some specific real time use.

Apache Spark is nowadays the best solution for stream processing. It provides high speed batch and micro-batch (streaming) processing and it widely used (more than 800 contributors) but its micro-batching nature can introduce some limitations that Apache Flink tries to resolve with its heavy optimization an the use of single event process. However, it is not already enough mature and bugs are still a constant.

	HADOOP	STORM	STORM+ TRIDENT	SAMZA	FLINK	SPARK STREAM- ING	NIFI
Current Version	2.9.0	1.0.5	1.0.5	0.13.0	1.3.2	2.0.0	1.5.0
Category	BPS	ESP/CEP	ESP/CEP	ESP	ESP/CEP	ESP	CEP
Event Size	-	Single	Micro-batch	Single	Single	Micro-batch	Single
Available Since	Dec 2011	Sep 2014	Sep 2014	Jan 2014	Dec 2014	Feb 2014	Nov 2014
Contributors	148	206	206	48	159	838	Unbounded
Main Backers	Hortonworks, Cloudera	Backtype, Twitter	Backtype, Twitter	Linkedin	dataArtisan	AMPLab, Databricks	Hortonworks
Delivery guarantees	-	At least once M2M	Exactly once	At least once	Exactly once	Exactly once, at least once (with non-fault-tolerant sources)	-
State management	-	Record, acks	Record, acks	Local and distributed snapshots	Distributed snapshots	Checkpoints	Failure, Success, Not-Match
Fault Tolerance	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Event prioritization	-	Programmable	Programmable	yes	Programmable	Programmable	Programmable
Windowing	-	Time-based, count-based	Time-based, count-based	Time-based	Time-based, count-based	Time-based	-
Back-pressure	-	No	No	Yes	Yes	No	-
Primary abstraction	Record	Tuple	TridentTuple	Message	DataStream	DStream	-
Latency	High	Very low	Medium	Low	Low (configurable)	Medium	
Resource management	HDFS, YARN	YARN, mesos	YARN, mesos	YARN	YARN	YARN, mesos	HDFS, FTP, File, etc.
API	Declarative	Compositional	Compositional	Compositional	Declarative	Declarative	-
Primarily written in	Java	Clojure	Java	Scala	Java	Scala	-
API languages	C++, Java, Python, PHP, Ruby, Erlang, Perl, Haskell, C#, Cocoa, Smalltalk, OCaml	Scala, Java, Clojure, Python, Ruby	Java, Python, Scala	Java	Java, Scala	Scala, Java, Python	-
Notable users	Yahoo!, Facebook, Twitter, LinkedIn, Google	Twitter, Yahoo!, Spotify, Groupon, Flipboard, The Weather, Channel, Alibaba, Baidu, Yelp, WebMD	Klout, GumGum, Crowd-Flower	Linkedin, Netflix, Intuit, Uber	King, Otto Group	Kelkoo, Localytics, AsiaInfo, Opentable, Faimdata, Guavus	-

TABLE 3.3: Comparison of IoT Processing Frameworks (enhanced version of <https://twitter.com/ianhellstrom/status/710917506412716033>)

Chapter 4

Syntactic Data Model and Domain Ontology

The Syntactic Data Model is extracted from information represented by sensors in according to Spatio-Temporal-Thematic (STT) dimensions. The adoption of this three dimension representation is largely used in the context of Smart Cities and smart mobility to define and represent data. This model is, however, flexible in order to be used in different contexts especially where missing information can be enriched and added through other sources of data.

The syntactical data model is anyway independent from the semantics associated with the information in a certain context of use. For this reason the adoption of an ontology is needed to represent sensors and the information they produce. As shown in Chapter 2 different ontologies have been proposed in the context of IoT. We decided to start from the SSN ontology in order to define a Domain Ontology (DO) because it is one of the most used in the context of IoT. This ontology has been extended with other ontologies to represent in the most adequate way the STT dimensions that we use in the syntactical data model and to represent the different spatio-temporal granularities. The resulting Domain Ontology is a general ontology that can be adapted in every application domain and that can be further extended in order to represent specific concepts and relationship of a specific application domain. In our motivating scenario, for instance, we extended the ontology to describe tweets coming from social sensors.

The chapter is organized as follows. Section 4.1 presents our STT Syntactic Data Model and provides a formal description of the three dimensions. Moreover, the definitions of spatio-temporal granularities are provided along with the structure of an event stream. Section 4.2 initially gives a formal and general definition of the concepts and of the structure of our Domain Ontology. Then, the ontological representation of each STT dimension is discussed. Finally, on Section 4.3, a brief summary of the concepts and requirements discussed in the chapter is provided.

4.1 The STT Syntactic Data Model

In the database research field the concept of temporal granularity is a well-known notion used to temporally qualify and aggregate classical information. Different temporal granularities in a lattice represent different qualification levels. We can say, for example, that an article has been published on December 2017 or considering a finer temporal granularity (i.e., finer temporal level of detail) on December 12th, 2017. Transposing the same idea to the spatial context, we obtain the notion of spatial granularity. We can define spatial granularity as the representation of any partition of a space domain (e.g., \mathbb{R}^2) in non-overlapped areas (e.g., Italian regions), called granules. This concept is not meant to represent the same object in different ways at different levels of detail, as happen in the multi representation approach [129], but to study and aggregate objects at different levels. For example, the position of a Congress Hall can be represented by a pair of coordinates but we can consider it at several levels, e.g., city, province, and region [161]. Relying on the concepts of temporal and spatial granularities, we exploit the concept of *event*, that is an instance of a thematic associated with a spatio-temporal granularity [110]. Granularities are used for identifying correlations among events produced by different sensors and for imposing consistency constraints in the composition of sensor events produced by heterogeneous devices.

We remark that all the dimensions of a given event are considered optional in our data model as well as the properties of the thematic. Indeed, some sensors might not be adequately equipped for associating all the required contextual information of a given concept of the Domain Ontology (e.g. a sensor can only provide a real number representing the temperature), or the time/location where the event has been generated. Further metadata (e.g. unit of measure, precision) can be associated with the event during the acquisition process. The possibility offered through this model to represent the STT dimensions generated by the sensors is somehow a "semantic" information extracted from data. For example a gateway in charge of a set of temperature sensors disseminated in a given zone of a city can calculate the average temperature and assign the time/location this observation refers to. However, this kind of semantic can be directly desumed by the events generated from the data and their formats and needs to be validated from the domain experts (as we will discuss in the following chapters). For what concern the spatial dimension, the technical report proposed by W3C [38], describes the best practices and requirements that support the publication of spatial data on the Web.

4.1.1 Spatial and Temporal Granularities and Thematic Dimensions

For the representation of spatio-temporal information of different granularities we adopt the notation developed in [26, 27]. Temporal and spatial granularities are

mapping functions from an index set \mathcal{IS} to the power sets of the temporal and spatial domains, respectively. The temporal domain is represented as a pair (\mathbb{N}, \leq) , where \mathbb{N} is the set of natural numbers representing the set of *time instants* and \leq is a relation order on \mathbb{N} . The spatial domain contains geometric objects represented in one or two dimensions (e.g. points, lines, and regions)¹. Examples of temporal granularity include `second`, `minute`, `day` with the usual meaning adopted in the Gregorian calendar, whereas, `meter`, `kilometer`, `feet`, `yard`, `zone` and `city` are examples of spatial granularities. Let \mathcal{G}_T and \mathcal{G}_S denote a set of time and spatial granularities.

As shown graphically in Figure 4.1(a), a *granule* is a sub set of a domain corresponding to a single granularity mapping, that is, given a granularity G and an index $i \in \mathcal{IS}$, $G(i)$ is a granule of G that identifies a subset of the corresponding domain. Granules of the same granularity are disjoint, so there are no granule overlapping. Figure 4.1(b) shows a violation of this definition while Figure 4.1(c) presents another violation of the definition of granule: non-empty temporal granules preserve the order of the temporal domains. Each non-empty granule of a granularity G is represented by means of the “textual representation” as a *label* (e.g. a label for day can be in the form `mm/dd/yyyy`). Granules are used to specify the valid spatio-temporal bounds on attribute values.

Different granularities provide different partitions of their domains because of the diverse relationships that can exist among granularities, depending on the inclusion and the overlapping of granules [26, 27]. A granularity G is said to be *finer than* a granularity H , denoted $G \preceq H$, if for each index i , there exists an index j s.t. $G(i) \subseteq H(j)$ [17]. For example, temporal granularity `second` is finer than `minute`, and granularity `month` is finer than `year`. Likewise, spatial granularity `zone` is finer than `city`. Figure 4.2 shows different four granularities that are in the *finer-than* relationship. Indeed, $\text{zone} \preceq \text{city} \preceq \text{province} \preceq \text{region}$. In the example, the zone/granule `z1` is a part the city/granule `Milan` which in turn is a part of the province/granule `MilanProvince` and of the region/granule `Lombardy`.

A semantically rich sensor network, besides spatial and temporal information, would provide thematic information for discovering and analyzing sensor events. Thematics represent the type of event that is generated by a sensor. Examples of thematic are `temperature`, `humidity`, `wind speed`, etc. In the following, \mathcal{TH} represents the set of available thematics.

4.1.2 Temporal and Spatial Types and Values

Starting from basic domains (like `int`, `real`, `boolean`), denoted by D_i , we consider structured types, like records and lists, represented according to the JSON format. The set of types is denoted by \mathcal{T} and is used for the representation of the thematic

¹points are modeled by two coordinates; lines by a list of points, regions by their boundary lines.

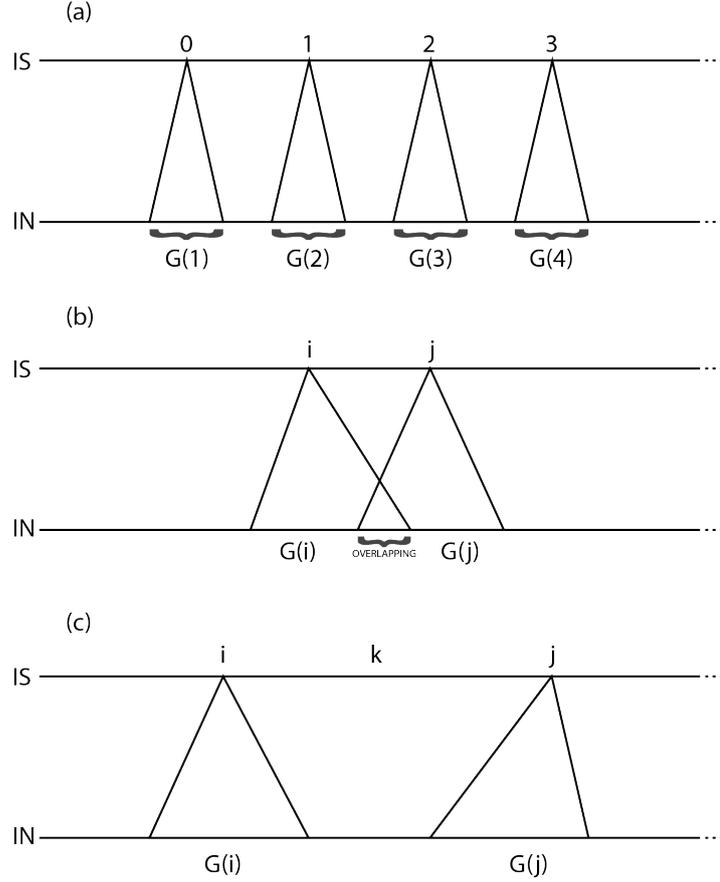


FIGURE 4.1: Graphical representation of temporal granularity

properties. Let \mathcal{A} be a set of labels, the set of legal values \mathcal{V} for the types \mathcal{T} is inductively defined as follows:

- if $d \in D_i$ and $a \in \mathcal{A}$, then $(a : d) \in \mathcal{V}$;
- if $d_1 \in D_1, \dots, d_n \in D_n$ and $a \in \mathcal{A}$, $(a : [d_1, \dots, d_n]) \in \mathcal{V}$;
- if $\{d_1, \dots, d_n\} \subseteq \mathcal{V}$ and $a \in \mathcal{A}$, $(a : [d_1, \dots, d_n]) \in \mathcal{V}$;
- if $(a_1 : d_1) \in \mathcal{V}, \dots, (a_n : d_n) \in \mathcal{V}$ with $a_i \neq a_j$ ($i \neq j$), then $(a_1 : d_1, \dots, a_n : d_n) \in \mathcal{V}$.

Given a type $\tau \in \mathcal{T}$, a spatial granularity $G_S \in \mathcal{G}_S$, and a temporal granularity $G_T \in \mathcal{G}_T$, we denote with: $Spatial_{G_S}(\tau)$, a spatial type; $Temporal_{G_T}(\tau)$, a temporal type, and $Spatio-Temporal_{(G_T, G_S)}(\tau)$ a spatio-temporal type ([26, 27]). Their legal values are defined as partial functions that map each granule $i \in \mathcal{IS}$ (or pair of granules (i, j) for spatio-temporal types) to the legal values for τ .

4.1.3 STT Events and Stream Data Model

Relying on the temporal and spatial granularities, we are now ready for the presentation of the concept of *event*. Our definition covers different kinds of events that

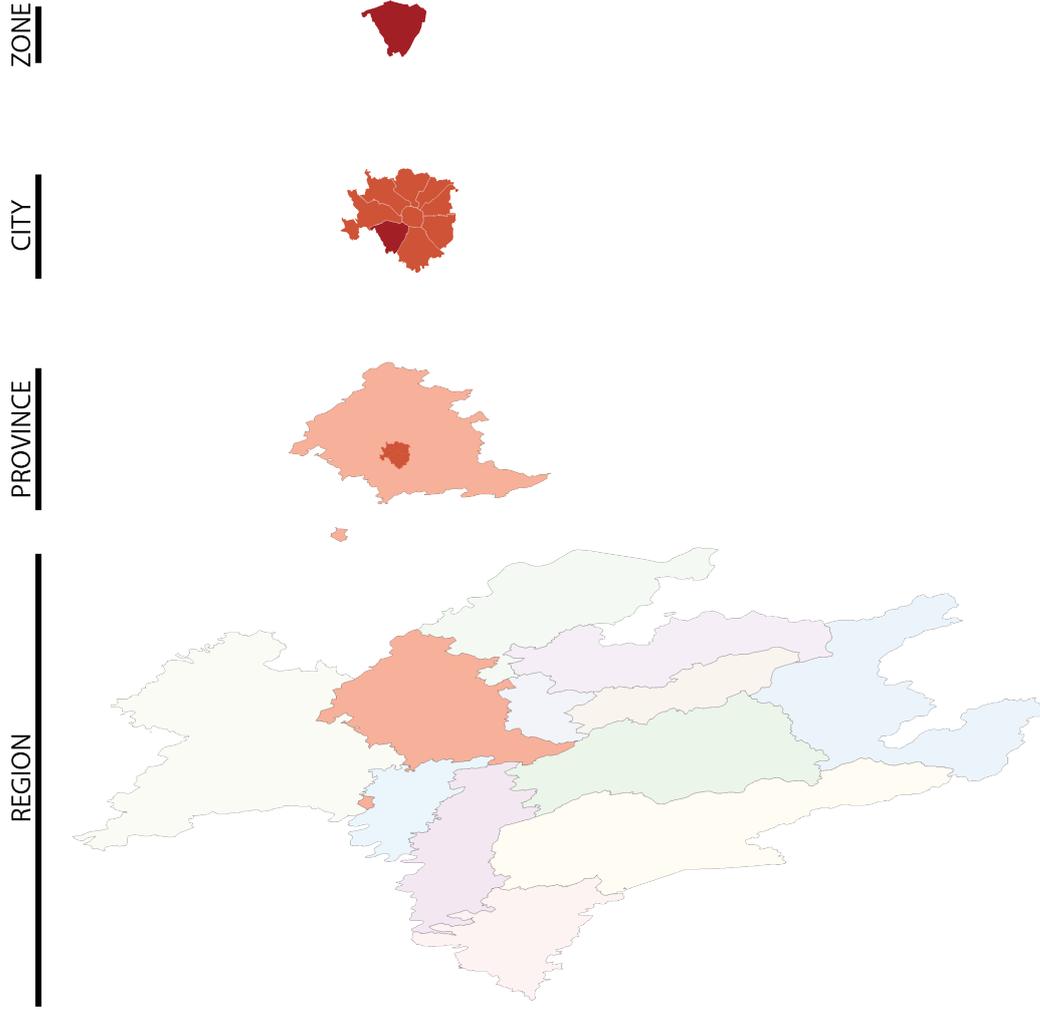


FIGURE 4.2: Graphical representation of spatial granularity

can be associated with the STT dimensions. In the definition \perp denotes a missing component.

Definition 4.1 (*Event Type*). Let $\tau \in \mathcal{T}$ be a type, $G_T \in \mathcal{G}_T \cup \{\perp\}$ a temporal granularity, $G_S \in \mathcal{G}_S \cup \{\perp\}$ a spatial granularity, and $th \in \mathcal{TH} \cup \{\perp\}$ a thematic. An Event Type, denoted $Event_{(G_T, G_S)}^{th}(\tau)$, can be:

- if $G_T = G_S = \perp$, a list of pairs $\langle th, \tau \rangle$.
- if $G_T = \perp$ and $G_S \neq \perp$, a partial function that maps G_S -granules (referred to by their indices) to pair $\langle th, \tau \rangle$ ($Spatial_{G_S}(\langle th, \tau \rangle)$).
- if $G_T \neq \perp$ and $G_S = \perp$, a partial function that maps G_T -granules to pair $\langle th, \tau \rangle$ ($Temporal_{G_T}(\langle th, \tau \rangle)$).
- if $G_T \neq \perp$ and $G_S \neq \perp$, a partial function that maps (G_T, G_S) -granules to pair $\langle th, \tau \rangle$ ($Spatio-Temporal_{(G_T, G_S)}(\langle th, \tau \rangle)$). \square

Example 4.1 Consider the scenario described in our motivating example. In this scenario we can identify four types of events generated by sensors as follows:

- $T_1: \text{Event}_{\langle 10 \text{ minute, point} \rangle}^{\{temperature\}}(temperatureVal:real)$.
- $T_2: \text{Event}_{\langle 20 \text{ minute, } \perp \rangle}^{\{temperature\}}(temperatureVal:real)$.
- $H_1: \text{Event}_{\langle 30 \text{ minute, point} \rangle}^{\{humidity\}}(humidityVal:real)$.
- $TW_1: \text{Event}_{\langle minute, zone \rangle}^{\{tweet\}}((tweets:list(string), numTweets:int))$.

This representation of the event type of each sensor is exploited for modeling the events generated by the sensors in our JSON format. This uniform representation of the sensor data is used for simplifying their processing. Note that the temperatures in T_1 are expressed in Celsius, where those in T_2 are expressed in Fahrenheit, and the notation *point* indicates the event geo-position (latitude, longitude). \square

Relying on the concept of events, we can characterize an event stream.

Definition 4.2 (Event Stream). Let $\tau \in \mathcal{T}$ be a type, $G_T \in \mathcal{G}_T \cup \{\perp\}$ a temporal granularity, $G_S \in \mathcal{G}_S \cup \{\perp\}$ a spatial granularity, $th \in \mathcal{TH} \cup \{\perp\}$ a thematic, $[t_s, t_e]$ a temporal interval, and S a set of spatial values. An event stream is a 6-tuple

$$\langle G_T, G_S, th, [t_s, t_e], S, \text{Event}_{\langle G_T, G_S \rangle}^{th}(\tau) \rangle$$

where: if $G_S \neq \perp$ then $G_S \preceq \text{Gran}_S(S)$, and if $G_T \neq \perp$ then $G_T = \text{Gran}_T(t_s) = \text{Gran}_T(t_e)$. \square

An event stream is thus a sequence of events that is associated with meta information related to the interval in which the flow is acquired and the STT dimensions (when available). The meta information is exploited for imposing the integrity constraints on the produced events.

Example 4.2 Consider the event type associated with the sensors of type TW_1 in our scenario. The event streams of each zone of Milano produced in the year 2016 can be characterized by the tuple $\langle minute, zone, tweet, ["1/1/2016-00 : 00", "1/1/2017-00 : 00"], \{z1, \dots, z9\}, \text{Event}_{\langle minute, zone \rangle}^{\{tweet\}}(tweets : list(string), numTweets:int) \rangle$, where $\{ \langle "15/05/2016-12 : 00", z1, tweet, \{ tweets : ['today is very hot', 'My home is an heater'], numTweets : 2 \} \rangle, \langle "15/05/2016-12 : 01", z1, tweet, (tweets : ['a nice day for me', 'hot,hot,hot', 'I am walking'], numTweets : 3) \rangle, \langle "15/05/2016-12 : 01", z1, tweet, (tweets : ['I am sweat', 'today is very warm'], numTweets : 2) \rangle \}$ is a stream of legal values. \square

4.2 The Domain Ontology

The STT data model so far presented allows one to produce events whose correctness is left to the user in charge of its creation. To support the user in this activity, a Domain Ontology is included within our system for each supported domain. The purpose of the adopted ontology is to guarantee that the properties specified for a given concept (in a certain domain) actually occur in the events produced by the

sensors and that the final events (i.e. those generated at the end of the Data Acquisition Plan) are compliant with the spatial, temporal and thematic concepts made available in the adopted Domain Ontology. When this requirement is addressed we can say that the generated events are consistent with respect to the adopted Domain Ontology.

For the design of the Domain Ontology we take into account two of the most notable works in this field, the IoT-Lite ontology [15] and the SSN ontology [31, 126] from which IoT-Lite ontology is derived. Then, these ontologies are aligned with other foundational ontologies in order to make spatial, temporal and thematic commitments explicit by using further concepts and relations for better explaining their intended meaning. They have been chosen as the upper ontologies because they have ontological frameworks and concepts (e.g. qualities, temporal entities, units of measurements, geospatial positions info) that are needed for making our Domain Ontology consistent with respect to the STT model. Figure 4.3 depicts the concepts of the ontologies integrated in our Domain Ontology and the main relationships between them. For example, DOLCE-UltraLite (DUL) [21] introduces the concept of `DUL:Amount` that is used to identify and specify the type of a value (real, integer, list-String, etc). `qu:QuantityKind`, and `qu:Unit` derives from `QU`² and they provides, respectively, information about instances for thematic values and measurement units (celsius, fahrenheit, kilometers, etc). `Time`³ is used to label temporal instants or interval while `Geo`⁴ introduces the class `Geo:Point` that is necessary in order to define latitude and longitude. Two new classes, used to represent instances coming from social sensors and characterized by the prefix `do:` in the Domain Ontology, have been introduced without taking into consideration any other existing ontologies: `do:Facebook` and `do:Twitter`. The ontology obtained by the integration of these components can be further extended by domain experts with concepts and relationships specifically tailored for representing peculiar characteristics of a given domain of interest. In our graphics we use standard notations for representing our Domain Ontology and its instances. Specifically, dashed rectangles represent instances, whereas dashed lines are used for linking related instances according to a given relation. The dashed circle is used for modeling the data value linked to an instance and created by the data property assertion `hasDataValue`. $\mathcal{I}(C)$ denotes the set of instances of class C , and $C_1 \sqsubseteq C_2$ denotes that C_1 is subclass of C_2 .

In the remainder of the section we discuss the ontological representation of the STT dimensions included in our Domain Ontology.

²<http://purl.oclc.org/NET/ssnx/qu/qu-rec20>

³<http://motools.sourceforge.net/timeline/timeline.html>

⁴<http://www.w3.org/2005/Incubator/geo/XGR-geo-ont/>

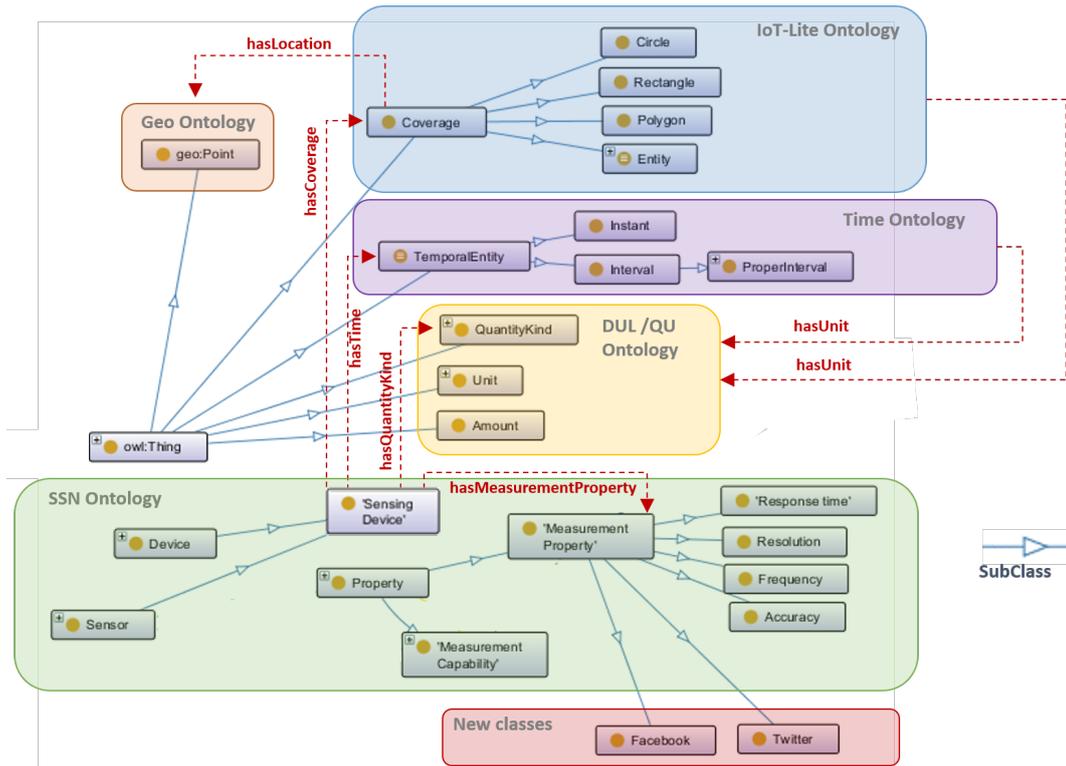


FIGURE 4.3: Relationships between the ontologies that compose our Domain Ontology

4.2.1 Spatial Dimension

The spatial dimension provides information regarding the sensor location, in terms of either a geographical reference system or named location. According to the specification of classes of the IoT-Lite ontology, the spatial information can be modeled using the class `iot-lite:Coverage` that acknowledges that a location can be related to the coverage of an IoT device (i.e. a temperature sensor inside a room has a coverage of that room).

The property `hasPoint` of the `iot-lite:Coverage` class states its location by using the `geo:Point` class and its latitude and longitude properties. By contrast, to specify that a location is a country, a region, a province, a city, etc new subclasses of the class `iot-lite:Entity` (subclass of `iot-lite:Coverage`) are inserted in the ontology. The spatial granularity G_S of a location is specified through a relation of order among the individuals of the subclasses of the class `iot-lite:Entity` and is guaranteed by the unary association `isPartOf` that, for instance, can be used for describing that a province is a part of a region that, in turn, is a part of a country. The spatial granularity G_S can be also specified by using the association `hasSystemReference` between the class `iot-lite:Entity` and the class `GeoSubdivisionStandard` that we defined for instantiating concepts concerning Standard Geographical Administrative Subdivisions. A possible instance of this class can be used for referring to the International

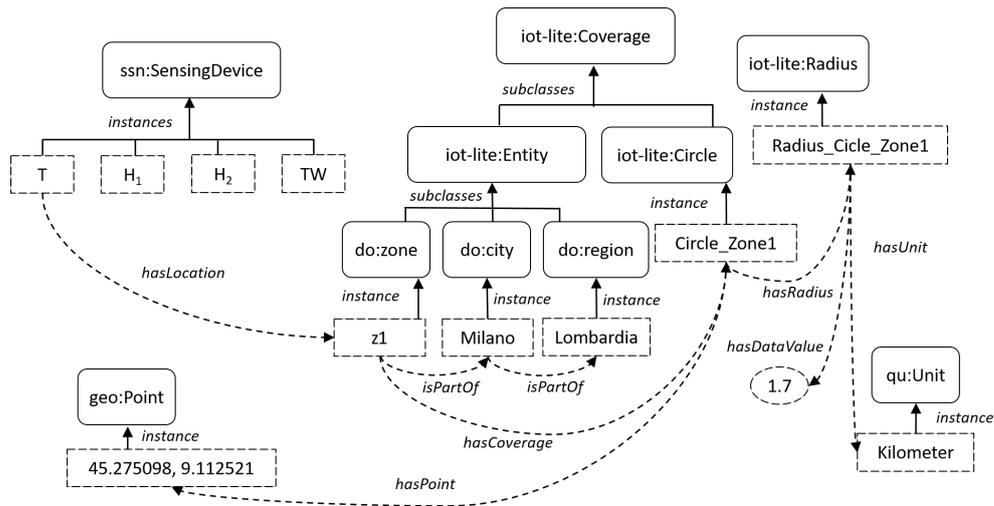


FIGURE 4.4: Twitter sensor of type T

Standard for country codes and the ISO 3166⁵ codes for their subdivisions. This standard defines the codes for the names of countries, dependent territories, special areas of geographical interest, and their principal subdivisions (e.g., provinces or states).

In the IoT-Lite ontology, the class `iot-lite:Coverage` is then associated to the classes: `iot-lite:Rectangle`, `iot-lite:Polygon`, and `iot-lite:Circle` in order to represent the coverage area. Instances of the class `iot-lite:Rectangle` are used for describing that the coverage is made up by giving two geographical points (instances of the class `geo:Point`) which are the opposite corners of a rectangle. Instances of the class `iot-lite:Polygon` describe that the coverage is made up by linking several geographical points by straight lines. Finally, instances of the class `iot-lite:Circle` specify that the coverage of a sensor is a circle with the center in a geographical point and with a given radius. The radius is specified by using the class `iot-lite:Radius` that has to be then associated with an unit of measure (instance of the class `qu:Unit` for indicating that the value of the radius is expressed in meter, kilometer, feet, yard, etc.).

For specifying the relation of order among different units of measurement (i.e. meters vs kilometers or minutes vs hours), the class `qu:Unit` can be linked to the class `qu:SystemOfUnits` that represents the concept of "system of units". This concept is defined as set of base units and derived units, together with their multiples and sub-multiples, defined in accordance with given rules, for a given system of quantities. For example, the most widely accepted and used systems of quantities and system of units are the International System of Quantities (ISQ) and the International System of Units (SI).

Example 4.3 In our running example we wish to represent a Domain Ontology for evaluating the human discomfort in differ zones of Milano. With this aim, domain experts have specified a Domain Ontology with a set of classes and relationships useful to detect human

⁵www.iso.org/iso/home/standards/country_codes.htm

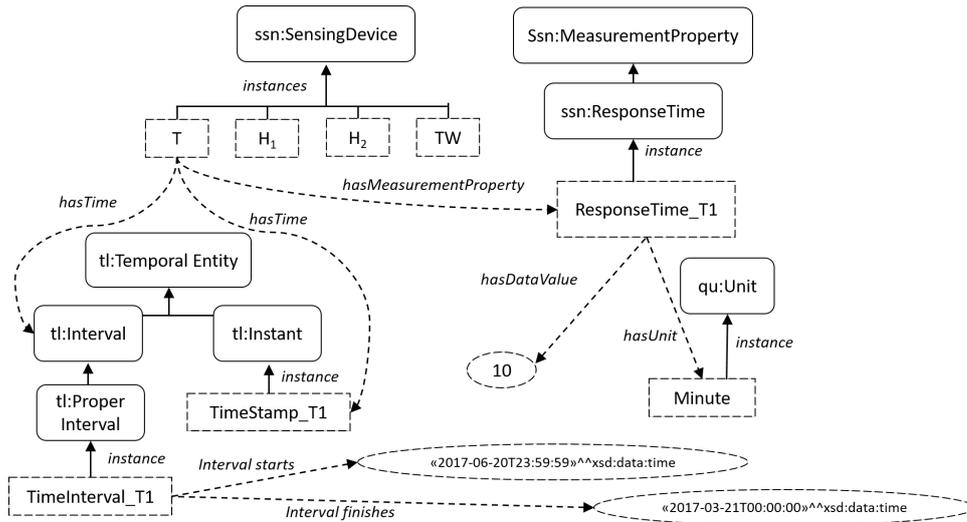
discomfort events. For modeling the spatial dimension, we wish to represent the single points where the sensors are located, the zones of Milano and the entire city. Moreover, we wish to model the finer-than relationships existing among these granularities. Figure 4.4 reports an example of spatial dimension instantiation for a sensor of type T of our running example. In this case the sensor is located in zone 1 of Milano (instance of the new class `do:zone` we inserted as subclass of the class `iot-lite:Entity`) whose shape is rounded by using a circle (instance of the class `iot-lite:Circle`). The Circle coverage is made up by giving the location of the sensor as the center of the circle (e.g. a geo point instance of the class `geo:Point`) and the radius as a `DataProperty`. The granularity is specified through the properties `isPartOf` that connect the instances `z1`, `Milano` and `Lombardia` of the new classes `do:zone`, `do:city` and `do:region` we introduced in the Domain Ontology. The other zones in Milano are represented in a similar way through the class `iot-lite:Polygon` for modeling the vertices delimiting each area. \square

4.2.2 Temporal Dimension

The temporal dimension in sensor events and its observation and/or measurement data are used for describing attributes such as time zone and measurement timestamp. For modeling such concepts, the ontology is integrated with the Timeline Ontology [116] that extends OWL-Time with various temporal concepts such as `Instant`, `Interval`, and `Interval` relationships. In detail, we are interested in two main subclasses of `Temporal Entity`: `t1:Instant` and `t1:Interval`. The instances of the class `t1:Instant` are used for describing instants of time, and the instances of the class `t1:Interval` are used for specifying intervals by means of which we describe that a sensor gathers events from time t_1 to time t_2 by means of the properties `Interval starts` and `Interval finishes` of the class `t1:Proper Interval` subclass of the class `t1:Interval`. As with the `iot-lite:Coverage`, through the association between the temporal entities and the class `qu:Unit`, it is possible to specify the granularity of the detected time (day, hour, minute, second) and evaluate their relationships.

Example 4.4 Figure 4.5 describes an instance of the temporal dimension associated with a sensor. In this case, the sensor of type T is linked to a temporal interval (instance of the class `t1:Proper Interval`) that states that this sensor gathers events during a interval of time specified by using the properties `Interval starts` and `Interval finishes`. \square

The temporal dimension $t \in T$ of events coming from a sensor at a granularity G_T , is strictly related to the instant (the timestamp) of gathering of the feature of interest we want to monitor in an event such as temperature, humidity, etc. This information is modelled by using an instance of the of the class `t1:Instant`. Through the property `hasMeasurementProperty` it is possible to link a sensor to the class `ssn:ResponseTime`, subclass `ssn:MeasurementProperty` (\sqsubseteq `iot-lite:Property`) of

FIGURE 4.5: Temporal dimension of sensor of type T

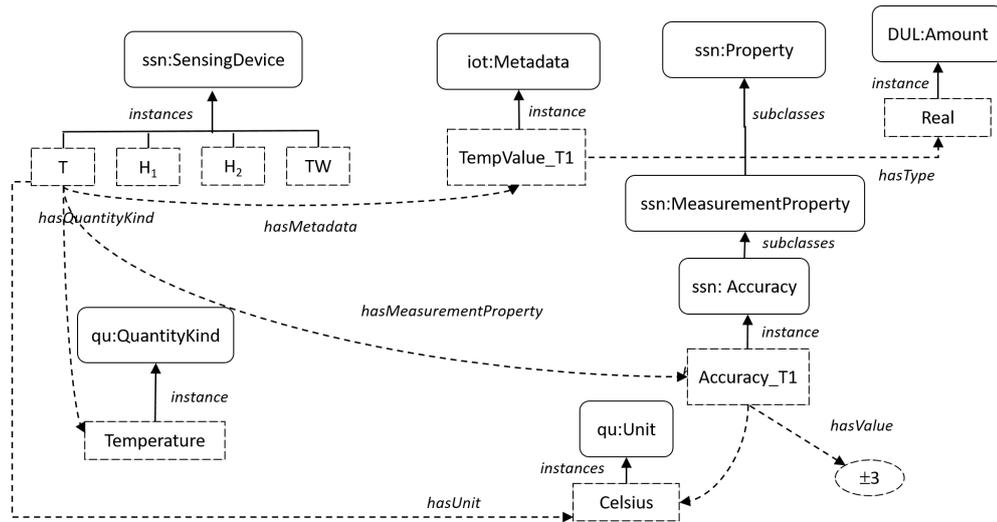
the SSN ontology for specifying the time of sampling. Therefore, the concept “response time” is used for describing the granularity at which events are generated by a sensor.

Example 4.5 In our motivating example, a sensor of type T gathers temperature every 20 minutes. This situation is described in Figure 4.5 by introducing an instance of the class `ssn:ResponseTime`. An instance of the class `qu:Unit` is then used for specifying the time granularity `minute`. Moreover, through an instance of the class `tl:Instant` we describe that the sensor granularity is expressed by using a timestamp. \square

4.2.3 Thematic Dimension

This dimension refers to the type of events that is observed and is described by a record of property-values pairs. We use this representation because the single observation can be enriched by other information that can be directly generated by the sensor or added during the data acquisition process.

In our ontology a thematic is an instance of the class `qu:QuantityKind` of the IoT-Lite ontology, and is used for describing the meaning of the values dispatched by a sensor. The abstract classifier `qu:QuantityKind` represents the concept of “kind of quantity” that is defined as “*aspect common to mutually comparable quantities*” [34]. A quantity is defined as a characteristic of a phenomenon, where it has a magnitude that can be expressed as a number (i.e. the degree of a thermometer) and a reference (i.e. the temperature). Through the instances of this class we are able to represent the kind of values gathered by a sensor such as `temperature`, `humidity`, `wind speed`, etc. Quantities of the same kind (e.g. the values gathered by two thermometers) have the same quantity dimension. However, quantities of the same dimension are not necessarily of the same kind (e.g. sensors T_1 and T_2 in our running example gather the

FIGURE 4.6: Thematic dimension of sensor of type T

temperatures using two different units of measurement: Celsius and Fahrenheit). For this reason a sensor associated with a given theme (e.g. temperature) can be linked to an instance of the class `qu:Unit` for specifying the unit of measure of the detected values (e.g. Celsius for a temperature).

In order to model the event type associated with a sensor, the class `iot:Metadata` is used. Its instances model the data type of the entity whose thematic is observed. As an example, for the representation of a temperature value, an instance of the class `iot:Metadata` is linked to the instance of the sensor by means of a `hasMetadata` link. The instance of the class `iot:Metadata` is then linked to an instance of the class `DUL:Amount` for expressing its domain.

In some contexts of use, the events gathered by sensors are coupled with some measurement properties that characterize their thematic. Measurement properties (e.g. accuracy, range, precision of `ssn:Property`) identify observable characteristics of a sensor's events or ability to make observations. Specifically, these properties can refer to: *i*) the observed characteristics of the measurement or; *ii*) other information that can be used for a given kind of analysis (e.g. the number of tweets related to high temperature). The first kind of properties can be modeled in the SSN ontology by means of the classes: `ssn:Accuracy`, `ssn:DetectionLimit`, `ssn:Frequency`, `ssn:Latency`, `ssn:MeasurementRange`, `ssn:Precision`, `ssn:Resolution`, `ssn:Drift`, `ssn:Sensitivity` and `ssn>Selectivity` \sqsubseteq `ssn:MeasurementProperty`. The second kind of properties can be modeled by introducing new classes in the Domain Ontology.

Example 4.6 Figure 4.6 shows how sensors of type T are semantically described by the class `ssn:SensingDevice` and whose thematics are modeled by the instance `temperature` of the class `qu:QuantityKind`. A sensor gathers temperature values in Celsius (instance of the class `qu:Unit`) and it is then linked to an instance of the class `iot:Metadata` for modeling

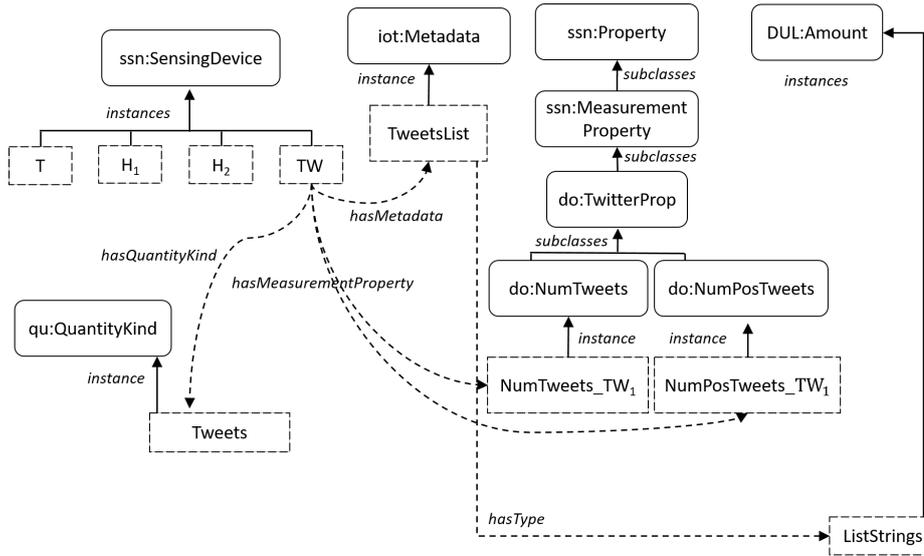


FIGURE 4.7: Thematic dimension of sensor of type TW

the temperature data type. Moreover the sensor of type T presents a measurement property for reporting the accuracy of the retrieved value (more or less 3 Celsius degrees), which is expressed by means of an instance of the class `ssn:Accuracy`. The instances of the class `qu:Unit` are used for specifying the unit of measurement (Celsius) of the accuracy.

As indicated in Figure 4.7 the sensor of type TW deals with the Twitter thematic. This concept is modeled by using the thematic `Tweets` instance of the class `qu:QuantityKind`. The data type associated to this sensor is a list of strings (tweets) modeled by using an instance of the class `iot:Metadata`. The set of properties of the retrieved tweets, that is, the total number of tweets, and the number of positive tweets (i.e. those that contain the terms *hot*, *heat*, and *sweat*) is modeled by the instances of the subclasses `do:NumTweets` and `do:NumPosTweets` of a new class `do:TwitterProp` \sqsubseteq `ssn:MeasurementProperty`. Two properties are linked to the sensor of type TW that are used for modeling the number of tweets and the number of positive tweets. In a similar way, we can introduce the instance `HumanDisc` for representing the thematic Human Discomfort and its properties required for the computation of the formula. \square

4.3 Concluding Remarks

In this chapter we described the Syntactic Data Model used to define the Spatio-Temporal-Thematic (STT) dimensions that a sensor can provide. Moreover, we defined a Domain Ontology able to describe each STT dimension discussed before. This work is the basis for the Semantic Virtualization phase that we will discuss in Chapter 5. In this phase we are able to semantically characterize the information produced by every sensor into the dimensions provided by our Syntactic Data Model

and we are able to generate transformation rules that generate instances and links in the Domain Ontology for the representation of the sensor and of its schema.

Chapter 5

Semantic Virtualization of Sensors

In a cross-domain context we assume the presence of different context brokers, one for each platform, that expose to external services the sensors and actuators that are public. However, the events observed by the sensors can be represented according to different formats (XML, JSON, CSV) and the associated semantics might not be expressively specified. In this context, there is the need to discover the presence of new sensors and to associate a semantics to the information they provide according to the Domain Ontology that we wish to adopt.

In this chapter we provide an approach that moves into this direction by introducing the "Semantic Virtualization" of sensors and the automatic translation of sensor data into our internal data model eventually labeled with concepts of the Domain Ontology. The Semantic Virtualization process is a complex activity that can require the interaction with the user and has the purpose to discover the presence of new sensors made available by a context broker, extract the sensor schema (i.e. the attributes that are generated by the sensor according to a given format), the semantic annotation of the sensor attributes with concepts of the Domain Ontology and the characterization of the sensor (and its schema) by means of instances of the Domain Ontology. We remark that both the semantic annotation and the semantic characterization of the sensor can be partial, because we do not assume it is always possible to associate a semantic annotation to the attributes contained in the sensor data model. The annotations (and also the semantic characterization) can be added by means of the data acquisition plan during the processing of the data generated by the sensors.

In other approaches such as the one proposed in [76], the semantic model of a structured data source is carried out by means of a mapping from the source itself to a Domain Ontology. In this approach, each attribute of the data source is properly mapped to a concept of the Domain Ontology. If the relationship between the attribute and an instance is not specified in a direct way, the corresponding relationship is expressed in terms of properties of the path can be inferred from the ontology. The main goal of this solution is to provide a complete mapping between the data source schema and the Domain Ontology concepts by trying to infer and, when necessary, disambiguate relationships between the attributes. The goal of our approach

is different. The semantic labelling, we propose aims at providing a partial description of the data source schema according to three dimensions: *spatial*, *temporal* and *thematic*. The final goal is not to provide a formal semantic model associated to the data source schema but to describe for each schema, the set of attributes that are necessary for combining the data flows coming from sensors according to the semantic of the themes they provide and aligning their temporal and spatial granularities.

The chapter is organized as follows. Section 5.1 discusses the Semantic Virtualization process. Specifically, it points out the sensor data model that can be determined by considering the format of the data produced by the sensor and a sample. Then, the different steps required for determining the sensor data model are discussed and finally the approach for associating a semantic labeling to each attribute in the sensor data model is discussed. Section 5.2 first introduces an algorithm for the semantic characterization of a sensor in the Domain Ontology. The algorithm has the purpose to create instances and links in the Domain Ontology for the representation of the sensor and of its schema. Since the characterization is partial, we then discuss the condition for considering it complete. With this aim, we introduce the concept of *consistency* and how it can be evaluated by considering the STT dimensions. Finally, in Section 5.3 we exploit the concepts presented in the chapter for the automatic creation of transformation rules that can be applied for the translation of sensor observations into our internal model eventually annotated with semantic concepts.

5.1 Semantic Discovery of Sensors

Context brokers are components of different IoT platforms that make available channels representing sensors producing different kinds of data according to different formats (XML, JSON, CSV). Sometimes these data can be coupled with meta information for simplifying their management and also their integration. However, this meta information can be heterogeneous in format and semantics and their management in a uniform way can be difficult. For this reason, we do not rely on them for the processing of data within our system.

In order to interact with the context broker, we assume that they provide a simple set of primitives that can be invoked by our system. These primitives allow us to realize the following services:

- Check the presence of a new sensor. This corresponds to the creation of a new channel in the context broker. The name of the created channel corresponds to the identifier of the sensor.
- Collect a sample of data produced by the sensor in its specific format.
- Read new observations from the sensor.

- Identify when the sensor is active (i.e. it produces observations that regularly appears in the context broker) or is inactive (i.e. for a certain amount of time, no data have been produced by the context broker, or the channel in the context broker has been removed).

Starting from that, we wish to create a process that is able to discover the presence of a new sensor by querying a context broker, extract its schema and provide a semantic annotation of the schema attributes with respect to the Domain Ontology. This semantic labeling of the information produced by a sensor is the first step towards the semantic characterization of the information produced by heterogeneous sensors made available by different context brokers and the automatic translation of the observations produced by the sensors in our internal data model. As discussed earlier, this labeling is only partial because we cannot assume that a Domain Ontology is always able to correctly represents all possible situations. Therefore, it is possible that attributes produced by a sensor cannot be directly labeled with concepts/properties of the Domain Ontology.

In the current stage of development, the user intervention is mandatory in order to produce meaningful semantic labeling. We are aware that when the number of sensors to be semantically labeled becomes quite high, the human effort can become too high. However, domain experts are the only people who have the adequate knowledge about the context in which these sensors are placed and can make proper decisions. Involving all these stakeholders in problem solving, providing them with opportunities to construct their own understanding and have control in the description of problems is necessary to foster successful solutions [46]. For this reason, we wish to maintain the interaction with the user and produce services that support his work from different points of view in order to maintain a great accuracy of the generated semantic labels. The final system is designed around an interaction strategy [33], [47], [107] able to support non-experts in computer science but experts in the domain of interest in describing the data source schema according to a specified Domain Ontology. That does not mean that in future we cannot integrate our solution with machine learning approaches for automatically suggesting to the user possible labeling for a given sensor. These suggestions will take into account previous classifications conducted by the user as well as by other users with similar characteristics, both positive and negative evaluation of the suggested classifications, and the context broker that makes available the data produced by the sensor.

In our setting, the context broker is an external service that can be simply queried. The sensor information along with the schema of the produced data, the semantic labeling and all the processes required for accessing and transforming the data generated from the sensors in our internal representation are maintained in our environment and stored in a internal database. The primitives reported in Table 5.1 are exploited for determining the format of a value produced by a sensor, for extracting

a sample, for generating the syntactic data model of the schema of a sensor, and for the application of a selector for extracting a component of the value.

In the remainder of the section, we first introduce the *Syntactic Data Model* for the representation of the sensor schema. This model is not yet compliant with the event stream data model discussed in Definition 4.2 of Chapter 4, but it is the first step towards such direction. Then, we present an algorithm for the extraction of the Syntactic Data Model from a new sensor that appears in a context broker. Finally, we discuss the semantic labeling approach used for annotating the attributes of the sensor schema.

5.1.1 Dealing with Different Formats

The goal is to produce data according to our internal data model presented in Chapter 4. However, data are produced according to different formats (CSV, XML, JSON) and there is the need to properly deal with these formats.

We need an intermediate data structure (denoted *Syntactic Data Model*) that is used for passing from the external data format of the sensor to our internal syntactic data model. We need to select attributes belonging to the data produced by sensors according to the different formats. In case of CSV format, the selector is simply the position in the list of values separated by a separator (a tab, a comma, or another symbol). We remark that we consider also structured values in the CSV that are list of values delimited by $\{ \}$ or $[]$ brackets. In case of XML format, the selector is a path expression for the identification of a single element/attribute or a set of elements/attributes. A position can be specified for selecting a specific element among those with the same tag identified from the current step of the path expression. In case of JSON format, a path expression (similar to the one for XML) can be specified that takes into account the record and array constructors made available by the language. The following definition formally presents the different kinds of selectors depending on the source format.

Definition 5.1 (*Selector*). Let v be a value produced by a sensor s according to a format f ($f \in \{XML, JSON, CSV\}$). A selector is an expression for identifying an attribute of the value v depending on the format f . A selector can be:

- $i[p]$, when $f = CSV$, $i \in \mathbb{N}$, $v = (v_1, \dots, v_n)$, $1 \leq i \leq n$, and p is a position that can be specified when v_i starts with $\{$ or $[$.
- x -path, when $f = XML$. x -path is a path expression in the hierarchical structure of an XML document that takes into account the presence of elements and attributes in the document; therefore x -path = $/a_1[p_1]/\dots/a_n[p_n]$, where $a_i \in \mathcal{A}$, $1 \leq i \leq n - 1$ is an element name in the XML document, a_n can be an element or attribute name (in the last case is represented as $@a_n$), and p_i is a position for distinguishing subelements

<p>A</p> <pre>{ "metadata": { "thematic": "temperature", "location": { "latitude": "45.464161", "longitude": "9.190336", "data": [{ "parameter": "timestamp", "dataType": "date", "value": "2016-05-15 12:10"}, { "parameter": "measurement", "dataType": "real", "value": 21.4}] } } }</pre>	<p>B</p> <pre><event> <temperature>71.24</temperature> <time>2016/05/15 12:10</time> </event> <event> <temperature>71.96</temperature> <time>2016/05/15 12:12</time> </event></pre>
<p>C</p> <pre>15-05-2016 13:08, {45.441524, 9.085241}, 'humidity', 32.3 15-05-2016 13:09, {45.441524, 9.085241}, 'humidity', 32.1 15-05-2016 13:10, {45.441524, 9.085241}, 'humidity', 31.9</pre>	<p>D</p> <pre>2016-05-15 12:00, z1, ['today is very hot', 'In my home there is much heat'], 2 2016-05-15 12:01, z1, ['a nice day for me', 'hot, hot, hot', 'I am walking'], 3 2016-05-15 12:20, z2, ['I am sweat', 'My, It's hot in here'], 2</pre>

FIGURE 5.1: Formats of the events generated by the sensors in the zones of Milano

with the same name (p_i can be omitted when subelements are distinct or a set of values should be retrieved).

- j -path, when $f = \text{JSON}$. j -path is a path expression in the hierarchical structure of a JSON document that takes into account the presence of set values associated with object properties; therefore, j -path = $/a_1[p_1]/\dots/a_n[p_n]$, where $a_i \in \mathcal{A}$, $1 \leq i \leq n$ is an object property name, and p_i is a position for accessing the i -esime component of an array value (for non-array values p_i can be omitted). \square

Example 5.1 For the sake of readability, Figure 5.1 reports the formats of our running example presented in the Introduction. The followings are examples of selectors for these values:

- $/\text{metadata}/\text{location}/\text{latitude}$, $/\text{data}/\text{dataType}$, $/\text{data}/\text{value}[1]$ are examples of path expressions for the JSON value in Figure 5.1 a). In the first and third case a single value is identified, whereas in the second case an array of values is identified.
- $/\text{event}/\text{temperature}$, $/\text{event}/\text{time}$ are examples of path expressions for the XML value in Figure 5.1 b).
- 1, 2 are examples of selectors for the CSV value in Figure 5.1 c). In the first case a single value is identified, whereas in the second case an array of values is identified. \square

A selector is thus able to identify a component of a value v and to return a pair (a_i, v_i) , where v_i is the selected component (either a single value of a set of values) and a_i is the associated label. The label is the name of the element/attribute in case

of XML, the name of the property in case of JSON, and is a system-generated label in case of CSV.

Example 5.2 Consider the selector of previous example, the following pairs can be returned by their application:

- $(\text{latitude}, 45.464161), (\text{dataType}, [“date”, “real”]), (\text{value}, “2016 – 05 – 1512 : 10”)$ are the values returned in the first case.
- $(\text{temperature}, 71.24), (\text{time}, “2016/05/1512:10”)$ are the values returned in the second case.
- $(\text{csv}_1, “15-05-201613:08”), (\text{csv}_2, [45.441524, 9.085241])$ are the values returned in the third case. □

Relying on a set of values on which the selector can be applied, it is possible to determine the basic domain (or a list of basic domain) that represents them. When the structure complexity of the values generated by sensor is limited, the selector can be automatically determined. For the general case, user interaction is required. In the remainder we consider cases in which the selector can be automatically determined.

Relying on the definition of selector, we introduce an intermediate abstract representation of the sensor schema. This representation points out the structure of the value generated by the sensor that is independent from the adopted format. With this representation we are also able to handle all the formats that can be translated (by means of a wrapper) to this format.

Definition 5.2 (*Syntactic Data Model*). A sensor s produces data according to the format f ($f \in \{\text{XML}, \text{JSON}, \text{CSV}\}$). Let \mathcal{A} be a set of labels, and D_i be basic domains or an array of values of basic domains, like *numeric*, *string*, *date*, *boolean*. The Syntactic Data Model of s is an abstract representation of the schema of the data that it produces and is represented as a 3-tuple:

$$\langle f, \text{status}, \{ \langle s_1, a_1, D_1 \rangle, \dots, \langle s_n, a_n, D_n \rangle \} \rangle$$

where: s_1, \dots, s_n are selectors according to Definition 5.1, $\{a_1, \dots, a_n\} \subseteq \mathcal{A}$ and $\text{status} \in \{\text{active}, \text{inactive}\}$. □

Example 5.3 The syntactic data model of the JSON value in Figure 5.1 a) is

$$\langle \text{JSON}, \text{active}, \{ \langle /metadata/thematic, thematic, string \rangle, \langle /metadata/location/latitude, latitude, numeric \rangle, \langle /metadata/location/longitude, longitude, numeric \rangle, \langle /data/parameter/value, timestamp, numeric \rangle, \langle /data/parameter/value, measurement, numeric \rangle, \} \rangle$$

The syntactic data model of the XML value in Figure 5.1 b) is

$$\langle \text{XML}, \text{active}, \{ \langle /event/temperature, temperature, numeric \rangle, \langle /event/time, time, timestamp \rangle \} \rangle$$

Notation	Meaning
$\text{format}(s)$	Given a sensor s , it returns the format (either XML, JSON, CSV) of the produced data
$\text{sample}_B(s)$	Given a sensor s , it returns a sample of the data produced by s and made available by the context broker B
$\text{schema}(s, f, \text{sample}_B(s))$	Given a sensor s that produce data in the format f and a sample of the produced data, it returns its syntactic data model
$\text{extract}_f^{\text{sel}}(v)$	Given a value v specified according to the format f , it extracts a value component according to the selector sel in a JSON format

TABLE 5.1: Notations

The syntactic data model of the CSV value in Figure 5.1 c) is

$$\langle \text{CSV, active}, \{ \langle 1, \text{csv_1, timestamp} \rangle, \langle 2[1], \text{csv_2, numeric} \rangle, \langle 2[2], \text{csv_3, numeric} \rangle, \langle 3, \text{csv_4, string} \rangle, \langle 4, \text{csv_5, numeric} \rangle \} \rangle$$

The syntactic data model of the CSV value in Figure 5.1 d) is

$$\langle \text{CSV, active}, \{ \langle 1, \text{csv_1, timestamp} \rangle, \langle 2, \text{csv_2, string} \rangle, \langle 3, \text{csv_3, list(string)} \rangle, \langle 4, \text{csv_4, numeric} \rangle \} \rangle \square$$

5.1.2 The Sensor Discovery Algorithm

The Syntactic Data Model presented in previous section allows the association of an abstract representation to the schema of data produced by a sensor in an automatically (or semi-automatically) way depending on the complexity of the formats used for the representation of the sensor data. We now present the `SensorDiscovery Algorithm 1` which is a monitor associated with a given context broker that checks the introduction (or removal) of sensors and keeps updated the list of current available sensors.

The algorithm exploits the primitives reported in Table 5.1 for determining the format of a value produced by a sensor, for extracting a sample, for generating the Syntactic Data Model of the schema of a sensor, and for the application of a selector for extracting a component of the value.

The algorithm is a continuous procedure that, at a given instant of time, checks whether in the context broker a new sensor appears (i.e. a new channel is activated in the context broker). When this happens two cases need to be handled. In the first case, the sensor is a new one (its identifier is not registered in our database). Therefore, the algorithm extracts its format, collects a sample of data, and generate the Syntactic Data Model for this sensor. Then, the sensor and its Syntactic Data Model are stored in our database and the sensor is marked active. In the second case, the sensor identifier is already present in our database. Therefore, the algorithm generates a Syntactic Data Model that is compared with the one already stored in the

Algorithm 1 The SensorDiscovery Algorithm

```

1: Let  $S$  be the set of known sensors
2: Let  $SX$  be the sensor schema associated with  $S$ 
3: Let  $B$  be a context broker made available by the cross-domain platform
4: while (true) do
5:   if (a new sensor  $s$  appears in  $B$  and  $s \notin S$ ) then
6:      $f = \text{format}(s)$ 
7:      $sx = \text{schema}(s, f, \text{sample}_B(s))$ 
8:      $S = S \cup \{s\}$ 
9:     Mark  $s$  as active
10:     $SX = SX \cup \{(s, sx)\}$ 
11:   end if
12:   if (a sensor  $s$  disappears from  $B$  and  $s \in S$ ) then
13:     Mark  $s$  as inactive
14:   end if
15:   if (a new sensor  $s$  appears in  $B$  and  $s \in S$ ) then
16:      $f = \text{format}(s)$ 
17:      $sx' = \text{schema}(s, f, \text{sample}_B(s))$ 
18:     Let  $sx \in SX$  the schema of  $s$ 
19:     if  $sx \neq sx'$  then
20:        $SX = SX \cup \{(s, sx')\} \setminus \{(s, sx)\}$ 
21:     end if
22:     Mark  $s$  as active
23:   end if
24: end while

```

database. If there are differences the new one is updated in the database in order to maintain fresh data in the database.

The algorithm also considers the possibility that a sensor is removed from the context broker. This can happen when the sensor is deactivated in its domain or its data are not any longer accessible from our applicative context. In this case the algorithm marks the sensor as inactive. Its information (sensor identifier and Syntactic Data Model) are kept in the database for future references. Periodically, sensors that are inactive for a given period of time are removed from the database.

We have to remark that in principle the syntactic data model is automatically generated. Therefore, no semantic information (in term of the STT dimensions) can be directly associated with it. However, by taking into account previous semantic labeling conducted by the user on similar data, it is possible to classify the sensor attributes in one of the three STT dimensions in order to reduce the effort required to the user in the semantic labeling process. It is an implementation of the Publish/-Subscribe that will provide different interfaces to perform several operations. This operation includes sensor registration, update sensor information, notify context information, and query context information. The context broker runs as a cluster on one or more servers that can span multiple data stores. The cluster stores streams of records in categories called topics.

5.1.3 Semantic Labeling

Given an ontology \mathcal{O} , we denote with $Class(\mathcal{O})$ the set of classes/entities specified within \mathcal{O} , with $Property$ the properties associated with classes in $Class(\mathcal{O})$, with $Property(C)$ the properties associated with a specific class $C \in Class(\mathcal{O})$, and with

\mathcal{I} the set of instances of the ontology \mathcal{O} . In order to simplify the presentation we denote with $C::p$ a specific property belonging to $Property(C)$ and with $subClasses(C)$ the set of subclasses (direct or indirect) of class C .

In our Domain Ontology, classes have been classified according to the STT dimensions. We thus use the notation $Class(\mathcal{O})|_d$ with $d \in \{\text{time, space, thematic}\}$ to identify the classes belonging to a specific dimension (we adopt the notation also for the class properties). This classification is particularly useful for showing to the user only the classes that can be used for each dimension and thus reducing the possibility of mistakes. Moreover, given an attribute a_i of the syntactic data model associated with a sensor s , we denote with $Dim(a_i) \in \{\text{time, space, thematic}\}$ at which dimension the user has deemed more accurate to classify this attribute (the default value is `thematic`).

As said in the previous section, we consider domain experts as the only people who have the adequate knowledge about the context in which these sensors are placed and about the real meaning of the data schemas in order to make proper decisions. For this reason, in our solution the user is in charge of generating the semantic labels to maintain a great accuracy of the semantic description process. Relying on these notations, we can introduce the concept of semantic labeling in our context. A semantic labeling is a partial function that associates the sensor schema attributes to concepts of the Domain Ontology. At the current stage, this function is specified by the user by means of a graphical interface that will be described in Chapter 7. The Web interface is a good means for supporting the user in this activity and in future we are planning to integrate the user's interaction with machine learning algorithms able to support him in the identification of concepts/properties and instances of the Domain Ontology to be associated with the syntactic schema of the sensor by taking into account the specific features of sensor data and the peculiarity of the adopted Domain Ontology.

The function is created by the user for steps depending on the STT dimension on which the user classifies an attribute in the sensor schema. When the attribute represents a temporal information, the class representing the instance of time when the measurement has been acquired is used to label the attribute. Since we usually adopt the Gregorian calendar, the class is `tl:Instant`. However, in case also other calendars are used, specific classes for representing them can be shown. When, the attribute represents a spatial dimension, either a class or a class property can be associated with the sensor attribute. This depends on the spatial granularity that is selected by the user. The labeling for an attribute of the thematic dimension is a tuple with two components. The first one is a class C among the subclasses of `ssn:MeasurementProperty` and those of `iot:Metadata` used for representing the properties of a thematic in the Domain Ontology (which is an instance of $\mathcal{I}(\text{qu:QuantityKind})$). The second component is optional and represents the unit of

measure according to the measurement has been acquired (for example in case of a temperature, the unit of measure can be Celsius).

We remark that in our context, it might happen that the concept represented by a sensor attribute is not included in the Domain Ontology. Therefore, a semantic labeling cannot be associated with such sensor attribute. However, it is possible to associate a label (eventually the same or a new one – named in the following a'_i) to the attribute when we wish to maintain it. Attributes in the sensor schema that do not belong to the domain of the semantic labeling function are dropped and no longer considered. This can be useful, when the sensor produces many information, but we consider relevant for the analysis that we have to conduct only a subset of them. The following definition formally presents the semantic labeling function.

Definition 5.3 (*Semantic Labeling*). Let $\langle f, status, \{\langle s_1, a_1, D_1 \rangle, \dots, \langle s_n, a_n, D_n \rangle\} \rangle$ be the Syntactic Data Model associated with a sensor s and \mathcal{O} be our Domain Ontology. A semantic labeling $\mathcal{L}abel$ is a partial function

$$\mathcal{L}abel : \{a_1, \dots, a_n\} \rightarrow ((Class(\mathcal{O}) \cup Property) \times \mathcal{I}(qu:Unit)) \cup \mathcal{A}$$

such that

$$\mathcal{L}abel(a_i) = \begin{cases} (C, unit) & \text{if } Dim(a_i) \in \{\text{time, space}\}, C \in Class(\mathcal{O})_{|Dim(a_i)}, \\ & unit \in \mathcal{I}(qu:Unit) \cup \{\perp\} \\ (C::p, unit) & \text{if } Dim(a_i) = \text{space}, C::p \in Property(C)_{|space}, \\ & unit \in \mathcal{I}(qu:Unit) \cup \{\perp\} \\ (C, unit) & \text{if } Dim(a_i) = \text{thematic}, unit \in \mathcal{I}(qu:Unit) \cup \{\perp\}, \\ & C \in subClasses(ssn:MeasurementProperty) \cup \\ & \quad \cup \{\text{iot:Metadata}\} \\ a'_i & \text{otherwise} \end{cases} \quad \square$$

Example 5.4 Consider the syntactic data model associated with the CSV value in Figure 5.1 c) in Example 5.3. The following semantic labeling function can be associated with it:

$$\begin{aligned} \mathcal{L}abel(csv_1) &= (tl:instant, \perp) \\ \mathcal{L}abel(csv_2) &= (geo:long, degree) \\ \mathcal{L}abel(csv_3) &= (geo:lat, degree) \\ \mathcal{L}abel(csv_5) &= (iot:Metadata, percentage) \end{aligned} \quad \square$$

The semantic labeling function is used for labeling the attributes of the sensor schema to concepts of the Domain Ontology. By means of the Web interface, it is also possible to associate further information that is used to better characterize the sensor itself and the values that it produces. The following definition details the meaning of such properties and characterizes the description of the sensor at the ontology level.

Operation	Meaning
<code>i=new("CLASS")</code>	Create a new instance of the class CLASS
<code>i.addLink(j,"REL")</code>	Include a link between <i>i</i> and <i>j</i> instance of the relationship REL

TABLE 5.2: Primitives for the modification of ontology instances

Definition 5.4 (*Sensor Descriptor*). Let *s* be a sensor, by means of the Web interface, the tuple $\langle t\text{-gran}, s\text{-gran}, \text{theme}, t\text{-start}, t\text{-end} \rangle$ can be associated by the user to *s*, where:

- *t-gran* is a pair of values (unit, num) such that $\text{unit} \in \mathcal{I}(\text{qu:Unit})$, and $\text{num} \in \mathbb{N}$, representing the number of unit of times according to which data are gathered from the sensor (i.e. the temporal granularity), when it is specified (\perp otherwise).
- *s-gran* is a class belonging to $\{\text{geo:Point}\} \cup \text{subClasses}(\text{iot-lite:Entity})$ the spatial granularity a measurement refers to, when it is specified (\perp otherwise).
- *theme* is an instance of the class `qu:QuantityKind`.
- *t-start* and *t-end* are two instants of time delimiting the interval of time in which the sensor produces observations, when they are specified (\perp otherwise). \square

Example 5.5 Consider the Syntactic Data Model associated with the CSV value in Figure 5.1 c) in Example 5.3. The following information is associated with the corresponding sensor.

- $t\text{-gran} = (\text{minute}, 10)$;
- $s\text{-gran} = \text{geo:Point}$;
- $\text{theme} = \text{humidity}$;
- $t\text{-start} = 2018\text{-}01\text{-}01\text{T}00:00:00.000$;
- $t\text{-end} = 2019\text{-}01\text{-}01\text{T}00:00:00.000$. \square

5.2 Evaluation of Sensor Consistency

In the previous section we outlined the Semantic Virtualization process according to which when sensors are registered in our system, the sensors' themselves and their schema are annotated with concepts of the Domain Ontology.

In this section we first exploit the annotations for the semantic characterization of the sensors. With this aim, we introduce an algorithm that automatically generates instances of the Domain Ontology for the representation of the sensor and of its schema. Instances are generated only when the required classes are present in the Domain Ontology. Otherwise, it means that the concept contained in the sensor schema is not covered by the ontology, that is the expressive power of the ontology is not sufficient for representing it. This situation can happen for two kinds of reasons. First, the Domain Ontology designers forgot to include a concept that is required in the domain. This kind of problem can be solved by evolving the Domain Ontology

and introduce missing concepts. The second reason is that the Domain Ontology designers considered such attributes illegal in the domain. Therefore, it is better to not consider such attributes.

Once the sensors are semantically characterized, there is the need to check their *consistency* with respect to the Domain Ontology. In the second part of this section we propose the concept of consistency and define it according to the STT dimensions exploited for the representation of the sensor schema.

5.2.1 Semantic Characterization of Sensors

By means of the information generated by Algorithm 1 (*SensorDiscovery*), the semantic labeling function discussed in Definition 5.3, and the sensor descriptor described in Definition 5.4, we are now able to create a semantic characterization of the sensor at the ontology level. This characterization is used for providing a uniform description of the sensors that are handled by our system, for simplifying their discovery when a Data Acquisition Plan needs to be formulated, and also for assessing the consistency of a data acquisition plan with respect to the adopted Domain Ontology.

The semantic characterization consists in creating an instance for a new sensor s in the Domain Ontology and to provide values that represent the STT dimensions specified for its schema. The basic operations reported in Table 5.2 are used for this purpose that allow us to create instances of a class and include links between pairs of instances.

Algorithm 2 (*SensorSemanticCharacterization*) provides the semantic characterization as follows. First, an instance of class `ssn:SensingDevice` is created for the representation of the new sensor and, by exploiting the information contained in the sensor description, properties are specified for representing the temporal granularity, the spatial granularity, the thematic, and the interval of time in which observations are generated by the sensor (from line 1 to line 16). Then, for each attribute a in the syntactic data model for which a semantic labeling has been specified, a representation of the attribute in the Domain Ontology is provided according to the dimension in which the attribute has been classified. Specifically, if a is a temporal information (representing an instance of time), a link named `hasTime` is created between the sensor and an instance of the class `tl:Instant` (from line 19 to 22). If a is a spatial information, it can represent a coordinate (e.g. latitude, longitude, etc.) or a coarser granularity (e.g. zone, city, region, etc.). A specific instance is thus created and linked to the instance representing the sensor through the property `hasLocation` (from line 23 to 33). If a is a thematic information, it can represent a measurement property (for representing observable characteristics of sensor's events) or a metadata (for representing the type of the data coming from the sensor). An instance of this class is created and linked with the sensor (from line 34 to 42). Finally, when

Algorithm 2 The SensorSemanticCharacterization Algorithm

Require: the sensor s ,
 the syntactic data model $\langle f, status, \{ \langle s_1, a_1, D_1 \rangle, \dots, \langle s_n, a_n, D_n \rangle \} \rangle$ of s
 the sensor descriptor $\langle t\text{-gran}, s\text{-gran}, theme, t\text{-start}, t\text{-end} \rangle$
 the semantic labeling function $Label$ on the attributes $\{a_1, \dots, a_n\}$ of the schema of sensor s .

```

1:  $s_1 = \text{new}(\text{"ssn:SensingDevice"})$ 
2: /* Temporal Granularity */
3:  $t_G = \text{new}(\text{"ssn:ResponseTime"})$ 
4:  $s_1.\text{addLink}(t_G, \text{"ssn:MeasurementProperty"})$ 
5:  $t_G.\text{addLink}(t\text{-gran}.unit, \text{"hasUnit"})$ 
6:  $t_G.\text{addLink}(t\text{-gran}.num, \text{"hasDataValue"})$ 
7: /* Spatial Granularity */
8:  $s_G = \text{new}(\text{"s-gran"})$ 
9:  $s_1.\text{addLink}(s_G, \text{"hasLocation"})$ 
10: /* Thematic Granularity */
11:  $s_1.\text{addLink}(theme, \text{"hasQuantityKind"})$ 
12: /* Validity Interval */
13:  $interval = \text{new}(\text{"properInterval"})$ 
14:  $s_1.\text{addLink}(interval, \text{"hasTime"})$ 
15:  $interval.\text{addLink}(t\text{-start}, \text{"interval starts"})$ 
16:  $interval.\text{addLink}(t\text{-end}, \text{"interval finishes"})$ 
17: /* Representation of Sensor Attributes */
18: for each  $a \in \{a_1, \dots, a_n\}$  s.t.  $Label(a)$  is defined and  $Label(a) = (x, unit)$  do
19:   if  $Dim(a) = \text{time}$  then
20:      $ts = \text{new}(\text{"tl:Instant"})$ 
21:      $s_1.\text{addLink}(ts, \text{"hasTime"})$ 
22:   end if
23:   if  $Dim(a) = \text{space}$  then
24:     if  $x = C::p$  then
25:        $pt = \text{new}(\text{"C"})$ 
26:        $s_1.\text{addLink}(pt, \text{"hasLocation"})$ 
27:        $coord = \text{new}(\text{"p"})$ 
28:        $pt.\text{addLink}(coord, \text{"p"})$ 
29:     else
30:        $sp = \text{new}(\text{"x"})$ 
31:        $s_1.\text{addLink}(sp, \text{"hasLocation"})$ 
32:     end if
33:   end if
34:   if  $Dim(a) = \text{thematic}$  then
35:     if  $x = \text{iot:Metadata}$  then
36:        $th_1 = \text{new}(\text{"iot:Metadata"})$ 
37:        $s_1.\text{addLink}(th_1, \text{"hasMetadata"})$ 
38:     else
39:        $th_1 = \text{new}(\text{"x"})$ 
40:        $s_1.\text{addLink}(th_1, \text{"hasMeasurementProperty"})$ 
41:     end if
42:   end if
43:   if  $unit \neq \perp$  then
44:      $s_1.\text{addLink}(unit, \text{"hasUnit"})$ 
45:   end if
46: end for

```

present, the information about the unit of measurement is associated with the sensor (from line 43 to 45).

5.2.2 Consistency of Sensor Schema w.r.t. the Domain Ontology

In the previous section, we have seen an algorithm for the semantic characterization of a new sensor with respect to the Domain Ontology. In this section we wish to introduce a mechanism for the verification of the consistency of its data model. This verification is required because of the flexibility of our approach that does not impose a complete match between the event stream $\mathcal{M} = \langle G_T, G_S, th, [t_s, t_e], S, Event_{(G_T, G_S)}^{th}(\tau) \rangle$ associated with a sensor according to Definition 4.2 and the semantic characterization discussed in previous section. In order to guarantee the consistency w.r.t. a given Domain Ontology \mathcal{O} , we need to verify the exact match of its spatial, temporal and thematic dimensions with the related concepts expressed in \mathcal{O} .

We remark that a Domain Ontology contains a set of classes and relationships considered valid by the experts of such domain. For what concern the thematic dimension, this means that, in order to be consistent, the properties specified in the streams should be conceptualized as classes and links in advance in the Domain Ontology by the experts. That is, the experts require that sensors of type T_1 for being consistent need to present a measurement property `ssn:Accuracy`, and that sensors of type TW_1 need to be connected with an instance of the class `TwitterProp` that contains the total number of tweets and positive tweets.

A sensor is "not consistent" w.r.t. the Domain Ontology if it does not provide all the spatial, temporal and thematic dimensions according to the following definitions.

Definition 5.5 (*Spatial Consistency*). \mathcal{M} is spatially consistent w.r.t. \mathcal{O} if:

1. $\forall s \in S$, an instance of the class `iot-lite:Entity` \sqsubseteq `iot-lite:Coverage` corresponding to s exists at the granularity G_S ;
2. G_S is guaranteed by the unary association `isPartOf` defined on one of the subclasses of the class `iot-lite:Entity` or by using the association `hasSystemReference` between the class `iot-lite:Entity` and the class `GeoSubdivision-Standard` (used for describing Standard Geographical Administrative Subdivisions);
3. the instance of one of the subclasses of the class `iot-lite:Entity` is linked to an instance of one of the classes `iot-lite:Rectangle`, `iot-lite:Polygon`, and `iot-lite:Circle` for specifying the coverage area. \square

Example 5.6 Consider the situation described in Example 4.3 and depicted in Figure 4.4. A sensor of type T_1 is spatially consistent because its location is described by an instance of the class `do:zone` subclass of the class `iot-lite:Entity` that is a part of the city Milano (instance of the class `do:city`) that in turn is part of the region Lombardia (instance of

the class `do:region`) thus fulfilling the relation of order defined by the granularity G_S . Moreover, the instance `ZoneT1` is linked to an instance of the class `iot-lite:Circle`. \square

Definition 5.6 (Temporal Consistency). \mathcal{M} is temporally consistent w.r.t. \mathcal{O} if:

1. the timestamp of the data gathered from the sensor is described by means of an instance of the class `tl:Instant`;
2. $[t_s, t_e]$ is described by means of an instance of the class `tl:Proper Interval` where t_s and t_e are specified by using the properties `Interval starts` and `Interval finishes`;
3. $\forall t \in T$, an instance of the class `ssn:ResponseTime` exists associated with the instance of the sensor that produces the stream at the granularity G_T ;
4. G_T is specified through a link between the instance of the class `ssn:ResponseTime` with an instance of the class `qu:Unit` for specifying the unit of measure and by the subsequential link to the instance of the class `qu:SystemOfUnits` that represents the concept of “system of units”. \square

Example 5.7 Consider the situation described in Example 4.4 and Example 4.5 and depicted in Figure 4.5. A sensor of type T_1 is temporally consistent because the timestamp of the data coming from the sensor is described by an instance of the class `tl:Instant`. Then, the sensor is linked to a temporal interval (instance of the class `tl:Proper Interval`) that states that this sensor gathers events during a specified interval of time. Moreover, the sensor is also linked to an instance of the class `ssn:ResponseTime` that specifies that the temperature is acquired once every 20 minutes. The instance of the class `qu:Unit` is used for specifying the time granularity `Minute`. \square

Definition 5.7 (Thematic Consistency). Let $\mathcal{P}rop(o)$ and $Gran_T(o) / Gran_S(o)$ denote the set of properties occurring in o and the temporal/spatial granularity of o , respectively. Consider an instance $i_{th} \in \mathcal{I}(qu:QuantityKind)$ corresponding to th . \mathcal{M} is thematically consistent w.r.t. \mathcal{O} if:

1. a link $(i_{th}, hasQuantityKind, s)$ exists in \mathcal{O} s.t. $s \in \mathcal{I}(ssn:SensingDevice)$ corresponds to the sensor that produces the events; $i_{th} \in \mathcal{I}(qu:QuantityKind)$ corresponds to the sensor thematic, and the two instances are linked by the `hasQuantityKind` association;
2. the data type associated to the sensor s is described by means of an instance of the class `iot:Metadata` which is in turn associated at an instance of the class `DUL:Amount` for expressing its domain;
3. $\forall a \in \mathcal{P}rop(\tau)$, a link $(s, hasMeasurementProperty, i_a)$ exists in \mathcal{O} s.t. $i_a \in \mathcal{I}(a)$ and $a \sqsubseteq ssn:Property$;
4. $\forall i_a \in \mathcal{I}(a)$ s.t. a link $(s, hasMeasurementProperty, i_a)$ exists, $a \in \mathcal{P}rop(\tau)$. \square

Example 5.8 Consider the situation depicted in Figure 4.6 and in Figure 4.7. T_1 and TW_1 are thematically consistent w.r.t. the adopted Domain Ontology because these instances are

linked to instances of the `qu:QuantityKind` class for representing their thematic, they are linked to instances of the class `iot:Metadata` for modeling the data type and in turn are linked to instances of the class `DUL:Amount` for describing the corresponding domains. Moreover, they provide a complete semantic description of the properties as specified in the Domain Ontology. In fact, T_1 and TW_1 present the connections with proper measurement properties `ssn:Accuracy` and `do:NumTweets` and `do:NumPosTweets`. \square

Finally, we can specify that an event stream is consistent only if it is temporally, spatially and thematically consistent according to the adopted Domain Ontology.

Definition 5.8 (Consistent Specification). \mathcal{M} is consistent if it is temporally, spatially and thematically consistent w.r.t. \mathcal{O} . \mathcal{M} is partially consistent if at least one of the dimensions is consistent. \square

Example 5.9 Let \mathcal{O} be the Domain Ontology. Sensors of type H_1 are consistent w.r.t. \mathcal{O} . Sensors of type T_1 are not thematic consistent w.r.t. \mathcal{O} because they lack the accuracy. Sensors of type T_2 are not spatially consistent w.r.t. \mathcal{O} because the spatial coordinates are missing. Sensors of type TW_1 are temporally and spatially consistent w.r.t. \mathcal{O} , but thematically inconsistent because they lack the property about the number of positive tweets, as discussed in previous examples. \square

5.3 Automatic Transformation of Sensor Events

Starting from the definitions introduced in previous section, we are now able to automatically generate a set of translation rules that are able to translate the value generated by a sensor in its own format, in our formal specification of Event Stream (see Definition 4.2 in Chapter 4). This representation is also labeled with the concepts occurring in the Domain Ontology (by means of the semantic labeling function previously described).

Definition 5.9 (Set of Transformation Rule). Let $\langle f, status, \{\langle s_1, a_1, D_1 \rangle, \dots, \langle s_n, a_n, D_n \rangle\} \rangle$ be the Syntactic Data Model associated with a sensor s , and $\mathcal{L}abel$ be the semantic labeling function associated with the attributes of the schema of sensor s . A set of transformation rules \mathcal{TR} can be automatically created starting from $\mathcal{L}abel$. The cardinality of \mathcal{TR} is equal to the cardinality of the domain of $\mathcal{L}abel$. Each transformation rule r_i , where a_i is an attribute for which $\mathcal{L}abel(a_i)$ is defined, is a tuple:

$$\langle sel_i, dim_i, name_i, type_i, unit_i \rangle$$

where:

- sel_i is the selector associated with an attribute a_i .
- $dim_i = Dim(a_i)$ is the dimension in which the attribute a_i has been classified (in case the dimension is not specified, it is assumed to be `thematic`).

- if $\mathcal{L}abel(a_i) = (x, unit)$, with $x \in \{C::p, C\}$, then $name_i = x$, $type_i = D_i$, and $unit_i = unit$. if $\mathcal{L}abel(a_i) = a'_i$, then $name_i = a'_i$, $type_i = D_i$, and $unit_i = \perp$. \square

Example 5.10 Consider the semantic labeling function described in Example 5.4. The following set of transformation rules are generated:

$$\begin{aligned} &\langle 1, \text{time}, \text{tl:instant}, \text{timestamp}, \perp \rangle \\ &\langle 2[1], \text{space}, \text{geo:long}, \text{numeric}, \text{degree} \rangle \\ &\langle 2[2], \text{space}, \text{geo:lat}, \text{numeric}, \text{degree} \rangle \\ &\langle 4, \text{thematic}, \text{iot:Metadata}, \text{numeric}, \text{percentage} \rangle \end{aligned} \quad \square$$

At this point we can present the algorithm used for transforming the value produced by the sensor and made available by means of the context broker in our internal representation. Algorithm 3 contains the pseudo-code.

The algorithm takes as input the set of transformation rules associated with the attributes belonging to a sensor s , the information specified by the user for the sensor s , and the context broker B . The algorithm checks for a new value v made available in the context broker B for the sensor (channel) s . When the value v is detected, each transformation rule is applied on v for extracting a component v_i and associate it to a set of pairs (*Time*, *Space*, and *Thematic*) that depends on the dimension on which v_i has been classified. The name to be associated with the value v_i as well as its classification dimension is reported in the the tranformation rule adopted. Once this process is concluded, a record is generated with these sets (the function `rec` is used for generating a record from a set of pairs). Moreover, the STT dimensions, the interval of time in which values are acquired by the sensor, and the possible values for the spatial dimension are retrieved from the information associated with the sensor s . All these data are used for the generation of the 6-tuple used for representing a value in the Event Stream model.

Theorem 5.1 Let \mathcal{TR} be the set of transformation rules generated according to Definition 5.9 for a sensor s . The value v , generated by a sensor s according to Algorithm 3, is an event stream according to Definition 4.2.

This theorem can be easily proved for induction on the structure of the possible values that can be generated for the Event Stream.

The proposed algorithm can be applied at different points of the data acquisition process. It can be applied:

- directly in the sensor or in the gateway in charge of handling the sensor;
- when the observation is posted on the context broker;
- in the cloud, before its processing.

Each one of these solutions presents advantages and disadvantages.

Algorithm 3 The SensorAcquisitionValues Algorithm

Require: the sensor s ,
the context broker B ,
the sensor descriptor specified by the user: $\langle t\text{-gran}, s\text{-gran}, theme, t\text{-start}, t\text{-end} \rangle$
the set of transformation rules \mathcal{TR} .

- 1: **while** (**true**) **do**
- 2: $Time = \emptyset$
- 3: $Space = \emptyset$
- 4: $Thematic = \emptyset$
- 5: Let v be a new value occurring in B for s
- 6: **for each** $r_i = \langle sel_i, dim_i, name_i, type_i, unit_i \rangle \in \mathcal{TR}$ **do**
- 7: $(a_i, v_i) = \text{extract}_{format(v)}^{sel_i}(v)$
- 8: **if** $dim_i = \text{time}$ **then**
- 9: $Time = Time \cup \{(name_i, v_i)\}$
- 10: **end if**
- 11: **if** $dim_i = \text{space}$ **then**
- 12: $Space = Space \cup \{(name_i, v_i)\}$
- 13: **end if**
- 14: **if** $dim_i = \text{thematic}$ **then**
- 15: $Thematic = Thematic \cup \{(name_i, v_i)\}$
- 16: **end if**
- 17: **end for**
- 18: $G_T = t\text{-gran}$
- 19: $G_S = s\text{-gran}$
- 20: $th = theme$
- 21: $S = \{inst \mid inst \in \mathcal{I}(G_S)\}$
- 22: **return** $\langle G_T, G_S, th, [t\text{-start}, t\text{-end}], S, (\text{rec}(Time), \text{rec}(Space), \text{rec}(Thematic)) \rangle$
- 23: **end while**

The first solution has the advantage that the observation is directly communicated in the internal format when it is produced. However, the algorithm should be implemented taking into account the operating system that works on the sensor/gateway and requires a deep control on the sensor that, in our cross-domain context, happens rarely. The second solution can be applied on the edge of the network when the observations are transferred from the sensors to the Cloud where they can be processed. The advantage of this solution is that transformations can be executed locally to where the data are produced, but it cannot take advantage of the processing power that the Cloud can offer. The last option has the advantage of taking advantage of the use of a scalable architecture as the one of Cloud, but the quantity of data collected by heterogeneous sensors can have negative effects on the quantity of observations to be handled in each second.

Chapter 6

Sound and Consistent Data Acquisition Plans

In the previous chapter we presented the services developed in order to transform sensor data in a common format and how is possible to semantically describe them in order to understand their meaning.

In this chapter we propose a set of basic services that can be exploited for manipulating the data generated by the sensor. These services allow to filter, aggregate, join, union the data generated by sensors. For aggregation we develop a specific operator to change the spatio-temporal granularities of data. The second ones can occur more often in our context and are easier to specify and understand. Moreover, sensor data can be enriched with meta information contained in the Domain Ontology or available in external databases. A trigger service is also included for activating/deactivating a stream relying on a condition on the values assumed by another stream. Finally, virtual attributes can be generated relying on a formula specified by the user and transformation functions can be applied for changing for example format, unit of measures, cases of the values generated by the sensors.

The proposed services can be combined in a Data Acquisition Plan (DAP) which is a direct acyclic graph presenting many sources and a single destination. Conditions for expressing the soundness of the DAP (i.e. when the DAP can be executed without errors on the sensor data) and consistency (e.g. when the output of the DAP is well described by the adopted Domain Ontology) are proposed along with an algorithm for the semantic characterization of the entire DAP and the evaluation of its consistency with respect to the Domain Ontology.

In the remainder of the chapter we discuss the available services with their applicability conditions in Section 6.1, then, in Section 6.2 a formal definition of Data Acquisition Plan is provided. Definitions of *sound* and *consistent* Data Acquisition Plan are presented in Section 6.3 and, finally, on Section 6.4 we describe how consistency of a DAP is checked with respect to our Domain Ontology.

<i>Cond</i>	::=	<i>BasicCond</i> <i>Cond</i> "and" <i>Cond</i> <i>Cond</i> "or" <i>Cond</i> "not(" <i>Cond</i> ")"
<i>BasicCond</i>	::=	<i>ValCond</i> <i>ExistsCond</i> <i>SizeCond</i> <i>TypeCond</i>
<i>ValCond</i>	::=	<i>Path</i> <i>CompOp</i> <i>Val</i> <i>Val</i> <i>SetOp</i> <i>Path</i>
<i>ExistsCond</i>	::=	"exists(" <i>Path</i> ")"
<i>SizeCond</i>	::=	"size(" <i>Path</i> ")" <i>CompOp</i> <i>Number</i>
<i>TypeCond</i>	::=	"type(" <i>Path</i> ")" <i>EqOp</i> <i>StructType</i>
<i>Path</i>	::=	<i>PName</i> <i>PName</i> [" <i>IdxPos</i> "] <i>Path</i> "/" <i>Path</i>
<i>IdxPos</i>	::=	<i>Number</i> "last()" "first()"
<i>StructType</i>	::=	"KV" "list" "record"
<i>EqOp</i>	::=	"=" "!="
<i>CompOp</i>	::=	"<=" ">=" "<" ">" <i>EqOp</i>
<i>SetOp</i>	::=	"=" "∈" "⊆" "⊇"

FIGURE 6.1: BNF predicate of basic conditions

6.1 Data Acquisition Services

Data Acquisition Services can be devised for processing and combining the streams produced by the sensors that adopt the event stream model. Other services are provided for manipulating the streams (for filtering, transforming, aggregating, composing and for event detection). Stream manipulation services are classified in *non-blocking* and *blocking* services. In the remainder of the section we discuss and detail the developed services. In the presentation we use the following notations. $\mathcal{P}rop(o)$ denotes the set of properties occurring in o , whereas $Gran_T(o)/Gran_S(o)$ represents the temporal/spatial granularity of o and $\mathcal{T}ype(a,s)$ denotes the type domain of property a in the event stream s .

6.1.1 Non-Blocking Services

These services are applied on each single event and thus do not require to maintain caches.

Filter. The `filter` service allows to remove events that do not adhere to a condition *cond* expressed on the values of the event stream s . *cond* is a boolean expression on the properties specified in the schema of the stream s that follows the syntax reported in Figure 6.1. The syntax takes into account the use of records and sets in the structured value of an event.

Definition 6.1 (*filter service*). Let $s = \langle G_T, G_S, th, [t_s, t_e], S, Event_{\langle G_T, G_S \rangle}^{th}(\tau) \rangle$ be an event stream and *cond* a boolean expression that follows the syntax reported in Figure 6.1 such that $\mathcal{P}rop(cond) \subseteq \mathcal{P}rop(\tau)$. The application of the *filter* service, named $\sigma(s, cond)$, on s produces a new stream s' having the same structure of s and containing only the events that meet the constraints imposed by *cond*. \square

Enrich. The `enrich` service allows to include extra information to the stream according to a knowledge base KB or the Domain Ontology or static information sources. The binding between the current event and one of these sources is realized through the join predicate $pred$. This service is particularly important for enriching the events with contextual information that are local to where the event is generated (for example for associating the local correction factor LHD in the computation of HD). The spatio-temporal granularity of s needs to be compliant with the one adopted in KB for a sound enrichment of the event stream.

Definition 6.2 (*enrich service*). Let $s = \langle G_T, G_S, th, [t_s, t_e], S, Event_{\langle G_T, G_S \rangle}^{th}(\tau) \rangle$ be an event stream and KB be a table belonging to a relational database or extracted from a knowledge base presenting the following schema $(b_1, \dots, b_n, b_{n+1}, \dots, b_k)$ whose data are compliant with the spatio-temporal granularities of s . Let $\{b_1, \dots, b_n\}$ be a subset of attributes of KB that are in common with $\mathcal{Prop}(\tau)$ ($\{b_1, \dots, b_n\} = \{b_1, \dots, b_k\} \cap \mathcal{Prop}(\tau)$). The application of the *enrich* service, named $\alpha_{pred}^{KB} s$, on s produces a new stream s' whose structure is $\langle G_T, G_S, th, [t_s, t_e], S, Event_{\langle G_T, G_S \rangle}^{th}(\tau') \rangle$ where τ' is τ enhanced with the attributes (b_{n+1}, \dots, b_k) whose values are taken from KB where the attributes (b_1, \dots, b_n) assume the same values in a event of s . \square

Virtual Property. The `virtual` property service allows to include a new property p to the schema of s according to the specification $spec$. $spec$ is an arithmetic expression allowing to determine the value of p relying on the properties of s .

Definition 6.3 (*virtual property service*). Let $s = \langle G_T, G_S, th, [t_s, t_e], S, Event_{\langle G_T, G_S \rangle}^{th}(\tau) \rangle$ be an event stream, $spec$ be an arithmetic expression such that $\mathcal{Prop}(spec) \subseteq \mathcal{Prop}(\tau)$, and p the name of a properties not used in τ ($p \notin \mathcal{Prop}(\tau)$). The application of the *virtual property service*, named $\uplus_s \langle p, spec \rangle$, on s produces a new stream s' whose structure is $\langle G_T, G_S, th, [t_s, t_e], S, Event_{\langle G_T, G_S \rangle}^{th}(\tau') \rangle$ where τ' is τ enhanced with the attribute p whose value is calculated according to the expression $spec$. \square

Transform. The `transform` service allows to apply the transformation function $trans$ on the properties a_1, \dots, a_n of events in s . At the current stage the following transformation functions have been considered: *i*) for changing the unit of measure (e.g. from yards to meters) or geographical coordinates (e.g. from one standard to another one); *ii*) for checking that data conform to given validation rules (e.g. dates conforming to given patterns); and *iii*) for changing the case of letters. However, further functions can be easily integrated in our framework.

Definition 6.4 (*transform service*). Let $s = \langle G_T, G_S, th, [t_s, t_e], S, Event_{\langle G_T, G_S \rangle}^{th}(\tau) \rangle$ be an event stream, $trans$ a transformation function that takes a single input a generates a new value, and $\{a_1, \dots, a_n\} \subseteq \mathcal{Prop}(\tau)$ a subset of the properties of s . The application of the *transform service*, named $\diamond_{trans}^{\{a_1, \dots, a_n\}} s$, on s produces a new stream s' having the same

structure of s , and the values of attributes $\{a_1, \dots, a_n\} \subseteq \mathcal{P}rop(\tau)$ transformed according to $trans$. \square

6.1.2 Blocking Services

These services are window-based, that is they require to maintain a cache of events for a temporal interval t . At the end of the interval, the events collected in the cache are processed by the operator and the obtained result produced to the upcoming operators.

Aggregation. The aggregation service allows to aggregate the events of s on the properties a_1, \dots, a_n and apply the aggregation function $op \in \{\text{count}, \text{avg}, \text{sum}, \text{min}, \text{max}\}$ on the other properties. The temporal granularity of t needs to be compatible with the one of s . Moreover, the properties a_1, \dots, a_n need to be included in those appearing in s . Note that since this service can be applied to spatio-temporal data, it requires that the aggregation is also applied on the spatio-temporal dimensions in order to produce meaningful data according to our event stream data model.

Definition 6.5 (aggregation service). Let $s = \langle G_T, G_S, th, [t_s, t_e], S, Event_{\langle G_T, G_S \rangle}^{th}(\tau) \rangle$ be an event stream, a_1, \dots, a_n the attributes on which the aggregation should be applied, and $op \in \{\text{count}, \text{avg}, \text{sum}, \text{min}, \text{max}\}$ the aggregation function. Let $a_1, \dots, a_n, a_{n+1}, \dots, a_k$ be the properties in τ . The application of the aggregation service, named $@_{op}^{t, \{a_1, \dots, a_n\}}(s)$, on s in the interval of time t produces a new stream s' with the same structure of s . The events collected in the interval t are grouped according to the attributes a_1, \dots, a_n and the spatio-temporal dimensions. For each group, a representative is generated by applying op on the values associated with each of the properties a_{n+1}, \dots, a_k . \square

Union. The union service allows the union of the events produced by different sensors and produces a new stream of events of thematic th' . Indeed, in this case the user can specify whether the generated sequence of events has a new thematic or maintain one of those of the incoming streams. We remark that this operation is allowed only when common properties of different incoming streams present the same type. However, no constraints is imposed on the other properties.

Definition 6.6 (union service). Let $s_1 = \langle G_{T1}, G_{S1}, th_1, [t_{s1}, t_{e1}], S_1, Event_{\langle G_{T1}, G_{S1} \rangle}^{th_1}(\tau_1) \rangle$ and $s_2 = \langle G_{T2}, G_{S2}, th_2, [t_{s2}, t_{e2}], S_2, Event_{\langle G_{T2}, G_{S2} \rangle}^{th_2}(\tau_2) \rangle$ be two event streams such that:

- $\mathcal{P}rop(\tau_1) = \{a_1, \dots, a_n, a_{n+1}, \dots, a_k\}$ and $\mathcal{P}rop(\tau_2) = \{a_1, \dots, a_n, a'_{n+1}, \dots, a'_k\}$ are the properties of the two streams with the first n properties presenting the same label and type;
- $G_{T1} = G_{T2}$ and $G_{S1} = G_{S2}$;
- th' is the thematic chosen by the user for the result of the application of this service.

The application of the *union* service, named $\cup^{t,th'}(\{s_1, s_2\})$, on s_1 and s_2 in the interval of time t produces a new stream s' with the following structure

$$\langle G_{T1}, G_{S1}, th', [\max(t_{s1}, t_{s2}), \min(t_{e1}, t_{e2})], S_1 \cup S_2, Event_{\langle G_{T1}, G_{S1} \rangle}^{th'}(\tau') \rangle$$

where τ' is a record formed with the properties $\{a_1, \dots, a_n, a_{n+1}, \dots, a_k, a'_{n+1}, \dots, a'_h\}$ whose types are the same of the original basic types of s_1 and s_2 . The events in s' are $\{(a_1 : v_1, \dots, a_n : v_n, a_{n+1} : v_{n+1}, \dots, a_k : v_k, a'_{n+1} : \text{null}, \dots, a'_h : \text{null}) \mid (a_1 : v_1, \dots, a_n : v_n, a_{n+1} : v_{n+1}, \dots, a_k : v_k) \in s_1\} \cup \{(a_1 : v'_1, \dots, a_n : v'_n, a_{n+1} : \text{null}, \dots, a_k : \text{null}, a'_{n+1} : v'_{n+1}, \dots, a'_h : v'_h) \mid (a_1 : v'_1, \dots, a_n : v'_n, a'_{n+1} : v'_{n+1}, \dots, a'_h : v'_h) \in s_2\}$. \square

Join. The join service allows to make in correspondence events of two streams when their temporal and spatial granularities are identical and the join predicate $pred$ is verified. It is executed on the events collected from the two streams in the temporal interval t and produces events of thematic th' .

Definition 6.7 (*join service*). Let $pred \equiv a_1 = b_1 \wedge \dots \wedge a_n = b_n$ be the join predicate, $s_1 = \langle G_{T1}, G_{S1}, th_1, [t_{s1}, t_{e1}], S_1, Event_{\langle G_{T1}, G_{S1} \rangle}^{th_1}(\tau_1) \rangle$ be an event stream and $s_2 = \langle G_{T2}, G_{S2}, th_2, [t_{s2}, t_{e2}], S_2, Event_{\langle G_{T2}, G_{S2} \rangle}^{th_2}(\tau_2) \rangle$ be another event stream such that:

- $G_{T1} = G_{T2}$ and $G_{S1} = G_{S2}$;
- $Prop(\tau_1) = \{a_1, \dots, a_n, a_{n+1}, \dots, a_k\}$ and $Prop(\tau_2) = \{b_1, \dots, b_n, b'_{n+1}, \dots, b'_h\}$ are the properties of the two streams;
- $\forall a \in \{a_1, \dots, a_n\}$, $a \in Prop(\tau_1)$, and $\forall b \in \{b_1, \dots, b_n\}$, $b \in Prop(\tau_2)$, and $Type(a, s_1) = Type(b, s_2)$;
- th' is the thematic chosen by the user for the result of the application of this service.

The application of the *join* service, named $s_1 \bowtie_{\{a_1, \dots, a_n\}}^{t, th'} s_2$, on s_1 and s_2 in the interval of time t produces a new stream s' with the following structure

$$\langle G_{T1}, G_{S1}, th', [\max(t_{s1}, t_{s2}), \min(t_{e1}, t_{e2})], S_1 \cup S_2, Event_{\langle G_{T1}, G_{S1} \rangle}^{th'}(\tau') \rangle$$

where τ' is a record formed with the properties $\{a_1, \dots, a_n, a_{n+1}, \dots, a_k, a'_{n+1}, \dots, a'_h\}$ whose types are the same of the original basic types of s_1 and s_2 . The events in s' are $\{(a_1 : v_1, \dots, a_n : v_n, a_{n+1} : v_{n+1}, \dots, a_k : v_k, b_{n+1} : v'_{n+1}, \dots, b_h : v'_h) \mid (a_1 : v_1, \dots, a_n : v_n, a_{n+1} : v_{n+1}, \dots, a_k : v_k) \in s_1 \text{ and } (b_1 : v_1, \dots, b_n : v_n, b_{n+1} : v'_{n+1}, \dots, b_h : v'_h) \in s_2\}$. \square

Trigger. The trigger on/off service is a kind of semaphore that allows to activate/deactivate the production of events from a streams s_1 when an aggregate condition $cond$ is verified on the events collected from s in the time interval t . The following definition introduces the trigger on service. The other is equivalent (the semaphore is green when the condition is false).

Definition 6.8 (*trigger on service*). Let $s_1 = \langle G_{T1}, G_{S1}, th_1, [t_{s1}, t_{e1}], S_1, Event_{\langle G_{T1}, G_{S1} \rangle}^{th_1}(\tau_1) \rangle$ and $s = \langle G_T, G_S, th, [t_s, t_e], S, Event_{\langle G_T, G_S \rangle}^{th}(\tau) \rangle$, and cond an aggregate condition on the properties of s ($Prop(cond) \subseteq Prop(\tau)$). The application of the trigger on service, named $\Theta_{tg,sg}^t(s, cf)$, on s in the interval of time t produces a new stream s' having the same structure of s_1 . s' corresponds to the events generated by the event stream s_1 in the same interval of time t , if the evaluation of cond on s is true, the empty set, otherwise. \square

Convert. The convert service allows to apply a coercion function cf for changing the spatio-temporal granularity of the stream s . The temporal/spatial granularities tg and sg should be coarser than the one used in s . Coercion functions [26] can be classified into three categories: *selective*, *aggregate*, and *user-defined* coercion functions. Selective coercion functions are `first`, `last`, `proj(index)`, `main`, and `all`. Coercion function `proj(index)`, for each granule in the coarser granularity, returns the value corresponding to the granule of position `index` at the finer granularity. Coercion function `first` and `last` are the obvious specializations of the previous one. Coercion function `main`, for each granule in the coarser granularity, returns the value which appears most frequently in the included granules at the finer granularity. Coercion function `all`, for each granule in the coarser granularity, returns the value which always appears in the included granules at the finer granularity if this value exists, the null value otherwise. Aggregate coercion functions are `min`, `max`, `avg`, and `sum` corresponding to the well-known SQL aggregate functions. User-defined coercion functions (named UDF) correspond to services that can be dynamically loaded in the system and preserve the relationships among granularities.

Definition 6.9 (*convert service*). Let $s = \langle G_T, G_S, th, [t_s, t_e], S, Event_{\langle G_T, G_S \rangle}^{th}(\tau) \rangle$ be an event stream, tg and sg the target temporal and spacial granularities, and $cf \in \{first, last, proj(index), main, all, min, max, avg, sum\} \cup UDF$ a coercion function. When $G_T \preceq tg$ and $G_S \preceq sg$, the application of the convert service, named $\Theta_{tg,sg}^t(s, cf)$, on s produces a new stream s' with the following structure

$$\langle tg, sg, th, [t_{s1}, t_{e1}], S_1, Event_{\langle tg, sg \rangle}^{th}(\tau) \rangle$$

where the events of s are aggregated according to the coercion function cf . \square

6.2 Data Acquisition Plan

A Data Acquisition Plan is a graph that represents the flow of operations on sensor data. This representation helps the user on composing services in order to transform, aggregate, filter and store information. In order to depict the concepts connected with the Data Acquisition Plan, in this chapter we use a formal graphical representation for its composition. The meaning of each Sensor, Service and Destination nodes and of the edges are explained on Table 6.1.

Symbol	Description
	Source node. It can have only 1 outgoing edge and no incoming edge.
	filter, enrich, virtual property, transform, aggregation and convert service. All these services have a single incoming and outgoing edge.
	Destination node. No outgoing edges are allowed and it can receives data from a single node.
	union service. It has two incoming edges and a single outgoing edge.
	join service. It presents the same shape of the union because it allows two incoming edges and a single outgoing edge.
	trigger service. It represents both the trigger event and action. It has 2 incoming edges and 1 outgoing edge.
	Edge that connects each node and service and expresses the flow direction.

TABLE 6.1: Formal Sensors, Services, Destination nodes and edges representation

The following formal definition presents the concept of Data Acquisition Plan as a directed acyclic graph in which the previously presented services can be composed.

Definition 6.10 (A Data Acquisition Plan). A Data Acquisition Plan is a direct acyclic graph $DAP = (V, E)$, where V is a set of vertices representing sensors (V_s), data acquisition services (V_{op}), and destination (V_d), and E represents the flow of events that adheres to our model. \square

Example 6.1 We are now ready for the construction of the Data Acquisition Plan for computing the human discomfort as described in our motivating example. We remark that standard wrappers are employed for transforming the format used for the representation of the events (see Figure 2) in a JSON format.

First, we have to take all the temperatures measured by the sensors of type T_1 and T_2 . Before computing the union of the observed temperatures, we need to convert them to the same spatio-temporal granularity, the same unit of measure, and include the accuracy. Therefore, on the streams produced by sensors of type T_1 the following services are invoked:

$$\begin{aligned} (\text{enrich}) \quad & t_1^1 = \alpha_{\text{accuracy}}^O t_1 \\ (\text{convert}) \quad & t_1^2 \ominus_{\text{hour,zone}}^{1\text{hour}} (t_1^1, \text{avg}) \end{aligned}$$

By exploiting the Domain Ontology the accuracy and zone of the sensors are included in the stream. Then, by exploiting the relationship between the point and zone spatial granularities and the relationship between the 10 minute and 1 hour temporal granularities, the

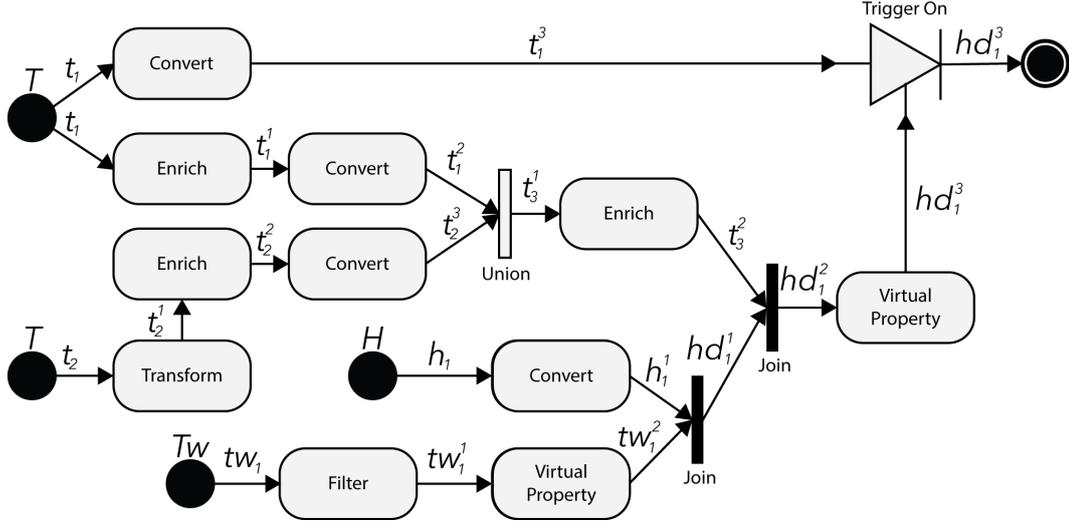


FIGURE 6.2: Graph representation of the Data Acquisition Plan of our running example

average temperature is computed for each hour and zone. Therefore, the event type stream of t_1^2 is (hour, zone, temperature, {temperatureVal:real, accuracy:real}).

Similar behavior is followed for the streams produced by sensors of type T_2 with the exceptions that: i) the temperature values are expressed in Fahrenheit and need to be transformed; ii) the spatial dimension is missing and can be included by means of the information contained in the Domain Ontology; and iii) temperatures are collected every 20 minutes.

$$\begin{aligned}
 (\text{transform}) \quad & t_2^1 \diamond_{\text{far2celsius}}^{\text{temperature}} t_2 \\
 (\text{enrich}) \quad & t_2^2 = \alpha_{\text{zone}}^O t_2^1 \\
 (\text{convert}) \quad & t_2^3 \ominus_{\text{hour,zone}}^{1\text{hour}} (t_2^2, \text{avg})
 \end{aligned}$$

(hour, zone, temperature, {temperatureVal:real}) is the event type stream of t_2^3 . Since the STT dimensions of t_2^3 and t_1^2 are the same, the union of the generated events is possible by the invocation $t_3^1 = \cup^{1\text{hour,temperature}} (\{t_1^2, t_2^3\})$. We remark that the types produced by the two kinds of sensors are different (accuracy is missing in t_2^3). Therefore the value 0.0 is associated for this property to the events of t_2^3 because its type is real. At this point we can enrich the stream t_3^1 with the local correction factor LHD (that are specified for each zone of Milan as specified in Figure 1(a)) by means of the service invocation $t_3^2 = \alpha_{\text{zone}}^{\text{table(lhd)}} t_3^1$. The resulting event type is therefore (hour, zone, temperature, {temperatureVal:real, accuracy:real, lhd:real}).

For what concern the humidity, we need to exploit the relationship between the point and zone spatial granularities and the relationship between the 30 minute and 1 hour temporal granularities to compute the average humidity per hour and per zone. By contrast, for the tweets we need to select those containing the terms hot, heat and sweat and add a virtual property containing the number of selected tweets.

(convert) $h_1^1 = \ominus_{hour,zone}^{1hour} (h_1, avg)$
 (filter) $tw_1^1 = \sigma(tw_1, like(tweets, ".*hot*.")) ||$
 $like(tweets, ".*heat*.")) ||$
 $like(tweets, ".*sweat*."))$
 (virtual property) $tw_1^2 = \uplus_{tw_1^1} \langle numPos, COUNT(tweets) \rangle$

(hour, zone, humidity, {humidityVal: real}) is the event type stream of h_1^1 , whereas the event type stream of tw_1^2 is (minute, zone, tweet, {tweets: list(string), numTweets: int, numPos: int}).

At this point the information about temperature, humidity and tweets are organized according to the same spatio-temporal granularities and can be joined to obtain the terms of the formula and a virtual property can be included in the resulting stream for storing the value of the human discomfort for each hour.

(join) $hd_1^1 = h_1^1 \bowtie_{h_1^1,zone=tw_1^2,zone}^{1hour,humanDisc} tw_1^2$
 (join) $hd_1^2 = hd_1^1 \bowtie_{hd_1^1,zone=t_3^2,zone}^{1hour,humanDisc} t_3^2$
 (virtual property) $hd_1^3 = \uplus_{hd_1^2} \langle hd, (numPos/numTweets) * (temperature + accuracy + (0.555 * (humidity - 10))) + lhd \rangle$

(hour, zone, humanDisc, {temperatureVal: real, accuracy: real, lhd: real, humidityVal: real, tweets: list(string), numTweets: int, numPos: int, hd: real}) is the event type stream of hd_1^3 .

The process for the calculation of the human discomfort is triggered only when the maximal temperature in Milan in the last hour is greater than 20° Celsius. Supposing to use only the sensors of type T_1 for the trigger, the events produced by these sensors need to be converted to the hour time granularity and city spatial granularity. The following two services are thus invoked:

(convert) $t_3^1 = \ominus_{hour,city}^{1hour} (t_1, max)$
 (trigger on) $hd = \oplus_{ON,1hour,humanDisc} (t_3^1, \{hd_1^3\}, temperature > 20)$.

Therefore, the process for the calculation of the stream hd_1^3 is activated only when the trigger condition is verified in the last hour. The event type of the final stream is the same of hd_1^3 . A graphical representation of this Data Acquisition Plan is reported in Figure 6.2. The figure points out the flows of events that are generated by the physical and social sensors and those produced by the application of the different services. Different symbols are used for representing physical and social sensors (bold circles), services (rectangles with rounded corners), the triangle for representing the event trigger and the corresponding action, and the thin rectangles for representing the union and join services) and the destination node (bold double line circle). \square

In the graph representing a Data Acquisition Plan we can specify constraints for obtaining sound execution. Given a Data Acquisition Plan $DAP = (V, E)$, the following functions are used: $deg^-(v)$ represents the number of edges incoming in $v \in V$, whereas $deg^+(v)$ is the number of outgoing edges from v .

6.3 Sound/Consistent Specification of Data Acquisition Plan

A Data Acquisition Plan is considered *sound* if: *i*) the number of input streams is equal to inputS ; *ii*) the required parameters are specified; and *iii*) its conditions can be evaluated on the input stream s and are verified.

Definition 6.11 (A Sound Data Acquisition Plan). Let op_1, \dots, op_n be the services used in a Data Acquisition Plan $DAP = (V, E)$.¹ DAP is sound when: $V_s \neq \emptyset$, $|V_d| = 1$; $\forall v \in V_s$, $\text{deg}^-(v) = 0$, $\forall v \in V_d$, $\text{deg}^+(v) = 0$; for each op_i , $v_i \in V_{op}$, $\text{deg}^-(v_i)$ corresponds to the number of input streams for op_i , each parameter required by op_i is correctly specified, the conditions associated with each service are verified on the the events of the incoming edges. \square

According to our definition of sound Data Acquisition Plan, we are able to acquire streams for which someone of the spatio-temporal-thematics dimensions are not specified or are specified but are not consistent according to Definition 5.8. This is particular useful in order to flexibly adapt to different situations and to post-pone the inclusion of the STT dimensions in the cloud. However, we provide here the notion of a consistent Data Acquisition Plan which is a sound plan where the ontology O and the event model associated with the destination node of the graph are consistent according to Definition 5.8.

Definition 6.12 (A Consistent Data Acquisition Pan). Let $G = (V, E)$ be a sound Data Acquisition Plan according to Definition 6.11, \mathcal{M} and O be the event model and ontology generated for the destination node. G is consistent if: $G_T \neq \perp$, $G_S \neq \perp$, and \mathcal{M} is consistent w.r.t. O . \square

Example 6.2 The Data Acquisition Plan described in example represented on Figure 6.2 is sound according to Definition 6.11. Moreover, the event type of the last instruction (hd) is consistent w.r.t. our Domain Ontology. \square

6.4 Verification of Consistency in a Data Acquisition Plans

In our setting sensors can produce streams of events that can be consistent, partial consistent or inconsistent w.r.t. the Domain Ontology. However, the application of the presented services can alter the schema of the events produced by the sensors and thus modify their consistency. There is therefore the need to check the consistency of the schema produced by the application of single services and of the entire Data Acquisition Plan.

We wish to allow internal nodes of a Data Acquisition Plan to be not consistent (or only partially consistent) w.r.t. the Domain Ontology in order to generate a flexible tool that is able to handle heterogeneous streams that do not perfectly match the

¹For the sake of simplicity, we assume that the service op_i corresponds to vertex v_i .

constraints imposed by the Domain Ontology. This is a key characteristic in the context of IoT where sensors can be included in the platforms in different moments and might not adhere to any pre-established constraints. However, when the final stream produced by the destination node of the Data Acquisition Plan is consistent we can guarantee that its semantics is well described at the ontological level.

In the remainder of the section we first introduce the operations required for the description of the event schema produced by each data acquisition service at the ontological level. Then, an algorithm is presented that starting from a sound Data Acquisition Plan, populates the instances of the Domain Ontology for its description, and determines whether its output is consistent.

6.4.1 Auxiliary Sensors in the Domain Ontology

Once a Data Acquisition Plan is considered sound (according to Definition 6.11), we wish to check its consistency w.r.t. the Domain Ontology (Definition 5.8). With this aim, at the ontological level we need to describe each service (that can be applied to the streams produced by the sensors) as an *auxiliary sensor*, instance of the class `ssn:SensingDevice`. An auxiliary sensor receives one or more incoming streams and produces a single stream whose schema, that is compliant with the STT model, can be made in correspondence with the concepts and relationships available in the Domain Ontology. Also for these auxiliary sensors we can check the consistency w.r.t. the Domain Ontology.

Example 6.3 Consider the situation depicted in Figure 4.6 and in Figure 4.7, T_1^1 and TW_1^2 described in Example 6.1 present ontological schemas similar to those depicted in the figures. In this case, the two sensors are examples of auxiliary sensors generated as result of the Data Acquisition Plan, which properties depend on the applied operator and on the streams of the incoming sensors (either physical or social). Instances are included in our Domain Ontology to represent such data acquisition services and then check the consistency on the resulting Ontology. \square

An auxiliary sensor is obtained in two steps. First, a new sensor j is generated (either by cloning the incoming sensor i or by generating a new one). Then, the new instance and its links are modified or other links are added in order to comply to the operator specification. These operations are specified by means of the simple primitives reported in Table 6.2. Starting from an instance i of the class `ssn:SensingDevice`, these operations introduce new instances and links in the Ontology (when this is possible according to the conceptualization).

The only exception to this general rule are the treatment of the `filter` service and of the `trigger` service. In the first case, we simply need to clone the incoming sensor and no further operations are required, while in the other case we need to clone and check if a trigger condition is verified. In both cases the schema of the

Operation	Meaning
$i = \text{clone}(j)$	Create a copy of the instance j with all the links that involve j
$i = \text{new}(\text{"CLASS"})$	Create a new instance of the class CLASS
$i.\text{addLink}(j, \text{"REL"})$	Include a link between i and j instance of the relationship REL
$i.\text{delLink}(j, \text{"REL"})$	Remove the link between i and j instance of the relationship REL

TABLE 6.2: Primitives for the modification of Ontology instances

incoming sensor is the same produced as output. By contrast, for the other services (transform, enrich, virtual property, aggregation, union, join and convert) the operations reported in Table 6.3 need to be applied.

For the services enrich and virtual property, new instances of the class a are added in the ontology \mathcal{O} in order to model the new properties that are inherited by other sensors (by using the enrich service) or created as new virtual properties (by using the virtual property service). The service transform enables to change the units of measurement of the sensor properties a according to the function *trans* or it applies a transformation by executing a specific code. The service aggregation creates a new auxiliary sensor whose values are gathered according to a new time value defined by a new instance of the class `ssn:ResponseTime` with a new unit of measure. The service join creates a new auxiliary sensor whose properties are the union of the properties of the two input sensors $a \in \text{Prop}(s_1) \cup \text{Prop}(s_2)$ whereas the service union allows us to union the events produced by different sensors and generates a new thematic (instance of the class `qu:QuantityKind`) and collect all properties of the input sensors. Finally, the service convert simply changes the temporal or spatial dimension according to the target granularity specified in input.

Example 6.4 Example 6.1 presents two auxiliary sensors T_1^1 and TW_1^2 generated by sensors of type T_1 and TW_1 . T_1^1 is created by applying an enrich service. As result of the application of this service, the instance T_1 is cloned in T_1^1 and a new link `hasMeasurementProperty` is added for connecting the auxiliary sensor to an instance of the class `ssn:Accuracy` for modeling the accuracy property. In the same way, TW_1^2 is cloned by TW_1 as result of the application of a virtualProperty service. Two new links `hasMeasurementProperty` are added for connecting the new auxiliary sensor to new instances of the class `do:NumTweets` and `do:NumPosTweets` used for reporting the number of tweets and positive tweets.

Now, consider the situation described in Example 6.1. The final formula used for calculating the human discomfort is based on the creation of the auxiliary sensor HD_1^3 which thematic `HumanDisc` is an instance of the class `qu:QuantityKind`. The auxiliary sensor is in turn linked to instances of the class `iot:Metadata` for modeling the associated data type `temperatureVal`, `humidityVal`, `TweetsList` and `hd`, and to instances of the class `ssn:MeasurementProperty` for modeling the measurement properties that characterize the thematic (i.e. `accuracy`, `numTweets`, `numPos` and `lhd`). Once the final auxiliary sensor HD_1^3 is generated, its consistency w.r.t Definition 5.8 is checked. As result, let \mathcal{O} be the Domain Ontology that we have described in the previous examples, we can say that the sensor of type HD_1^3 is consistent w.r.t. \mathcal{O} . \square

Service(Symbol)	InsOntology
Enrich – $\circ_{pred}^{KB} s$	<pre> j=clone(i) for a in Prop(KB) \ Prop(s) do j.addLink(new("a"), "t") (where t is the type of a) end for </pre>
Virtual property – $\uplus_s(p, spec)$	<pre> j=clone(i) Let t be the type of p j.addLink(new("p"), "t") </pre>
Transform – $\diamond_{trans}^{\{a_1, \dots, a_n\}} s$	<pre> j=clone(i) for a in {a1, ..., an} do if trans = CHANGE_UNIT then Consider $i_a \in \mathcal{I}(qu : Unit)$ corresponding to a for instance i j.delLink(i_a, "hasUnit") j.addLink(new("a"), "hasUnit") end if if trans = X then exec specific code for the function X end if end for </pre>
Aggregation – $@_{op}^{\{a_1, \dots, a_n\}}(s)$	<pre> j=clone(i) Let $i_t \in \mathcal{I}(ssn : ResponseTime)$ s.t. a link (i, hasMeasurementProperty, i_t) exists j.delLink(i_t, "hasMeasurementProperty") $j_t = new("ssn : ResponseTime")$ j.addLink(j_t, "hasMeasurementProperty") Consider $j_u \in \mathcal{I}(qu : Unit)$ corresponding to the temporal granularity of t $j_t.addLink(j_u, "hasUnit")$ </pre>
Join – $s_1 \bowtie_{pred}^{t, th'} s_2$	<pre> j = new("ssn : SensingDevice") $j_{th'} = new("qu : QuantityKind")$ $j_{th'}.namedIndividual = th'$ j.addLink($j_{th'}$, "hasQuantityKind") for a in Prop(s_1) do Let (i_1, "t", i_a) be a link to the instance of class a j.addLink(j_a, "t") end for for a in Prop(s_2) \ {b_1, \dots, b_n} do Let (i_2, "t", i_a) be a link to the instance of class a j.addLink(j_a, "t") end for </pre>
Union – $\cup^{t, th'}(\{s_1, s_2\})$	<pre> j = new("ssn : SensingDevice") $j_{th'} = new("qu : QuantityKind")$ $j_{th'}.namedIndividual = th'$ j.addLink($j_{th'}$, "hasQuantityKind") for k in {1, ..., n} do for a in Prop(s_k) do Let (i_k, "t", i_a) be a link to the instance of class a j.addLink(j_a, "t") end for end for </pre>
Convert – $\ominus_{tg, sg}^t(s, cf)$	<pre> j=clone(i) if $tg \neq Gran_T(s)$ then Let (i, "hasMeasurementProperty", i_t) be a link s.t. $i_t \in \mathcal{I}(ssn : ResponseTime)$ j.delLink(i_t, "hasMeasurementProperty") $j_t = new("ssn : ResponseTime")$ j.addLink(j_t, "hasMeasurementProperty") Consider $j_u \in \mathcal{I}(qu : Unit)$ corresponding to the temporal granularity of tg $j_t.addLink(j_u, "hasUnit")$ end if if $sg \neq Gran_S(s)$ then Let (i, "hasLocation", i_s) be a link s.t. $i_s \in \mathcal{I}(iot-lite : Entity)$ or its subclasses j.delLink(i_s, "hasLocation") $j_s = new("ssn : hasLocation")$ j.addLink(j_s, "hasLocation") Consider $j_u \in \mathcal{I}(qu : Unit)$ corresponding to the spatial granularity of sg $j_t.addLink(j_u, "hasUnit")$ end if </pre>

TABLE 6.3: Instructions for modifying the Ontology Instances according to the service

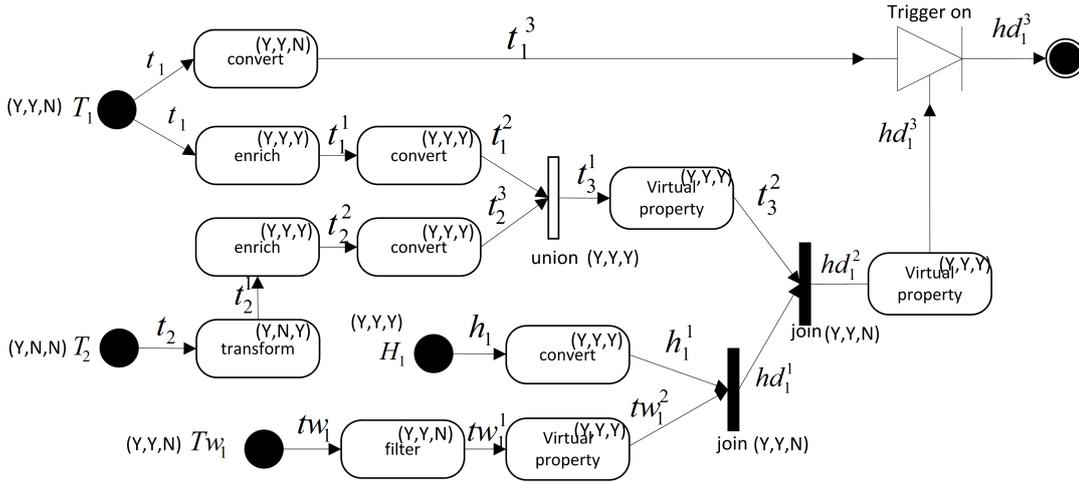


FIGURE 6.3: Evaluation of consistency of the running example's Data Acquisition Plan

6.4.2 A Consistent Data Acquisition Plan

Starting from the operations reported in Table 6.2 for the representation of each single service at the ontological level, we here discuss the algorithm for populating the instances of the Domain Ontology with the information about a Data Acquisition Plan and for verifying whether it is also consistent.

Given a sound Data Acquisition Plan $DAP = (V, E)$ and a Domain Ontology O , we need to create a binding between the schema of the sensors occurring in DAP and the concepts of O . The user carries out this activity by means of a Web tool for specifying a mapping between the properties of the sensor schema and the concepts of the Domain Ontology (details of the GUI are in Section 7.2).

Once the sensors have been mapped to the Domain Ontology, the Data Acquisition Plan is visited in post-order starting from the destination node, and for each service the operations reported in Table 6.3 are applied to provide their representation at the ontological level. These activities are automatically executed and do not require any human interaction and lead to the introduction of the ontology instances for representing the Data Acquisition Plan. The user can check the consistency of each service by means of the graphical interface. At this point it is possible to check the consistency of the entire Data Acquisition Plan DAP according to the following definition.

Definition 6.13 (A Consistent Data Acquisition Plan). *Let $DAP = (V, E)$ be a sound Data Acquisition Plan according to Definition 6.11, \mathcal{M} and \mathcal{O} be the event model and Ontology generated for the destination node. DAP is consistent if: $G_T \neq \perp$, $G_S \neq \perp$, and \mathcal{M} is consistent w.r.t. \mathcal{O} . \square*

Example 6.5 Figure 6.3 reports the graph representation of the Data Acquisition Plan discussed in Example 6.1 annotated with the consistency of the initial flows produced by the

sensors and of the streams produced by the application of the internal services. The triple $(Y/N, Y/N, Y/N)$ is used to denote whether the stream is spatially, temporally, and thematically consistent with respect to the adopted Domain Ontology. The triple is reported within the rounded rectangles denoting services for representing that the consistency of the stream produced by the service. Note that, in this case, even if some services are not consistent, the entire Data Acquisition Plan is consistent because the last stream (hd) is consistent w.r.t. our Domain Ontology. \square

Starting from the consistency of the sensors adopted in a Data Acquisition Plan, it is possible to determine the consistency of its services and of the produced final stream without the need to verify the consistency conditions as specified by the following lemma.

Lemma 6.1 *Let $\mathcal{M}_{b_1}, \dots, \mathcal{M}_{b_n}$ be the event data models of the incoming streams to a service op and \mathcal{O}_b be the ontology before the application of op . Let \mathcal{M}_a be an event data model and \mathcal{O}_a be the ontology after the application of op . The following statements hold:*

- *if $op = \text{filter}$, then $n = 1$ and $\mathcal{O}_a = \mathcal{O}_b$, and the consistency w.r.t \mathcal{O}_a is the same of \mathcal{O}_b .*
- *if $op \in \{\text{enrich}, \text{virtualproperty}\}$ and \mathcal{M}_{b_1} is consistent w.r.t \mathcal{O}_b , then \mathcal{M}_a loses the thematic consistency w.r.t. \mathcal{O}_a .*
- *if $op = \text{convert}$ and the thematic of s is t , the consistency w.r.t \mathcal{O}_a is the same of \mathcal{O}_b .*
- *if $op = \text{union}$ and the thematics of s_1 and s_2 are th' , the consistency w.r.t \mathcal{O}_a is the same of \mathcal{O}_b .* \square

This lemma can be easily demonstrated for induction by taking into account the specification of each single data acquisition service.

Chapter 7

The StreamLoader Prototype

The facilities so far described have been implemented in the context of the StreamLoader system. This system is specifically tailored for domain experts that need to develop different kinds of analysis on streams of events produced by sensors belonging to cross-domain platforms. Since domain experts are usually not computer experts, specific attentions have been devoted to create easy to use interfaces that support them in the Semantic Virtualization of heterogeneous sensors according to the Domain Ontology and in manipulating the flow of events that they produce by means of Data Acquisition Plans that easily adapt to their mental model. The key concept is to involve domain experts in activities for labeling and translating their professional knowledge into proper vocabularies, notations, and suitable visual structures of navigation among interactive systems interface elements. In our context of use, the purpose is to obtain a system that allows the domain experts to focus on the analysis of data-flows rather than on technical details related to the configuration of software and hardware components and on the development of code.

The obtained system thus leverages the complexity of generating code in sophisticated processing systems, configure cluster of machines and set up communication protocols in a easy to use Web application in which the user can focus on the kind of Data Acquisition Plan that he/she wishes to develop, specify the basic service parameters and configuration options, and graphically draw the Data Acquisition Plan. The system takes care to check the soundness and consistency of the developed plan and provide feedbacks for correcting mistakes. Once the DAP is ready, all the required code is automatically generated and deployed in a cluster of machines for its execution.

The chapter is organized as follows. Section 7.1 provides an overview of the graphical environment along with details on the technologies used for its development. Section 7.2 discusses the graphical facilities for the Semantic Virtualization process of sensors made available from the cross-domain IoT platform, while Section 7.3 presents the facilities for the specification of a DAP and for checking its soundness and consistency. Finally, Section 7.4 reports the details for translating a DAP into a Spark Streaming script.

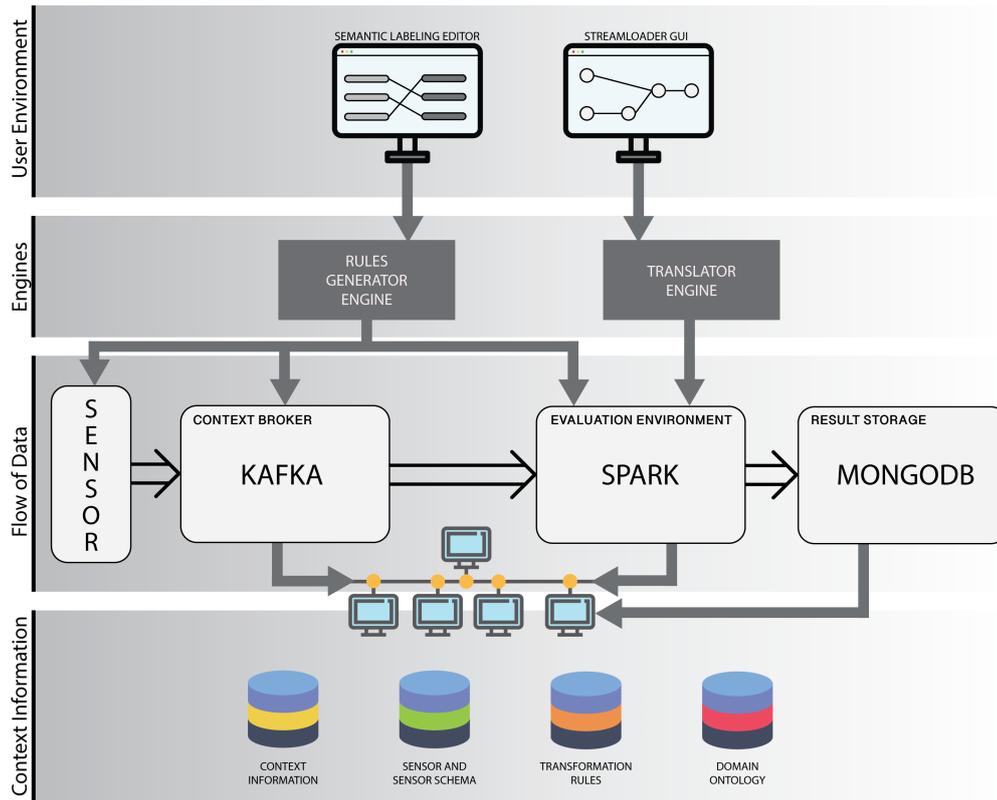


FIGURE 7.1: The overall architecture

7.1 The Overall StreamLoader Environment

The architecture at the base of our system is depicted in Figure 7.1, which extends with further details the one presented in Figure 3. By taking into account the functions exploited by the architecture components, we can identify four main layers:

- *User Environment*. It contains the graphical interfaces for the Semantic Virtualization of the attributes of the sensors' schema and for the composition of sound and consistent Data Acquisition Plans.
- *Engines*. They are responsible for the generation of the transformation rules and the translation of the graphical specification of DAPs into executable Scala scripts containing the Apache Spark Streaming primitives.
- *Flow of Data*. Once the spark process corresponding to the script is executed, the events generated from the sensors are collected and transmitted to the spark process by means of the Kafka context broker. The obtained results are finally stored in the MongoDB storage for further analysis.
- *Context Information*. Static data, context information, transformation rules, registered sensors and the corresponding sensor schemas, semantic labeling are maintained in a MongoDB storage and exploited in different parts of the architecture by means of its query language. Moreover, the Domain Ontology

and the instances created for representing sensors and services exploited in the DAP are maintained in Apache Jena server¹ and queried/updated by means of SparkQL.

As shown in the Flow of Data layer in Figure 7.1, the *context broker*, the *evaluation environment* and the *result storage* are executed in a cluster of machines. Even if the cluster should be set up in advance with respect to the execution of the Spark script, it is quite easy to change its configuration without having to modify the Stream-Loader structure.

In the development of the graphical interfaces we use HTML 5, CSS 3 and JavaScript (JS) technologies. Specifically, Cytoscape.js [50], an open source graph library, has been used for the specification of a DAP. Cytoscape.js is implemented as a standalone JS library that does not have dependencies and does not require browser plugins or the installation of other libraries. However, it includes hooks to several useful libraries and environments, including CommonJS/Node.js, AMD/Require.js, jQuery, Bower, npm, spm and Meteor. This allows Cytoscape.js to integrate into a wide variety of JS-based software systems and to be used heedlessly (i.e. without a graphical user interface) or as a visualization component of a HTML 5 canvas. Cytoscape.js can be used in several domains (e.g. biological networks, social graphs).

For what concern the Semantic Virtualization of sensors, the graphical interface has been realized through JSP pages that interact from one side with Apache Kafka for discovering the presence of new sensors and with our MongoDB database for the management of registered sensors and, from the other side, with our Domain Ontology that is represented through an RDF file [115]. We use SPARQL to query the Domain Ontology using Apache Jena. Apache Jena is an open source framework that provides utilities to build semantic web and linked data applications. This framework offers several APIs that are used to query our Domain Ontology and to introduce instances. It can be seen as a SPARQL [125] server that provides facilities for querying and updating an Ontology by means of SPARQL query language over HTTP.

7.2 Semantic Virtualization Graphical Specification

In order to semantically label the sensor schema, a graphical interface has been created that supports domain experts on this activity.

Our interface shows the list of sensors that are exposed by the context broker and stored in our MongoDB database according to the following criteria: *i) Not mapped*, sensors that have been just discovered and none of their attributes have been yet semantically annotated with a concept of the Domain Ontology; *ii) Partially mapped*,

¹<http://jena.apache.org/>

The screenshot shows a web interface titled "Semantic Labeling". On the left is a "Menu" sidebar with categories: "Newly Arrived" (red), "Partially Mapped" (yellow), and "Fully Mapped" (green). Under "Newly Arrived", several sensors are listed: Sensor78945, Sensor1245, Sensor12 (highlighted in blue), Sensor5938, and Sensor2587. The main area is divided into two panels. The left panel, "Sensor Information", contains fields for "Sensor Name" (with "Sensor12" entered), "Sensor Reading Interval", "Start Date" (format mm/dd/yyyy), and "End Date" (format mm/dd/yyyy). The right panel, "STT Granularities", has three tabs: "Spatial Granularity", "Temporal Granularity", and "Thematic Granularity". Under "Spatial Granularity", there is a "Geographic Point" section with a radio button for "Geo-Point". Under "Thematic Granularity", there is a "Geographic Entity" section with radio buttons for "Zone", "Region", and "City". A "Proceed to Labeling" button is at the bottom right.

FIGURE 7.2: Specification of the STT granularities

sensors that are partially consistent with the Domain Ontology according to Definition 5.8 (e.g. only a part of their attributes have been semantically labeled); and, *iii*) *Fully mapped*, sensors in which their schema is consistent with the Domain Ontology according to Definition 5.8. Different colors are used to provide a visual distinction among the different categories of sensors: *red*, for newly arrived and unmapped sensors; *yellow* for partially mapped and *green* for fully mapped. This graphical distinction can be seen in Figure 7.2.

The interaction with the context broker is realized by means of the *SensorDiscovery Algorithm 1* discussed in Chapter 5. When the *SensorDiscovery Algorithm* starts, it creates a consumer object that discovers the list of available sensors (*channels*) and then list all registered sensors if there are already in the pool. Then, periodically (based on a configured interval of time), it checks the available list of registered sensors against the current list. When a new sensor shows up in the context broker, it is detected and the format of the observed data. In the data format identification phase, the algorithm checks the first few characters of the data to determine its format (JSON, XML or CSV). According to the detected format, the algorithm determines the name and type (for example string, number, date) of each attribute of the sensor schema by inspecting a sample of the observations produced by the sensor through the context broker. Then, the new sensor is labeled *Not mapped* and added to the unmapped sensor collection of our MongoDB storage.

Whenever a new unmapped sensor is stored in the database, it becomes accessible by our Semantic Labeling Editor for semantic labeling. The new unmapped sensor is appended at the list of unmapped sensors on the left panel of the interface in Figure 7.2. At this point, the domain expert can select it (or any other sensors that he wishes to inspect) and specify the semantic labeling. The Semantic Labeling Editor serves two purposes: firstly, to support domain experts in specifying, for each attribute of the sensor schema which of the ontology concepts is related to; secondly,

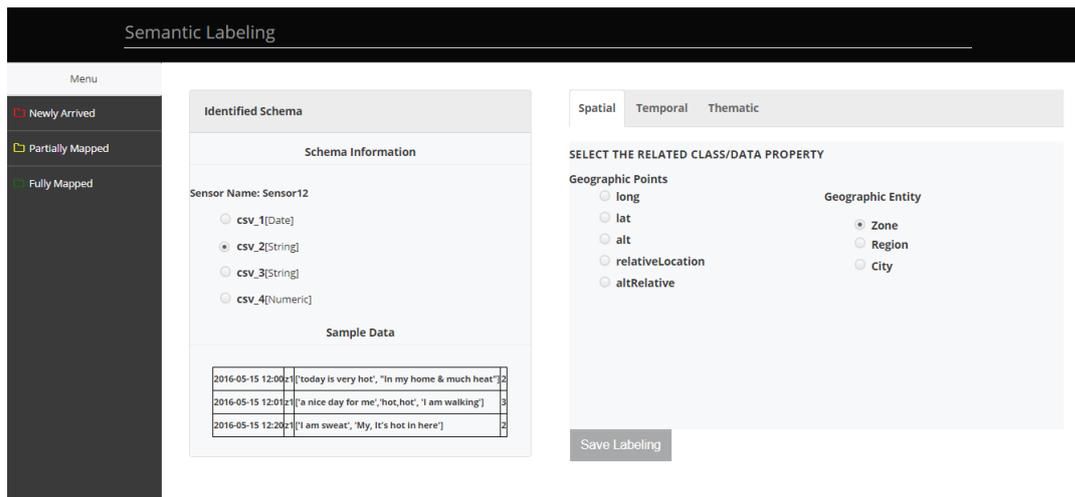


FIGURE 7.3: Spatial Labeling

to allow domain experts to describe some information related to the granularities of the retrieved data.

Once a sensor from the *Not mapped* list is selected, an interface is shown to the domain expert on the right panel of the interface for the specification of its thematic and the spatial and temporal granularities (see Figure 7.2).

This information is not indicated in the sensor schema but the expert has to specify it according to his/her knowledge of the domain context. For example, in Figure 7.2 the domain expert specifies the *Sensor Reading Interval* that indicates the sensor starts to gather data from to a *Start Date* and finishes to an *End Date*. After that, the domain expert can specify the granularities related to the sensor. For the specification of the spatial granularity, a list of concepts related to location (geographic entities – `do:zone`, `do:city`, `do:region` and the geographic point – `geo:Point` class) are extracted from the Domain Ontology with the support of Apache Jena and presented to the user.

The same approach is used for the temporal granularity by showing all the concepts related to the `ssn:ResponseTime` class that have `unit` object property and `value` data property. Specifically, the system retrieves from the Domain Ontology concepts related to temporal duration class (i.e. `time:temporalUnit`, `time:DurationDescription`) to represent the unit (such as `second`, `minute`, `hour`, `week`) of the response time while its value is a data property manually inserted by the domain expert. Finally, for the thematic, the system presents all concepts related to the `iot-lite:QuantityKind` class.

Once the thematic and spatio-temporal granularities have been specified, the domain expert can continue with the semantic labeling procedure in order to describe how each attribute of the sensor is linked to a proper concept of the Domain Ontology. A new interface is shown through which it is possible to specify and/or modify

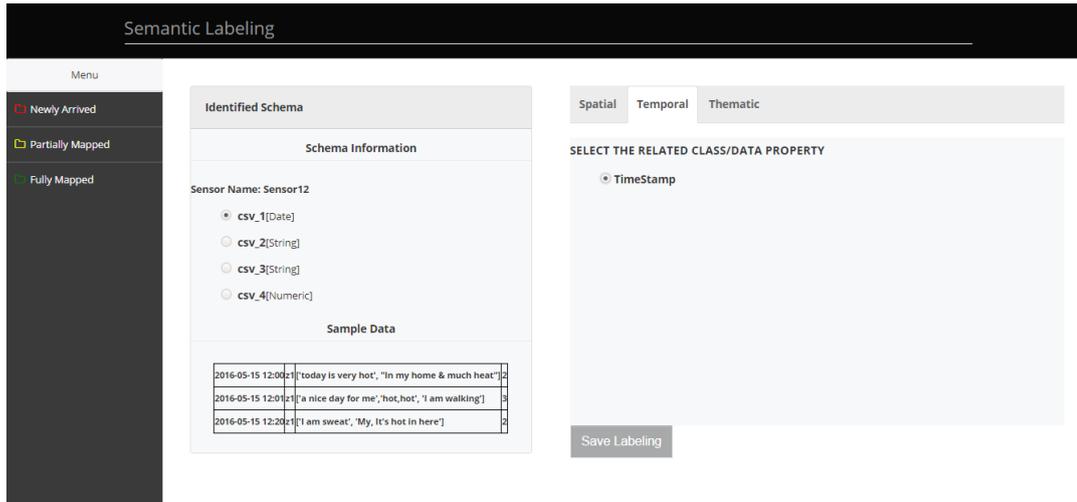


FIGURE 7.4: Temporal Labeling

the semantic labeling w.r.t. the Domain Ontology according to the three STT dimensions (see Figure 7.3). For each attribute of the schema showed in the central wall, the right side of the interface presents the ontology concepts that can be used for the semantic labeling classified according to the STT dimensions in three tabs. Moreover, in the bottom part of the interface a sample of the data produced by the sensor is shown to the user. The sample is useful to the domain expert for better identifying the concepts to be used for annotating the attributes of the sensor schema.

The concepts that can be selected for the three tabs are retrieved from the Domain Ontology through SPARQL queries posed to Apache Jena. The queries also exploit the data-modeling vocabulary available in the RDF Schema to retrieve, for example, all sub-classes of geographic entity. In this case the SPARQL query is `?geoEntity rdfs:subClassOf iot-lite:Entity`. For the spatial dimension, concepts related to geographical point are presented (such as latitude and longitude by using the SPARQL query: `?allGeoPoints rdfs:domain wgspos:Point`) along with the concepts related to the geographic entity `?allGeoEntity rdfs:subClassOf iot-lite:Entity`. In the same way, for the temporal dimension, the interface presents concepts and classes that are related to the timestamp. For thematic dimension, the query `?allMeasProp rdfs:subClassOf ssn:MeasurementProperty` is used to retrieve and present all concepts related to the class `ssn:MeasurementProperty` (such as `ssn:Frequency`, `ssn:Accuracy`, `ssn:Latency`, `do:NumTweets`, `do:NumPosTweets`, and so on). We remark that since the Domain Ontology is represented through RDF, the SPARQL queries return the URIs that identify the corresponding concepts such as `http://pur1.oclc.org/NET/ssnx/ssn#Accuracy` that is used for pointing the class `ssn:Accuracy`, or `http://www.w3.org/2006/time#Instant` that is used for the class `tl:Instant`. For the sake of readability, a dictionary is maintained for the automatic translation of the URIs in more meaningful terms (in the example the URIs are substituted with the terms `Accuracy` and `Timestamp`).

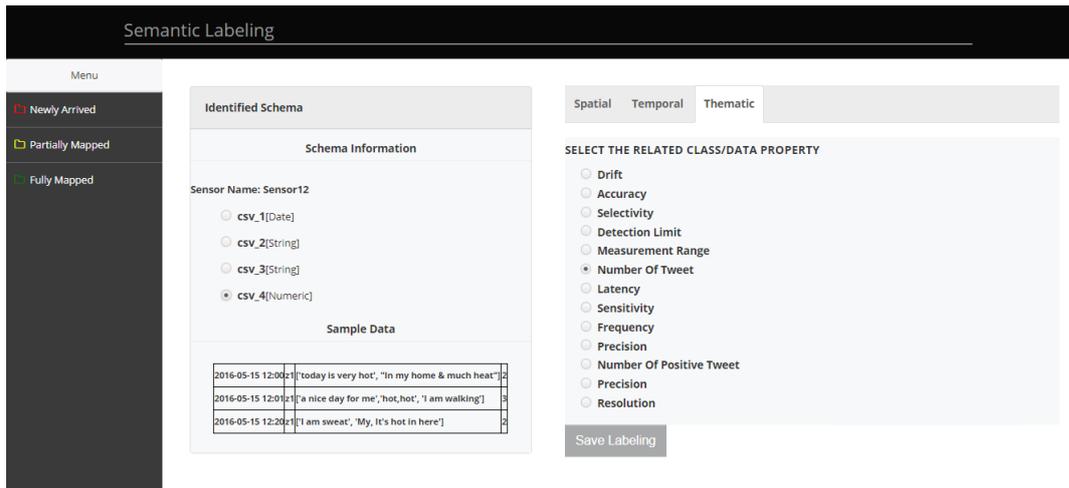


FIGURE 7.5: Thematic Labeling

By exploiting the SPARQL queries so far proposed, the content of our interface can be dynamically populated. Therefore, in case of modifications to the Domain Ontology, the new introduced concepts can automatically be shown in the interface without having to modify its structure. This is an important feature to exploit the developed interface in different context of use.

Figure 7.3 provides an example of labeling of the sensor attribute `csv_2` with a concept belonging to the spatial tab. By looking at the sample, the domain expert sees that the possible value is `z1` that are used for representing city zones and can identify in the `do:zone` class a right concept with which the attribute can be annotated. In the same way, Figure 7.4 shows an example of labeling of the sensor attribute `csv_1` with a concept belonging to the temporal tab. Since, in the current version of the Domain Ontology only the `timestamp` concept is present, this annotation can be expressed by the domain expert. `timestamp` is a more readable and meaningful term that can be used in place of the URI `http://www.w3.org/2006/time#Instant` that identifies the class `t1:Instant`.

An example of thematic labeling is shown in Figure 7.5 where the sensor attribute `csv_4` can be annotated with one of the concepts of the provided list. Since, its type is numeric and sample points out that it is used to report the number of tweets, the domain expert can associate it with the `Number of tweets` that is linked to the class: `do:NumTweets` of the Domain Ontology.

If each attribute of the sensor schema is properly annotated with a concept of the Domain Ontology according to one of the three STT dimensions, and the conditions specified in Definition 5.8, the sensor can be moved to the list of mapped or partially mapped sensors. The specified annotations are then stored in the MongoDB along with the sensor and its schema. Moreover, the `SensorSemanticCharacterization Algorithm 2` presented in Chapter 5 is applied for the semantic characterization of the sensor in the Domain Ontology. The algorithm introduces an instance of the

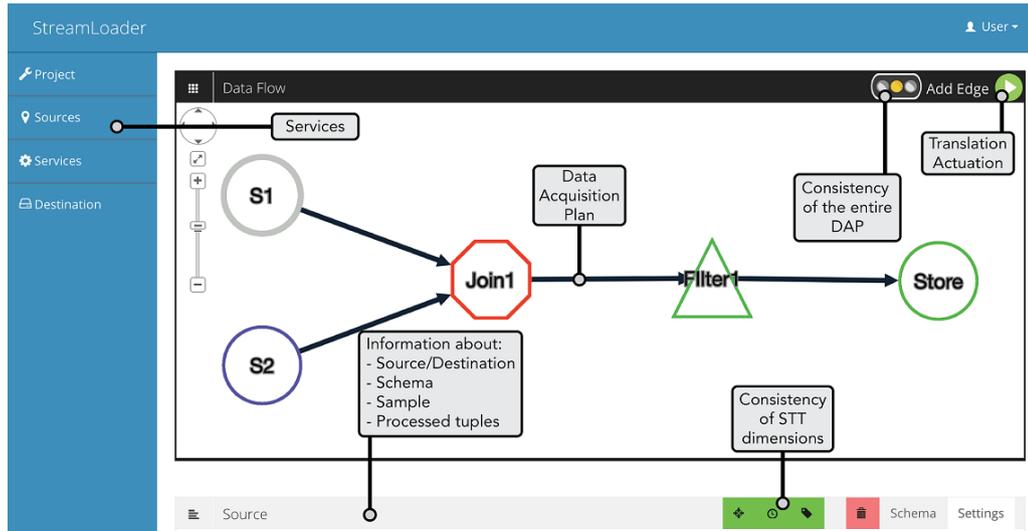


FIGURE 7.6: Main screen of the Web Application

class `ssn:SensingDevice` for the representation of the sensor. In turn, proper instances and properties are generated in the Domain Ontology for representing all three dimensions. For instance, for the sensor discussed so far, since the sensor produce events related to a city zone, the instance representing the sensor is linked to an instance of the class `do:zone` through the property `hasLocation`.

7.3 Data Acquisition Plan Graphical Specification

This interface supports domain expert on the composition of the services that constitute a Data Acquisition Plan. It is an HTML web page developed with the Bootstrap framework.

As shown in Figure 7.6 the interface is composed of a left sidebar menu. The menu offers the following functionalities: *Projects*, for the management of the data acquisition projects (create, delete, save, and open projects); *Sources*, for selecting and discovering sensors from which event streams can be acquired; *Services*, for selecting one of the services described in Section 6.1 of Chapter 6; *Destination* for specifying the repository where the resulting stream should be stored. If a sensor, which is shown on the *Sources* menu, is not mapped or it is partially mapped is it possible to directly move to the Semantic Virtualization Graphical Specification application (presented on Section 7.2), in order to semantically label its attributes to the concepts of the Domain Ontology. By clicking on each of them, a modal menu appears through which it is possible to select icons to be included in the main canvas of the web interface. Moreover, connections can be drawn in the canvas among the services, the sensors and the destination nodes.

Service	Symbol	Service	Symbol
Source		Filter	
Enrich		Virtual Property	
Transform		Aggregation	
Join		Union	
Convert		Trigger Event	
Trigger ON-OFF		Destination	

TABLE 7.1: Graphical representation of source, services and destination

When the icon is placed in the canvas its border is colored in red to represent the fact that the domain expert needs to specify parameters to obtain a sound specification. Once the parameters are specified, the associated conditions are evaluated and, when it is sound, the border of the icon is colored in blue. The border color becomes green when also the consistency constraints are met.

At the bottom of the canvas, a horizontal tab menu area appears when icons (e.g. services or connections) are double-clicked. Relying on the type of element, it shows different buttons and information and is used to define the name of the services, the type of sensors, the filtering conditions, and the trigger condition. It also provides the event types that are processed and the conditions for being sound and consistent. The consistency of the STT dimensions of each node is provided by the 3 icons (a point for the spatial dimension, a clock for the temporal dimension, a tag for the thematic dimension) on the right hand side of the horizontal tab menu header. When the consistency is met, the icon is coloured green. The consistency of the entire Data Acquisition Plan is shown in the top right border of Figure 7.6 by means of a traffic light (green is consistent, yellow is partially consistent, red is not consistent).

As specified in Section 6.1 several services are provided by the interface for processing and combining the streams produced by the sensors. Table 7.1 reports the graphical representation of the services.

During the creation of the Data Acquisition Plan, the interface implements the control of soundness and shows the identified errors. These messages support the domain expert in the development of sound and consistent Data Acquisition Plans. Figure 7.7 provides an example of an error that is shown to the user during the creation of a DAP with two sensors and an union service. Sensor s1 has been successfully connected to union service, while, if the domain expert tries to connect sensor s2, that have a different temporal and/or spatial granularity, the system does not allow this operation and raises an error with a suggestion of the type of service that should be added on the Data Acquisition Plan.

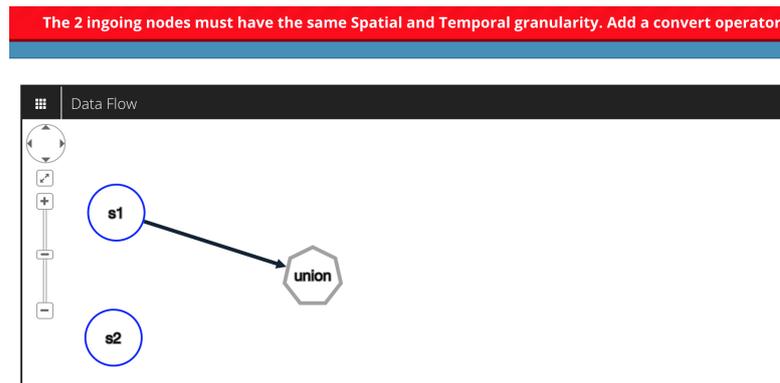


FIGURE 7.7: Identification of an error during the DAP creation

Horizontal tab menus consist of sections that provide different information based on the type service and there are some difference between every operator. In the remainder we describe how each service has been developed on the interface and what kind of information can be added to each of them.

Source. It represents the sensor. The horizontal tab provides information about STT dimensions and granularity and allows the domain expert to modify the name of the sensor (to be shown in the interface). Figure 7.8 shows the source horizontal tab. The second tab (*Schema tab*) provides the stream event type that is produced by the sensor.

The screenshot shows a 'Source' tab with a toolbar containing icons for expand, stop, and refresh. Below the toolbar is a 'New Property Name' input field containing 'aaa'. Underneath is the 'STT Model' section, which is a table with three columns: 'Spatial Dimension', 'Temporal Dimension', and 'Thematic Dimension'. The rows contain the values 'zone z1', 'seconds 30', and 'humidity'.

Spatial Dimension	Temporal Dimension	Thematic Dimension
zone z1	seconds 30	humidity

FIGURE 7.8: Source tab menu

Filter. This service allows domain experts to remove events that do not adhere to specific conditions. This tab allows domain expert to define if the filtering conditions

must be combined with AND or OR clause, by clicking on the two blue buttons, as shown on Figure 7.9. The add and remove buttons allow domain expert to add/re-

FIGURE 7.9: Filter tab menu

move a three fields conditions to the filter service. The first field is the attribute that the domain expert needs to filter and it is the list of the attribute inherited from the incoming node. The second field is the comparison operator (<, >, <=, >=, ==, !=) while the last field is the value.

Enrich. This operator has been developed with the aim of adding stored information to the sensor. It gives the possibility to add information taken from tables stored on a database or it gives the possibility to enrich the sensor with information provided from the Domain Ontology. For this reason the horizontal tab menu is composed of three elements: *Table*, *Domain Ontology* and *Schema*. Figure 7.10 shows the element of the Table tab. This tab is similar to the Join *Setting* tab (described below) because the enrich can be seen as a join of a stream of data with some stored information. The tab is composed of the service name, a new thematic name, the enrich window size, the selection of the stored table and the predicate. With add and remove buttons domain expert can add or remove conditions to the enrich predicate.

Virtual property. This service allows the domain expert to create a new property based on arithmetic expression. As shown in Figure 7.11 at the bottom of horizontal tab a list of all the properties (from the incoming sensor) is provided. In a text area the domain expert can create a function using the mathematical operators (+, -, *, /), using brackets and by adding the properties of the incoming streams by just clicking on their names shown in the list. Before applying the calculation, the domain expert is asked to give a name to the new (virtual) property. The function calculated in the text area is the Human Discomfort formula of our motivating scenario. The second tab (*Schema tab*) is used to show the event type that is generated by the service. In this tab the domain expert can check whether the included virtual property is also described at the ontological level.

FIGURE 7.10: Enrich tab menu

FIGURE 7.11: Virtual Property tab menu

Transform. This service is used to transform or modify the value of some attributes applying some transformation functions. As shown on Figure 7.12, the tab menu allows the domain expert to select either the attribute to transform and the transformation function. By applying the transform function the thematic associated with the node may no longer adhere and for this reason the operator allows the domain expert to specify a new thematic dimension.

Aggregation. It allows to perform an aggregation function to a selected attribute, taken in a specific time interval. The three parameters (aggregation function, attribute, aggregation interval) can be selected directly from the horizontal tab menu, as presented on Figure 7.13. Also in this case a new thematic can be provided.

Transform

Transform Name
transform

New Thematic Name
temperature

Select the attribute to Transform
measurement

Select the Transformation Function
Celsius to Fahrenheit

FIGURE 7.12: Transform tab menu

Aggregate

Aggregate name
aggregation

New Thematic Name
temperature

Select the attribute to aggregate
measurement

Select Aggregation Type
MIN

Select the value
20

Select Unit
minutes

FIGURE 7.13: Aggregation tab menu

Union. It is used to merge events produced by different sensors. This operator can be applied only if the two sensors have the same temporal and spatial dimensions, otherwise an error is raised to the domain expert, as shown on Figure 7.7. The horizontal tab is used to specify the union interval and shows the thematic of the two incoming streams. The domain expert can choose one of these two thematics or define a new one for the outgoing stream of events. Figure 7.14 shows the union horizontal tab menu.

Join. This service corresponds to the SQL JOIN and the associated icon has two incoming edges and a single outgoing edge. It easily allows the join between the different attributes of the schema coming from the ingoing streams. Figure 7.15 shows the *Settings* sections that compose a join service. It contains the name associated with the join service, the temporal windows according to which the join is evaluated, the

FIGURE 7.14: Union tab menu

FIGURE 7.15: Join tab menu

generated thematic, and the join predicate. By means of the add and remove buttons a domain expert can add/remove conditions to the join predicate.

We remark that the domain type of the join result is composed by the union of the properties of the incoming streams (even if the domain expert can decide which properties should be made available to the outgoing service). Property names are disambiguated relying on the names of the incoming services.

Trigger. It is composed of two different services: the trigger event and the trigger actions (ON and OFF). This service, differently from the other services, has three different sections. The *Schema tab* is similar to the others with the exception that it introduces the *Settings ON tab* and *Settings OFF tab*. Figure 7.16 shows the Settings ON section (Settings OFF section is exactly the same). First, domain expert must select the trigger event ON or OFF count conditions. In other word the condition that

FIGURE 7.16: Trigger Event tab menu

"activates" a Trigger Action ON or OFF and it can be greater or less than a certain numeric value. In order to activate a Trigger Action the tuples must meet the specified conditions (that are combined with AND or OR clause), similarly to the definition of filtering conditions.

Convert. This service has been defined with the aim of modify temporal and spatial granularities. Similarly to the Trigger Event service it has three different sections. The *Schema tab* is similar as the other but it introduces *Temporal tab* and *Spatial tab* as shown on Figure 7.17. These two sections show to the domain expert the actual temporal and spatial granularities and give the possibility to the domain expert to change them. The new granularities must be a multiple of the actual ones otherwise an error will be raised.

Example 7.1 Figure 7.18 shows the Data Acquisition Plan that is created for our running example and corresponds to the plan of Example 6.1. Note that the shapes of the icons are coloured in blue or green depending on the consistency of the operator w.r.t. the Domain Ontology and that the traffic light in the top right corner is green because the Data Acquisition Plan is consistent w.r.t. Definition 6.13. □

7.4 Data Acquisition Plan Translation

Once the DAP has been graphically drawn by means of the Cytoscape.js library and we have checked that is sound according to Definition 6.11 in Chapter 6, it can be translated in a language that allows the execution of the defined services in a cluster of machines. In Chapter 1 we have described many frameworks that can be exploited for processing big streams of sensor data. Among them we decided to adopt Apache Spark Streaming because it allows to handle both stored and stream data. However, also other frameworks can be adopted (with a slightly modification of the supported services). Among the possible languages that support Apache Spark

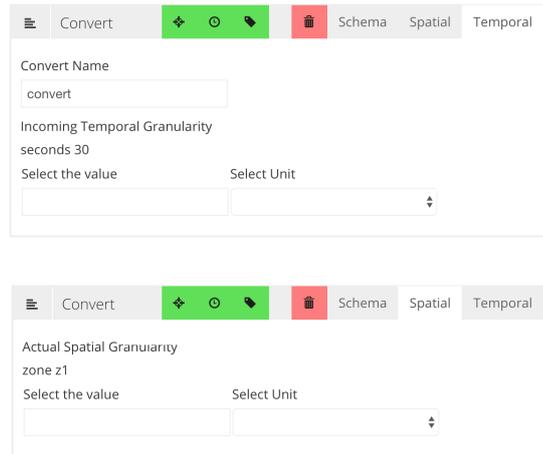


FIGURE 7.17: Convert Temporal and Spatial menus

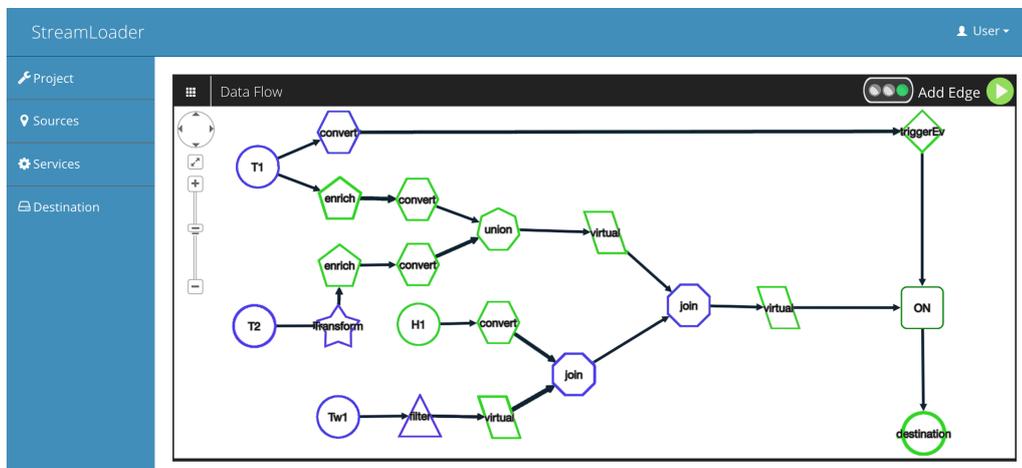


FIGURE 7.18: Data Acquisition Plan for the running example realized with the provided GUI

Streaming we decided to translate the DAP in Scala code because of its functional programming support.

For the presentation of the DAP translation we will refer to an excerpt of our running example reported in Figure 7.19. In the example we use this part of the running example to provide enough details of the translation process with a limited amount of code to be reported in the text.

In the remainder of the section we first discuss the JSON representation of the DAP. Then, we show its translation in a Spark streaming script. Specifically, we describe the general configuration, the representation of the sources and destination, the representation of the single services, and finally we provide an example of translation of an entire DAP.

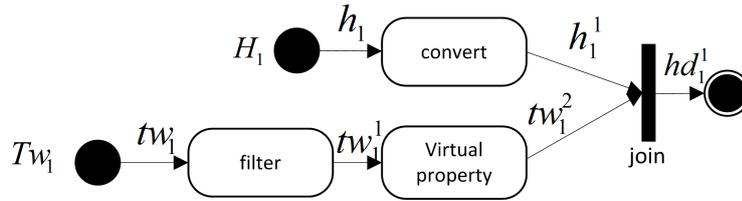


FIGURE 7.19: Excerpt of the DAP to be translated

7.4.1 JSON Representation of a DAP

As discussed in previous chapter, a DAP is a graph and its JSON representation points out this model. Specifically, the JSON representation of a DAP is a list of data elements used for modeling:

- sensors, destination, and, configuration parameters for the deployment of the code in a specific architecture;
- services for the manipulation of sensor data;
- edges for determining the preceding ordering among the services.

In the remainder we present these representations that are then exploited for the translation of the DAP in the Scala code.

Sensors, Destination, and Configuration Sensors are represented by means of a data object presenting a property object whose value is source. Beside a system generated identifier (*id*) and the name specified by the domain expert in the graphical representation, the object contains the information about the STT dimensions through the key *thematic*, *temporal_gran* and *spatial_gran*. Moreover, the attributes belonging to the sensor schema are represented by means of the array named *nodesattributes* array and is composed of arrays of three elements: *i*) the attribute name; *ii*) the attribute type (e.g., string, numeric or datetime); and *iii*) the (optional) unit of measure. The general structure of this object is reported in the following code.

```

1 "data": {
2   "id": "node_id",
3   "name": "node_name",
4   "thematic": "node_thematic",
5   "object": "source",
6   "temporal_gran": "node_temporal_granularity",
7   "spatial_gran": "node_spatial_granularity",
8   "nodesattributes": [
9     ["attribute_name1", "attribute_type1", "attribute_unit1"], ...
10    ["attribute_name_n", "attribute_type_n", "attribute_unit_n"]]

```

CODE 7.1: Node representation

The representation of a destination is analogous with the exception of the value that assume the key object (which is *destination*) and the presence of three extra keys. The first one, *source*, contains the identifier of the service/source from which the events arrive and needs to be stored in the destination node. The second one, *configuration*, contains information about the Spark cluster master node host, batch dimension and the url of the Apache Kafka server, whereas the last one, *storage*, reports the type of storage, the location of the database, the name of the database and of the collection/table where data must be saved.

Consider the sensors H_1 and Tw_1 , and the destination on the DAP in Figure 7.19. The following JSON representation is generated for them.

```

1 "data": {
2   "id": "s1",
3   "name": "h1",
4   "thematic": "humidity",
5   "object": "source",
6   "temporal_gran": "minutes 30",
7   "spatial_gran": "point",
8   "nodesattributes": [
9     ["timestamp", "datetime", ""], ["geo:lat", "numeric", "degree"],
10    ["geo:long", "numeric", "degree"], ["humidityVal", "numeric", "percentage" ]]],
11 "data": {
12   "id": "s2",
13   "name": "tw1",
14   "thematic": "twitter",
15   "object": "source",
16   "temporal_gran": "hours 1",
17   "spatial_gran": "zone z1",
18   "nodesattributes": [
19     ["timestamp", "datetime", ""], ["zone", "string", ""],
20     ["tweets", "string", ""], ["numTweets", "numeric", "" ]]],
21 "data": {
22   "id": "dest1",
23   "name": "destination",
24   "object": "destination",
25   "sources": "j1",
26   "thematic": "humanDisc",
27   "temporal_gran": "hours 1",
28   "spatial_gran": "zone z1",
29   "nodesattributes": [
30     ["zone", "string", ""], ["v1_timestamp", "datetime", ""],
31     ["v1_tweets", "string", ""], ["v1_numTweets", "numeric", "" ],
32     ["v1_numPos", "numeric", "" ], ["c1_timestamp", "datetime", "" ],
33     ["c1_geo:lat", "numeric", "degree"], ["c1_geo:long", "numeric", "degree"],
34     ["c1_humidityVal", "numeric", "percentage" ]],
35   "configuration": ["spark://0.0.0.0:7077", "hours", "1", "http://0.0.0.0:9092"],
36   "storage": ["mongodb", "http://0.0.0.0:27017", "StreamLoader", "Results" ]}

```

Services for the Manipulation of Sensor Data Each service made available in the StreamLoader system is associated with a data object presenting a key object whose value is the name of the service. The attributes of the data object are similar to the

one presented for the sensors. In addition, it provides other keys for the representation of the service parameters.

Consider the four services reported in Figure 7.19. The following JSON representation is generated. As we can see each service provides different parameters and attributes. For example the `filter` service has the `connector` and `filterconditions` attributes that contain the connector (AND or OR) used for binding the filtering conditions. For the `convert` service, the JSON representation contains the initial spatial and temporal granularities (`old_temporal_gran` and `old_spatial_gran`) and the target ones (`temporal_gran` and `spatial_gran`). For the `virtual property` service, the JSON representation contains the name of the new property (`new_property`) and the function (`function`), manually defined by the domain expert, used for calculating its value. The function is represented as a string and it is assigned to the `function` key. For the `join` service, the `nodesattributes` key contains the list of the attributes of the two incoming streams, while the `newnodesattributes` key contains the schema generated by the application of the join operator. The sensor attributes used for the specification of the join condition are reported in the `onattributes` key while the type of join is contained in the `policy` key. Finally, `rates` contains the dimension of the join window.

```

1  "data": {
2    "id": "f1",
3    "name": "filter",
4    "object": "filter",
5    "nodesattributes": [
6      ["timestamp", "datetime", ""], ["zone", "string", ""],
7      ["tweets", "string", ""], ["numTweets", "numeric", ""]],
8    "thematic": "twitter",
9    "temporal_gran": "hours 1",
10   "spatial_gran": "zone z1",
11   "connector": "AND",
12   "sources": "s2",
13   "filterconditions": [
14     ["tweets", "MATCHES", "hot"], ["tweets", "MATCHES", "sweat"],
15     ["tweets", "MATCHES", "heat"]]],
16  "data": {
17    "id": "c1",
18    "name": "convert",
19    "object": "convert",
20    "old_temporal_gran": "minutes 30",
21    "old_spatial_gran": "point",
22    "thematic": "humidity",
23    "sources": "s1",
24    "temporal_gran": "hours 1",
25    "spatial_gran": "zone z1",
26    "nodesattributes": [
27      ["timestamp", "datetime", ""], ["geo:lat", "numeric", "degree"],
28      ["geo:long", "numeric", "degree"], ["humidityVal", "numeric", "percentage"]]],
29  "data": {
30    "id": "v1",
31    "name": "virtual",
32    "object": "virtual_property",
33    "new_property": "numPos",
34    "sources": "f1",

```

```

35 "thematic": "twitter",
36 "temporal_gran": "hours 1",
37 "spatial_gran": "zone z1",
38 "nodesattributes": [
39     ["timestamp", "datetime", ""], ["zone", "string", ""], ["tweets", "string", ""],
40     ["numTweets", "numeric", ""], ["numPos", "numeric", ""]],
41 "function": "tweets.count('hot', 'heat', 'sweat')",
42 "data": {
43     "id": "j1",
44     "name": "join",
45     "object": "join",
46     "sources": ["c1", "v1"],
47     "thematic": "humanDisc",
48     "temporal_gran": "hours 1",
49     "spatial_gran": "zone z1",
50     "nodesattributes": [
51         ["v1_timestamp", "datetime", ""], ["v1_zone", "string", ""],
52         ["v1_tweets", "string", ""], ["v1_numTweets", "numeric", ""],
53         ["v1_numPos", "numeric", ""], ["c1_timestamp", "datetime", ""],
54         ["c1_geo:lat", "numeric", "degree"], ["c1_geo:long", "numeric", "degree"],
55         ["c1_humidityVal", "numeric", "percentage"]],
56     "newnodesattributes": [
57         ["zone", "string", ""], ["v1_timestamp", "datetime", ""],
58         ["v1_tweets", "string", ""], ["v1_numTweets", "numeric", ""],
59         ["v1_numPos", "numeric", ""], ["c1_timestamp", "datetime", ""],
60         ["c1_geo:lat", "numeric", "degree"], ["c1_geo:long", "numeric", "degree"],
61         ["c1_humidityVal", "numeric", "percentage"]],
62     "onattributes": ["v1_location.name", "c1_location.name"],
63     "policies": "inner",
64     "rates": ["1", "hours"]}

```

Edges The sensors, destination, and service nodes so far discussed are connected by means of edges that specify the precedence relationships among the nodes. This precedence needs to be maintained in the generated code in order to guarantee its correct representation. Edges are represented by means of data objects containing two properties `source` and `target` besides the object identifier. The following code contains the representation of the edges of our running example.

```

1 "data": { "id": "s1c1", "source": "s1", "target": "c1" }
2 "data": { "id": "s2f1", "source": "s2", "target": "f1" }
3 "data": { "id": "f1v1", "source": "f1", "target": "v1" }
4 "data": { "id": "c1j1", "source": "c1", "target": "j1" }
5 "data": { "id": "v1j1", "source": "v1", "target": "j1" }
6 "data": { "id": "j1dest1", "source": "j1", "target": "dest1" }

```

7.4.2 Configuration

In order to efficiently manipulate the information coming from the sensors, each sensor is treated as a scala `case class`. Case classes are like regular classes which export their constructor parameters and provide a recursive decomposition mechanism via pattern matching that is particularly useful for modeling immutable data. The use of scala `case class` allows domain expert to easily define the attributes (with the corresponding types) obtained by the application of the service. In our system, classes are created to handle sensor data in our internal data model and for the representation of the STT dimensions. Specifically, the `Location` case class models the latitude, longitude and zone, province or region name where an event has been generated. The STT dimensions are defined by the case class `Stt` that contains the properties: `spatial`, `temporal` and `thematic`. The object values associated with these properties are instances of the case class `Spatio`, `Time` and `Thematic`. These classes are defined at the beginning of the code because are common to every DAP and used for the representation of the sensors.

For what concern the Spark and Kafka communication environment, it is necessary to configure the Spark cluster and the Kafka server correctly. The `sparkConf` property provides the Spark Context, the host of the Master node in the Spark cluster, while the `ssc` property sets the size of the streaming batch. The size of the batch is defined manually by the domain expert and it is a multiple of the temporal granularity of the last service that compose the DAP. These settings of the configuration are provided by the `destination` node of the JSON representation of the DAP. The following Kafka parameters are defined in the `kafkaParams` key: *i*) the Kafka server URL; *ii*) the `group.id`, that is a string that uniquely identifies the group of consumer processes to which this consumer belongs for balance the records over the consumer instances; *iii*) the use of a `Deserializer` (keyword used to identify a Kafka Consumer); and, *iv*) the `auto.offset.reset`, that is the policy used for consuming data when a failure occurs in a topic (from the beginning or from the "point of failure").

The following code presents the case classes for our running example and the configuration of the Spark and Kafka environment.

```

1 case class Thematic(
2     name: String,
3     metadata: JObject)
4 case class Time(

```

```

5     unit: String,
6     count: Double,
7     metadata: JObject)
8 case class Spatio(
9     unit: String,
10    metadata: JObject)
11 case class Stt(
12    spatial: Spatio,
13    temporal: Time,
14    thematic: Thematic)
15 case class Location(
16    latitude: Double,
17    longitude: Double,
18    name: String)
19
20 val sparkConf = new SparkConf().setAppName("SparkScript").setMaster("spark://0.0.0.0:7077")
21 val ssc = new StreamingContext(sparkConf, Hours(1))
22
23 val kafkaParams = Map[String, Object](
24    "bootstrap.servers" -> "http://0.0.0.0:9092",
25    "key.deserializer" -> classOf[StringDeserializer].getCanonicalName,
26    "value.deserializer" -> classOf[StringDeserializer].getCanonicalName,
27    "group.id" -> "test_luca",
28    "auto.offset.reset" -> "latest")

```

Internally, the Apache Spark script works as follows. Spark Streaming receives live input data streams and divides the data into batches of a fixed dimension specified in the `StreamingContext`. Spark Streaming provides a high-level abstraction called *discretized stream* or *DStream*, which represents a continuous stream of data. DStreams, in our case, are input data streams generated by Kafka. Internally, a DStream is represented as a sequence of RDDs.

7.4.3 Translation of Sources and Destination

As described above we use `case classes` to manipulate the data model of each sensor and common information about each sensor are instantiated at the beginning of the code. Since each sensor presents a specific schema of data and topic, two case classes are created for each sensor included in the DAP. The first one, named `Data` concatenated to the `Id` of the sensor, contains the properties (name and type) of the sensor. The second one, named `Sensor` concatenated to the `Id` of the sensor, contains the following properties: *i*) `sensor_name`, the name of the sensor; *ii*) `start_date` and `end_date`, representing the interval of time in which the sensor produces data; *iii*) `data`, it is the schema of the data produced by the sensor and its type is the `Data` case class that has been dynamically defined; and, *iv*) `stt`, a property of type `Stt` that reports the spatio-temporal granularity and the thematic of the sensor. Moreover, a specific topic is created in Kafka for gathering the data produced by the sensor and made available through Kafka.

Once a sensor is connected with a stream of Kafka records, it is needed its parsing in the right format defined by the `Sensor case class`. If the incoming data are already in the internal format we do not need to perform any other specific operation. Otherwise, it is necessary to apply the transformation rules in order to convert the sensor format into the internal data model, and then parse it in the right format defined by the `Sensor case class`. An example of a sensor in our running example is provided by the code below.

```

1 //Source s1
2 case class Data_s1(
3     location : Location,
4     timestamp : String,
5     humidityVal : Double)
6 case class Sensor_s1(
7     sensor_name: String,
8     start_date: String,
9     end_date: String,
10    data: Data_s1,
11    stt: Stt)
12 //Topic s1
13 val topics_s1 = Array("topics1")
14 //Kafka Stream s1
15 val stream_s1 = KafkaUtils.createDirectStream[String, String](ssc, PreferConsistent, Subscribe[String,
16     String](topics_s1, kafkaParams))
17 //Sensor map s1
18 val s1 = stream_s1.map(record => { implicit val formats = DefaultFormats
19     parse(record.value).extract[Sensor_s1]})

```

As we can see from the code, the schema of the data is defined by the class `Data_s1` and the class `Sensor_s1`, representing the internal data model of the specific sensor, is defined. The sensor needs to be initialized as a Kafka Consumer. The `KafkaUtils.createDirectStream` takes as parameters the Spark Context, a `LocationStrategies.PreferConsistent`, in order to distribute partitions evenly across available executors and a `Subscribe` method, where information about the topic and the Kafka environment must be provided and instantiate a new Kafka stream. In the last part of the code, the sensor already transformed before the transmission, is parsed into the internal data model.

For what concern the destination, besides providing configuration information, it contains also information about the storage of the processed data into a database. The `storage` attribute included in the JSON representation of the DAP allows to generate the code below in the Spark script. `MongodbConfig.Host`, `mongoDbDatabase` and `mongoDbCollection` contain the configuration of the DB while in the `foreachRDD` every processed tuple is converted into a specific schema and then saved on the `mongodb` database.

```

1 val mongoDbFormat = "com.stratio.datasource.mongodb"
2 val mongoDbDatabase = "StreamLoader"
3 val mongoDbCollection = "Results"
4
5 val MongoDbOptiops = Map(MongodbConfig.Host -> "http://0.0.0.0:27017",

```

```

6   MongoddbConfig.Database -> mongoDbDatabase,
7   MongoddbConfig.Collection -> mongoDbCollection)
8
9   val dest = j1 .foreachRDD { rdd =>
10    val destination=spark.read.schema(Sensorj1).json(rdd)
11    destination.write.format(mongoDbFormat).mode(SaveMode.Append).options(MongoDbOptions).save()

```

7.4.4 Translation of Services

We describe now the format of every service in the translation. We use the variables PN, PN1 or PN2 in order to define the id of the node associated with the service. The operations are executed on each *DStream* which is a sequence of Spark RDD collected in every batch.

Filter. This operator takes in input two parameters: the *filtering conditions* and the *connector* that concatenate every conditions. The default connector is AND and if there is only one condition it is omitted in the Spark generation.

Suppose we wish to filter the temperature events whose values are contained in the interval 20 and 30 Celsius degrees. The following code is generated.

```

1   val f1 = PN.filter { r => r.data.temperatureVal.>(20) && r.data.temperatureVal.<(30) }

```

The variable *r* in the `filter` method represents a single tuple in the sequence of RDDs that is currently processed. The result of its execution is "stored" in a new *DStream* named `f1`.

Virtual Property. This service requires two parameters to be executed: the name of the new property and the function to compute the value to be assigned to the new property. Since this service modifies the schema of the sensor, a new `Sensor` case class is introduced. This class has the same structure of the incoming data flow plus the new attribute. The code allows to copy all the incoming information to the new data structure and to assign the calculated value to the new attribute.

In our running example we add a new property `numPos` to the `Tw1` sensor for counting the number of tweets that contain the words "hot", "heat" and "sweat". As shown in the code below, the new schema is provided by the case class `Data_v1` and the new `Sensor` class `Sensor_v1` is defined. The function for computing the value of the property `numPos` is defined as `(r.data.tweets.count((_ == 'hot') || (_ == 'heat') || (_ == 'sweat')))`.

```

1   case class Data_v1(
2     location : Location,
3     timestamp : String,
4     zone : String,

```

```

5     tweets : String,
6     numTweets : Double,
7     numPos : Double)
8 case class Sensor_v1(
9     sensor_name: String,
10    start_date: String,
11    end_date: String,
12    data: Data_v1,
13    stt: Stt)
14 val v1 = PN.map { r => new Sensor_v1(r.sensor_name, r.start_date, r.end_date,
15    new Data_v1(new Location(r.data.location.latitude,
16    r.data.location.longitude,
17    r.data.location.name),
18    r.data.timestamp,
19    r.data.zone,
20    r.data.tweets,
21    r.data.numTweets, (r.data.tweets.count(_ == 'hot') || (_ == 'heat') || (_ == 'sweat'))),
22    new Stt(new Spatio(r.stt.spatial.unit, r.stt.spatial.metadata),
23    new Time(r.stt.temporal.unit,
24    r.stt.temporal.count,
25    r.stt.temporal.metadata),
26    new Thematic(r.stt.thematic.name, r.stt.thematic.metadata))}

```

Transform. This operator is similar to the Virtual Property service but the functions are not manually defined, they are taken from a list of functions stored in a DB. It takes in input an array composed of: *i*) the attribute on which the function must be applied; *ii*) the name of the function; *iii*) the value to replace (optional, used on the *replace* function); *iv*) the replaced value (optional, used on the *replace* function); and *v*) a Spark code of a function (optional, used for some specific transformation like Celsius to Fahrenheit or Fahrenheit to Celsius).

Suppose that one of the sensor produces temperatures in Fahrenheit and we need to generate the corresponding temperatures in Celsius. It is necessary to apply a function that convert the value in Fahrenheit to Celsius. The following code presents this transformation applied to the `temperatureVal` attribute with the function `r.data.temperatureVal.*(1.8).+(32)`.

```

1 val t1 = PN.map { r => new Sensor_PN(
2     r.sensor_name,
3     r.start_date,
4     r.end_date,
5     new Data_PN(
6         new Location(r.data.location.latitude,
7             r.data.location.longitude,
8             r.data.location.name),
9         r.data.timestamp,
10        r.data.temperatureVal.*(1.8).+(32)),
11    new Stt(new Spatio(r.stt.spatial.unit,
12        r.stt.spatial.metadata),
13    new Time(r.stt.temporal.unit,
14        r.stt.temporal.count,
15        r.stt.temporal.metadata),
16    new Thematic(r.stt.thematic.name,

```

17

r . stt . thematic . metadata))))))

Aggregation. The following parameters should be specified in the JSON representation of the service for the application of the aggregation: *i*) the attributes on which grouping the events; *ii*) the aggregation function to apply to the remaining attributes; and *iii*) the temporal dimension of the aggregation.

Given a sensor that produces data about temperature every 10 seconds on different zones, it could be necessary, for example, to have the minutes average value of temperature of that sensor and it should be aggregated on the same location name. The function `reduceByKeyAndWindow` shown in the code below allows to apply auxiliary functions to the events that have been grouped together. These functions are specific for any basic data type (e.g. integer, string, double) and allow to identify a specific element (e.g. the first, the last) or to apply an aggregation (e.g. AVG, MAX, MIN). For example, if we group the events according to the location name, in a single group we might have different values associated with the attribute `sensor_name` (which is of type string). By means of the auxiliary function `RetStr` we impose to return the last sensor name among those occurring in the current processed `DStream`. Moreover, we can apply the auxiliary function `AVGD` on the attribute `temperatureVal` which is of type `Double` for returning the average temperature.

```

1  val a1map = PN.map { r => (r.data.location.name,
2      (r.sensor_name, r.start_date, r.end_date, r.data.location.latitude,
3      r.data.location.longitude, r.data.temperatureVal, r.data.timestamp, r.stt.spatial.unit,
4      r.stt.spatial.metadata, r.stt.temporal.unit, r.stt.temporal.count, r.stt.temporal.metadata,
5      r.stt.thematic.name, r.stt.thematic.metadata)) }
6  val a1red = a1map.reduceByKeyAndWindow(
7      (a:(String, String, String, Double,
8      Double, Double, Long, String,
9      JObject, String, Int, JObject,
10     String, JObject),
11     b:(String, String, String, Double,
12     Double, Double, Long, String,
13     JObject, String, Int, JObject,
14     String, JObject)
15     )=> (retStr(a._1,b._1), retStr(a._2,b._2),
16         retStr(a._3,b._3), retD(a._4,b._4),
17         retD(a._5, b._5), AVGD(a._6,b._6),
18         retLong(a._7,b._7), retStr(a._8,b._8),
19         retMeta(a._9,b._9), retStr(a._10,b._10),
20         retI(a._11,b._11), retMeta(a._12,b._12),
21         retStr(a._13,b._13), retMeta(a._14,b._14)),
22     Minutes(1))
23  val a1 = a1red.map(x => new Sensor_PN(x._2._1, x._2._2, x._2._3
24      new Data_PN(new Location(x._2._4, x._2._5, x._1), x._2._6, x._2._7),
25      new Stt(new Spatio(x._2._8, x._2._9),
26      new Time(x._2._10, x._2._11, x._2._12),
27      new Thematic(x._2._13, x._2._14))))

```

Join. This service applies the join operation between two incoming streams PN1 and PN2. The two streams need to be specified at the same spatio-temporal granularities. In the JSON representation of this service, the join predicate is specified along with the kind of join to be applied and the dimension of the join window. The two nodes that needs to be joined are defined as a couple of *DStreams*. The first element of the couple corresponds to the type of the join attribute whereas the second element has the same type of the class sensor of the node itself. The window is not applied directly on the join because it introduces further delays in the processing time but it is applied on the result of each couple of *DStream* previously defined. After these phases the join is applied. A new class is defined that present the attributes of the join predicate and all the PN1 and PN2 specific attributes. A prefix is added to the name of the attributes of the new class for distinguish them on the base of the node from which they have been taken. This operation is necessary to map the result of the join into the new `sensor case class`.

Consider our running example. The streams produced by the sensors H1 and Tw1 need to be joined each hour according to the `x.data.location.name` attributes. As we can see from the following code, the join attribute for each source is specified and the result is a window of *DStream* couples (`j1PN1win` and `j1PN2win`). The join operation on line 5 introduces a new schema of the data. For this reason the new `Sensor_j1 case class` is defined and the attributes of the resulting tuples are mapped to this class.

```

1  val j1PN1 = PN1.map(x => (x.data.location.name, (x)))
2  val j1PN2 = PN2.map(x => (x.data.location.name, (x)))
3  val j1PN1win = j1PN1.window(Hours(1), Hours(1))
4  val j1PN2win = j1PN2.window(Hours(1), Hours(1))
5  val j1pre = j1PN1win.join(j1PN2win)
6  case class Data_j1(
7      location : Location,
8      zone : String,
9      v1_timestamp : String,
10     v1_tweets : String,
11     v1_numTweets : Double,
12     v1_numPos : Double,
13     c1_timestamp : String,
14     c1_humidityVal : Double)
15 case class Sensor_j1(
16     sensor_name: String,
17     start_date: String,
18     end_date: String,
19     data: Data_j1,
20     stt : Stt)
21 val j1 = j1pre.map { r => new Sensor_j1("j1",
22     r._2._1.start_date,
23     r._2._1.end_date,
24     new Data_j1(new Location(r._2._1.data.location.latitude,
25     r._2._1.data.location.longitude,
26     r._2._1.data.location.name),
27     r._2._2.data.zone,
28     r._2._2.data.timestamp,
29     r._2._2.data.tweets,
```

```

30         r._2._2.data.numTweets,
31         r._2._2.data.numPos,
32         r._2._1.data.timestamp,
33         r._2._1.data.humidityVal),
34     new Stt(new Spatio(r._2._1.stt.spatial.unit,
35                     r._2._1.stt.spatial.metadata),
36     new Time(r._2._1stt.temporal.unit,
37             r._2._1.stt.temporal.count,
38             r._2._1.stt.temporal.metadata),
39     new Thematic(r._2._1.stt.thematic.name,
40                 r._2._1.stt.thematic.metadata)))}

```

Enrich. This service is a special case of the join operator between stream and static data. The Spark application and execution of this operation is similar to the join. It is done by defining a join attribute between the stream and the static information.

The first thing that we have to do is load a table and cache it so that it can be joined with streams. Then, as the join requires, we need to identify the "join" attributes for the stream and for the static data. At this point we proceed as we have done for the join service.

Suppose we wish to enrich sensor T2 with data contained in the LHD table that contains a tuple for each zone of Milan. The following code is generated.

```

1  val lhdInfo = sqlContext.table("LHD").rdd.map(row => (row(0).toString(),
2                    row(1).toDouble()), partitionBy(partitioner) .cache()
3  val e1PN1 = PN1.map(x => (x.data.location.name, (x)))
4  val e1PN2 = lhdInfo.map(x => (x.(0), (x)))
5  val e1PN1win = e1PN1.window(Minutes(20), Minutes(20))
6  val e1pre = e1PN1win.join(e1PN2)
7  case class Data_e1(
8      location : Location,
9      timestamp : String,
10     temperatureVal : String,
11     accuracy : Double,
12     lhd: Double)
13  case class Sensor_e1(
14     sensor_name: String,
15     start_date: String,
16     end_date: String,
17     data: Data_e1,
18     stt : Stt)
19  val e1 = e1pre.map { r => new Sensor_e1(
20     r._2._1.sensor_name,
21     r._2._1.start_date,
22     r._2._1.end_date,
23     new Data_e1(new Location(
24         r._2._1.data.location.latitude,
25         r._2._1.data.location.longitude,
26         r._2._1.data.location.name),
27         r._2._1.data.timestamp,
28         r._2._1.data.temperatureVal,
29         r._2._1.data.accuracy,
30         r._2._2._2),
31     new Stt(new Spatio(r._2._1.stt.spatial.unit, r._2._1.stt.spatial.metadata),

```

```

32         anew Time(r._2._1.stt.temporal.unit,
33                 r._2._1.stt.temporal.count,
34                 r._2._1.stt.temporal.metadata),
35         new Thematic(r._2._1.stt.thematic.name,
36                    r._2._1.stt.thematic.metadata)))))

```

Trigger On/Off. This service is a composition of Trigger Event and Trigger Actions. The Trigger works on two different streams of data. The first one is used to verify certain conditions on a specific time interval. The second one is activated only if the trigger conditions on the first flow are verified. Depending on the type of trigger condition that has been verified a specific type of Trigger Actions is activated.

Suppose to have two sensors H1 and T1 that produce respectively humidity and temperature values and we wish to obtain the humidity values greater than 50%, produced by sensor H1, only if in the last hour the number of temperature values less than 15 degrees produced by sensor T1 exceeds 10. In the following code we define a trigger event boolean value `teon` that allows the script to activate the Trigger Action ON if true.

```

1  var teon = false
2
3  val f1 = t1.filter { r => r.data.temperatureVal.<(15)}
4  val f1win = f1.window(Hours(1), Hours(1))
5  f1win.foreachRDD{ rdd =>
6    var count = rdd.count
7    teon = count > 10
8    count = 0
9  }
10
11 if(teon){
12   val ta = h1.filter {r => r.data.humidityVal.>(50)}
13 }

```

Union. This service, as the `join`, takes in input two incoming streams. If the schema of the data of the two streams is exactly the same it does not introduce a new `sensor case class` otherwise a new `sensor case class` has to be defined. The schema of the new class is composed of all the attributes that the two sensors have in common, taken only one time, and the attributes that are not present in one or in the other sensor.

Consider our running example. The temperatures produced by sensors T1 and T2 need to be included in a single stream. The STT dimensions of the two sources are the same with the exception of accuracy that is missing in T2. In order to create the union of the values produced by the two sensors this attribute needs to be included in the schema of T2 with the values 0.0. The code below shows that a new `sensor`

`case class` is defined and the two sources are mapped to the new `case class`. After that an union is performed every hour on the two *DStreams*.

```

1  case class Data_u1(
2      location : Location,
3      timestamp : String,
4      temperatureVal : String,
5      accuracy : Double)
6  case class Sensor_u1(
7      sensor_name: String,
8      start_date: String,
9      end_date: String,
10     data: Data_u1,
11     stt: Stt)
12  val u1PN1 = PN1.map { r => new Sensor_u1(
13
14     r.sensor_name,
15     r.start_date,
16     r.end_date,
17     new Data_u1(new Location(
18         r.data.location.latitude,
19         r.data.location.longitude,
20         r.data.location.name),
21     r.data.timestamp,
22     r.data.temperatureVal,
23     r.data.accuracy),
24     new Stt(new Spatio(r.stt.spatial.unit,
25         r.stt.spatial.metadata),
26         new Time(r.stt.temporal.unit,
27             r.stt.temporal.count,
28             r.stt.temporal.metadata),
29         new Thematic(r.stt.thematic.name,
30             r.stt.thematic.metadata))})
31  val u1PN2 = PN2.map { r => new Sensor_u1(
32
33     r.sensor_name,
34     r.start_date,
35     r.end_date,
36     new Data_u1(new Location(
37         r.data.location.latitude,
38         r.data.location.longitude,
39         r.data.location.name),
40     r.data.timestamp,
41     r.data.temperatureVal,
42     0.0),
43     new Stt(new Spatio(r.stt.spatial.unit,
44         r.stt.spatial.metadata),
45         new Time(r.stt.temporal.unit,
46             r.stt.temporal.count,
47             r.stt.temporal.metadata),
48         new Thematic(r.stt.thematic.name,
49             r.stt.thematic.metadata))})
50  val u1 = u1PN1.union(u1PN2).window(Hours(1), Hours(1))

```

Convert. This service is responsible of converting the temporal and spatial granularities. It is not mandatory that both new granularity must be defined. Rules that allows to efficiently and correctly convert the temporal and spatial granularities are stored on collections in a DB. These collections define the hierarchy of granularity

(i.e. seconds is finer than minutes, city coarser than zone) and the way to compute the conversion (i.e. a minute is composed of 60 seconds, city Milan is composed of 9 zones).

Suppose that we need to convert the temporal granularity of the sensor T1. First of all we need to identify the attribute that provides information about timestamp. Then, the conversion is realized by means of the `reduceByKeyAndWindow` method. It takes in input the sequence of *RDDs* and converts them to the new temporal granularity. Example of this operator is presented on the code below.

```

1  case class Data_c1(
2      location : Location,
3      timestamp : String,
4      humidityVal : Double)
5  case class Sensor_c1(
6      sensor_name: String,
7      start_date: String,
8      end_date: String,
9      data: Data_s1,
10     stt : Stt)
11 val c1map = PN.map(x => (x.data.timestamp, (x)))
12 val c1pre = c1map.reduceByKeyAndWindow((a:Int, b:Int) => (a+b), Minutes(20), Hours(1))
13 val c1 = c1pre.map { r => new Sensorc1(
14     r.sensor_name,
15     r.start_date,
16     r.end_date,
17     new Data_c1(new Location(
18         r.data.location.latitude,
19         r.data.location.longitude,
20         r.data.location.name),
21     r.data.timestamp,
22     r.data.humidityVal),
23     new Stt(new Spatio(r.stt.spatial.unit, r.stt.spatial.metadata),
24     new Time(r.stt.temporal.unit,
25         r.stt.temporal.count,
26         r.stt.temporal.metadata),
27     new Thematic(r.stt.thematic.name, r.stt.thematic.metadata)))))

```

7.4.5 The Overall Translation Algorithm

We now provide the description of how an entire DAP is translated into an Apache Spark Streaming Scala script. The most important issue in the translation of a DAP is to maintain the order of the operators as defined in the DAP. In our system we demand the right "ordering" of the DAP to the internal way of handling *RDDs* by Apache Spark. As described in [81] Spark implements a *stage-oriented scheduling* that transforms a Logical Execution Plan (i.e. *RDD lineage of dependencies*) to a Physical Execution Plan through a *DAG scheduler*. *RDD Lineage* (also named *RDD operator graph* or *RDD dependency graph*) is a graph of all the parent *RDDs* of a *RDD*. Each *RDD* maintains a pointer to one or more parents along with the metadata about what type of relationship it has with the parent. For example, when we call

`val b = a.map()` on a RDD, the RDD `b` keeps a reference to its parent `a`, that is a lineage. It is built as a result of applying transformations to the RDD and creates a Logical Execution Plan [82]. A Logical Execution Plan starts with the earliest RDDs (those with no dependencies on other RDDs or reference cached data) and ends with the RDD that produces the result of the action that has been called to execute. In this way the execution order is maintained also if the operations are written in the Apache Spark script without following the right order as they appear in the DAP.

Anyway in order to have a clearer and more maintainable script in the translation phase we first provide all the information about the imports of packages and libraries that are required for the right execution of the code, then the information about the configuration of the Spark and Kafka environment are included and subsequently all the sensor nodes are written on the script. The nodes that represent the operations are then translated and included in the code while in the last part we provide the translation of the storage information defined by the destination node.

The example that we provide on Figure 7.19 will be translated as follows.

```

1 object App {
2   def main(args : Array[String]) {
3     case class Thematic(
4       name: String,
5       metadata: JObject)
6     case class Time(
7       unit: String,
8       count: Double,
9       metadata: JObject)
10    case class Spatio(
11      unit: String,
12      metadata: JObject)
13    case class Stt(
14      spatial: Spatio,
15      temporal: Time,
16      thematic: Thematic)
17    case class Location(
18      latitude: Double,
19      longitude: Double,
20      name: String)
21
22    val sparkConf = new SparkConf().setAppName("SparkScript").setMaster("spark://0.0.0.0:7077")
23    val ssc = new StreamingContext(sparkConf, Hours(1))
24
25    val kafkaParams = Map[String, Object](
26      "bootstrap.servers" -> "http://0.0.0.0:9092",
27      "key.deserializer" -> classOf[StringDeserializer].getCanonicalName,
28      "value.deserializer" -> classOf[StringDeserializer].getCanonicalName,
29      "group.id" -> "test_luca",
30      "auto.offset.reset" -> "latest"
31    )
32    //Source s1
33    case class Data_s1(
34      location : Location,
35      timestamp : String,
36      humidityVal : Double)
37    case class Sensor_s1(

```

```

38     sensor_name: String,
39     start_date: String,
40     end_date: String,
41     data: Data_s1,
42     stt : Stt)
43 //Topic s1
44 val topics_s1 = Array("topics1")
45 //Kafka Stream s1
46 val stream_s1 = KafkaUtils.createDirectStream[String, String](ssc, PreferConsistent, Subscribe[String,
47     String](topics_s1, kafkaParams))
48 //Sensor map s1
49 val s1 = stream_s1.map(record => { implicit val formats = DefaultFormats
50     parse(record.value).extract[Sensor_s1]})
51 //Source s2
52 case class Data_s2(
53     location : Location,
54     timestamp : String,
55     zone : String,
56     tweets : String,
57     numTweets : Double)
58 case class Sensor_s2(
59     sensor_name: String,
60     start_date: String,
61     end_date: String,
62     data: Data_s2,
63     stt : Stt)
64 //Topic s2
65 val topics_s2 = Array("topics2")
66 //Kafka Stream s2
67 val stream_s2 = KafkaUtils.createDirectStream[String, String](ssc, PreferConsistent, Subscribe[String,
68     String](topics_s2, kafkaParams))
69 //Sensor map s2
70 val s2 = stream_s2.map(record => { implicit val formats = DefaultFormats
71     parse(record.value).extract[Sensor_s2]})
72 //Filter f1
73 val f1 = s2.filter { e => e.data.tweets.contains("hot") && e.data.tweets.contains("sweat")
74     e.data.tweets.contains("heat") }
75 //Convert c1
76 case class Data_c1(
77     location : Location,
78     timestamp : String,
79     humidityVal : Double)
80 case class Sensor_c1(
81     sensor_name: String,
82     start_date: String,
83     end_date: String,
84     data: Data_c1,
85     stt : Stt)
86 val c1map = s1.map(x => (x.data.timestamp, (x)))
87 val c1pre = c1map.reduceByKeyAndWindow((a: Int, b: Int) => (a+b), Minutes(30), Hours(1))
88 val c1 = c1pre.map { r => new Sensor_c1(r.sensor_name, r.start_date, r.end_date, new Data_c1(new
89     Location(r.data.location.latitude, r.data.location.longitude, "z1"), r.data.timestamp,
90     r.data.humidityVal), new Stt(new Spatio("zone", r.stt.spatial.metadata), new Time(r.stt.temporal.unit,
91     r.stt.temporal.count, r.stt.temporal.metadata), new Thematic(r.stt.thematic.name,
92     r.stt.thematic.metadata)))}
93 //Virtual v1
94 case class Data_v1(
95     location : Location,
96     timestamp : String,
97     zone : String,

```

```

89     tweets : String,
90     numTweets : Double,
91     numPos : Double)
92 case class Sensor_v1(
93     sensor_name: String,
94     start_date: String,
95     end_date: String,
96     data: Data_v1,
97     stt : Stt)
98 val v1 = f1.map { r => new Sensor_v1(r.sensor_name, r.start_date, r.end_date, new Data_v1(new
    Location(r.data.location.latitude, r.data.location.longitude, r.data.location.name), r.data.timestamp,
    r.data.zone, r.data.tweets, r.data.numTweets, (r.data.tweets.count(_ == 'hot') || (_ == 'heat') || (_
    == 'sweat')))), new Stt(new Spatio(r.stt.spatial.unit, r.stt.spatial.metadata), new
    Time(r.stt.temporal.unit, r.stt.temporal.count, r.stt.temporal.metadata), new
    Thematic(r.stt.thematic.name, r.stt.thematic.metadata))})
99 //Join j1 rates: 1 hours
100 val j1c1 = c1.map(x => (x.data.location.name, (x)))
101 val j1v1 = v1.map(x => (x.data.location.name, (x)))
102 val j1c1win = j1c1.window(Hours(1), Hours(1))
103 val j1v1win = j1v1.window(Hours(1), Hours(1))
104 val j1pre = j1c1win.join(j1v1win)
105 case class Data_j1(
106     location : Location,
107     zone : String,
108     v1_timestamp : String,
109     v1_tweets : String,
110     v1_numTweets : Double,
111     v1_numPos : Double,
112     c1_timestamp : String,
113     c1_humidityVal : Double)
114 case class Sensor_j1(
115     sensor_name: String,
116     start_date: String,
117     end_date: String,
118     data: Data_j1,
119     stt : Stt)
120 val j1 = j1pre.map { r => new Sensor_j1(j1, r._2._1.start_date, r._2._1.end_date, new Data_j1(new
    Location(r._2._1.data.location.latitude, r._2._1.data.location.longitude, r._2._1.data.location.name),
    r._2._2.data.zone, r._2._2.data.timestamp, r._2._2.data.tweets, r._2._2.data.numTweets,
    r._2._2.data.numPos, r._2._1.data.timestamp, r._2._1.data.humidityVal), new Stt(new
    Spatio(r.stt.spatial.unit, r.stt.spatial.metadata), new Time(r.stt.temporal.unit,
    r.stt.temporal.count, r.stt.temporal.metadata), new Thematic(r.stt.thematic.name,
    r.stt.thematic.metadata))})
121 //Destination
122 val mongoDbFormat = "com.stratio.datasource.mongodb"
123 val mongoDbDatabase = "StreamLoader"
124 val mongoDbCollection = "Results"
125 val MongoDbOptiops = Map(MongodbConfig.Host -> "http://0.0.0.0:27017",
126     MongodbConfig.Database -> mongoDbDatabase,
127     MongodbConfig.Collection -> mongoDbCollection)
128 val dest = j1.foreachRDD { rdd =>
129     val destination=spark.read.schema(Sensorj1).json(rdd)
130     destination.write.format(mongoDbFormat).mode(SaveMode.Append).options(MongoDbOptiops).save()
131 }
132 //Execution of the script
133 ssc.start ()
134 ssc.awaitTermination()
135 }
136 }

```

Chapter 8

Experimental Evaluation

In this chapter we provide the experiments that we have carried out for evaluating the performances of the Apache Spark Streaming scripts generated by means of the StreamLoader system. These experiments are mainly focused on the evaluation of the performances of the generated scripts and to determine the number of events that can be handled for each second by our Data Acquisition Plans. The experiments are oriented both in the evaluation of the Data Acquisition Plan when the data produced by the sensors are already organized according to our internal data model and when the transformation rules presented in Chapter 5 need to be applied for the transformation of the data from the original format into our internal representation.

The chapter is organized as follows. Section 8.1 presents the environment that we have exploited for the experiments and the metrics that will be used for comparing the performances of the Apache Spark scripts generated from the Data Acquisition Plans. Section 8.2 discusses the experiments in the execution of simple Data Acquisition Plans in a single machine or in a cluster of machines. In Section 8.3 we describe further experiments that we conducted to better understand the scalability of Apache Spark Streaming and to determine the number of events that can be handled. Section 8.4 deals with Data Acquisition Plans that require the transformation of the sensors data into the internal format. The section discusses two architectures in which the transformation rules can be applied and the associated advantages and disadvantages. Some concluding remarks are finally drawn in Section 8.5

8.1 Environment and Tests Configuration

In this section we provide details on the machines used for conducting our experiments and on the kinds of experiments that will be shown in the chapter along with the metrics for evaluating the performances of our system.

8.1.1 Local and Cluster Configuration

Experiments have been conducted on a single machine and in a cluster with a variable number of machines (3, 5 and 10 machines). Figure 8.1 represents the two approaches. In the remainder we detail the characteristics of the machines used.

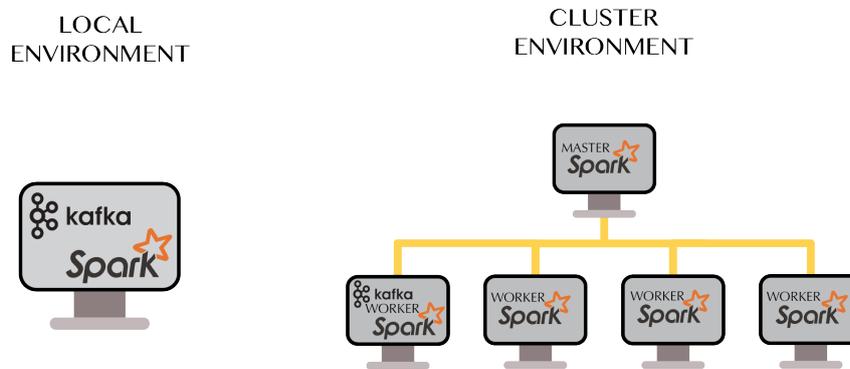


FIGURE 8.1: Local execution and cluster execution environments

- Local: an Ubuntu 16.04 LTS (GNU/Linux 4.4.0-96-generic x86_64) machine with 8GB RAM, 2 core processor, 250 GB HDD and 2799.202 MHz CPU clock speed. Apache Kafka, Kafka Producer and Apache Spark run on this machine.
- Cluster 3: three Ubuntu 16.04 LTS (GNU/Linux 4.4.0-96-generic x86_64) machines with 8GB RAM, 2 core processor, 250 GB HDD and 2799.202 MHz CPU clock speed. A machine acts as *Master* while two machine are the workers/slaves. In order to reduce the CPU consumption we run Apache Kafka Server and the Kafka Producer on a machine and the Apache Spark script on a different machine.
- Cluster 5: five Ubuntu 16.04 LTS (GNU/Linux 4.4.0-96-generic x86_64) machines with 8GB RAM, 2 core processor, 250 GB HDD and 2799.202 MHz CPU clock speed. A machine acts as *Master* while four machine are the workers/slaves. In order to reduce the CPU consumption we run Apache Kafka Server and the Kafka Producer on a machine and the Apache Spark script on a different machine.
- Cluster 10: ten Ubuntu 16.04 LTS (GNU/Linux 4.4.0-96-generic x86_64) machines with 8GB RAM, 2 core processor, 250 GB HDD and 2799.202 MHz CPU clock speed. A machine acts as *Master* while nine machine are the workers/slaves. In order to reduce the CPU consumption we run Apache Kafka Server and the Kafka Producer(s) on a machine and the Apache Spark script on a different machine.

All the machines are virtual machines that reside on an **Ovirt** Virtual Datacenter¹

¹oVirt is a complete open-source virtualization management platform that builds on the powerful kernel based virtual machine (KVM hypervisor) and on the RHEV-M management server. Details in <https://www.ovirt.org/>.

Completed Batches (last 202 out of 202)

Batch Time	Input Size	Scheduling Delay ⁽¹⁾	Processing Time ⁽¹⁾	Total Delay ⁽¹⁾	Output Ops: Succeeded/Total
2017/11/07 12:05:54	214888 records	10 s	9 s	20 s	1/1
2017/11/07 12:05:48	219944 records	6 s	10 s	16 s	1/1
2017/11/07 12:05:42	193080 records	4 s	8 s	12 s	1/1
2017/11/07 12:05:36	193048 records	1 s	9 s	10 s	1/1
2017/11/07 12:05:30	119896 records	1 s	6 s	8 s	1/1
2017/11/07 12:05:24	80960 records	2 s	6 s	7 s	1/1

FIGURE 8.2: Batch information provided by the Spark UI interface

8.1.2 Data Acquisition Plan Configuration and Metrics

When a Spark script is launched, its activity is monitored by the SparkContext daemon. The execution activities can be shown to the user by means of a web application that displays useful information about the application. Among the information reported to the user, the most useful in our context are:

- a list of scheduler stages and tasks,
- a summary of RDD sizes and memory usage,
- environmental information, and
- information about the running executors.

For what concern the execution of an Apache Spark script, the Spark UI interface provides these metrics: *Processing Time*, *Scheduling Delay* and *Total Delay*.

- *Processing Time*: The time required to compute a given batch for all its jobs, end to end. This means a single job which starts at the first operation in the script and ends at the last operation, and assumes as a prerequisite that the job has been submitted.
- *Scheduling Delay*: The time taken by the Spark Streaming scheduler to submit the jobs of the batch. If the batch reads from the source every 4 seconds and a given batch takes 8 seconds to compute this means that we are now $8 - 4 = 4$ seconds behind, thus making the Scheduling Delay 4 seconds long.
- *Total Delay*: it is composed of Scheduling Delay + Processing Time. Following the same example, if we are 4 seconds behind, meaning our Scheduling Delay is 4 seconds, and the next batch takes another 8 seconds to compute, this means that the Total Delay is now $8 + 4 = 12$ seconds long.

Figure 8.2 shows an excerpt of the Web application. For each batch reports the time in which it has been scheduled, the number of tuples that need to be processed in the batch, the Scheduling Delay, the Processing Time and total times required for processing the batch and the execution result.

8.2 Data Acquisition Plan Execution Experiments

We conducted a set of experiments in order to provide information about the performances of the Apache Spark Data Acquisition Plan that is generated with our interface when it is executed locally and on a cluster of different dimension (3 or 5 machines). In order to have an high throughput, data is randomly produced by a scala script and sent, in the internal format, through an Apache Kafka producer. The code below reports an example of a tuple that is randomly generated.

```

1 {
2     "sensor_name": "s1",
3     "start_date": "2016-03-01T00:00:00.000",
4     "end_date": "2018-09-01T00:00:00.000",
5     "data1": {
6         "location": {
7             "latitude": 45.00,
8             "longitude": 12.00,
9             "name": "z1"
10        },
11        "timestamp": 1521204365,
12        "temperatueVal": 24.5
13    },
14    "stt": {
15        "spatial": {
16            "unit": "point",
17            "metadata": ""
18        },
19        "temporal": {
20            "unit": "Minutes",
21            "count": 10,
22            "metadata": ""
23        },
24        "thematic": {
25            "name": "temperature",
26            "metadata": ""
27        }
28    }

```

CODE 8.1: Node representation

In the first experiment we have considered the Data Acquisition Plan reported in Figure 8.3 that is composed by a different combination of blocking (union) and non-blocking (filter, transform and virtual property) services. These services are applied on data made available through the Kafka context broker from two different sensors. Each sensor ingests 1 million tuples that are organized according to our internal data model (i.e. no transformation of the format is required).

The second experiment is a sequence of 10 cascaded aggregate services as represented in Figure 8.4.

Both experiments have been conducted on a single machine and on a cluster with 3 machine (1 master and 2 workers) and in a cluster with 5 machines (1 master and 4

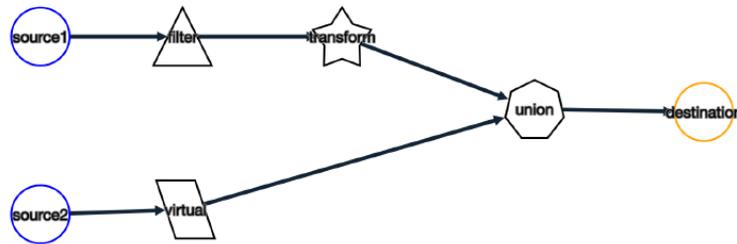


FIGURE 8.3: Composition of blocking and non-blocking services

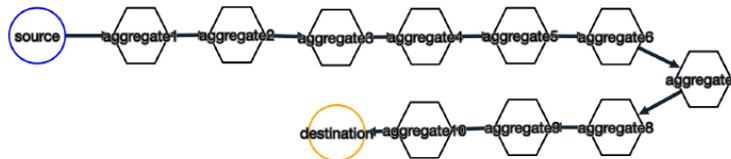


FIGURE 8.4: Composition of 10 aggregate services in cascade

workers). In this way we were able to compare the performances of the scripts with clusters of different sizes.

8.2.1 Results

As shown in Figure 8.5 and Figure 8.6 the performances have definitely improved, in both experiments, from the local execution to the cluster execution. The Processing Time in the first experiment passed from 5,3 seconds locally to 2,3 seconds in the 3 node cluster and 2,5 seconds in the 5 node cluster, while the seconds experiment passed from 2,3 second in the local mode to 1,41 seconds and 1,48 seconds respectively for the cluster of 3 machines and the cluster of 5 machines.

The Scheduling Delay (SD) has significantly improved from the local execution to the cluster execution. Locally we have a Scheduling Delay of 10 seconds for the first experiment and 5,3 seconds for the second experiment. In the cluster of 3 machines we have 71 ms delay for the first experiment and 4 ms for the second one. By contrast, the cluster with 5 machines introduces in both experiments a Scheduling Delay of 4 ms.

The Total Delay follows the same trend of the previous described metrics. The local execution produces the worst results in both experiments (152 seconds and 253 seconds) while the execution in the cluster improves the results with 101 seconds and 156 seconds for the experiments conducted on the cluster with 3 machines and 99 seconds and 152 seconds on the cluster with 5 machines.

The observation that we can point out from these experiments is that the execution of the script in a cluster of machines is always better than the local execution even if the improvement from three to five machines, in this case, is limited. As highlighted

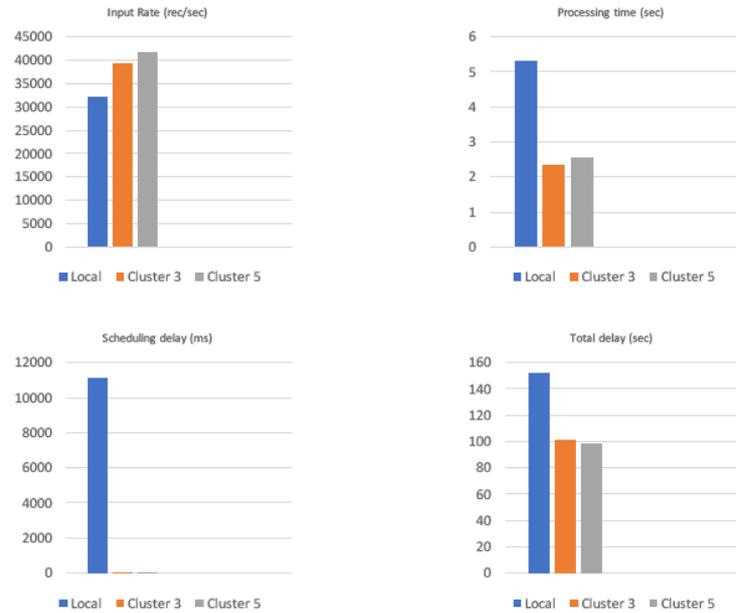


FIGURE 8.5: Input rate, Processing time, Scheduling delay and Total Delay of the blocking and non-blocking experiment

from the experiments, the Scheduling Delay has been deeply effected from the execution on the cluster, whereas, in some case, the Processing Time has been slightly worsened. We suppose that this is mainly due to the limited size of the batch that did not influenced the Processing Time.

8.3 Further Data Acquisition Plan Execution Experiments

In these experiments we modified the number of machines that execute the script and the number of Apache Kafka producer that publish data. The experiments have been conducted locally and on clusters of 5 and 10 machines. These experiments are mainly focused on the evaluation of the horizontal scalability of Apache Spark Streaming and to determine the number of events that can be handled per second. The data is randomly generated as in the previous experiments.

The DAPs that we used for these experiments contain both non-blocking and blocking services at increasing complexity. The first is a simple DAP that reads data from a sensor and stores the obtained value in a file. The second one applies a filter condition and only the events that meet the condition are stored in a file. The third one applies an aggregation on the considered events and the aggregated events are stored in a file. The last one compute a simplified version of the Humidex Factor (see Figure 8.7) described in the motivating scenario. A first set of sensors of type H_1 gathers humidity, expressed in hPa , every 10 minutes. The sensor observations are associated with the sensor identifier but no geo-spatial locations of the sensors are provided (but we have a database that associates each sensor with its longitude

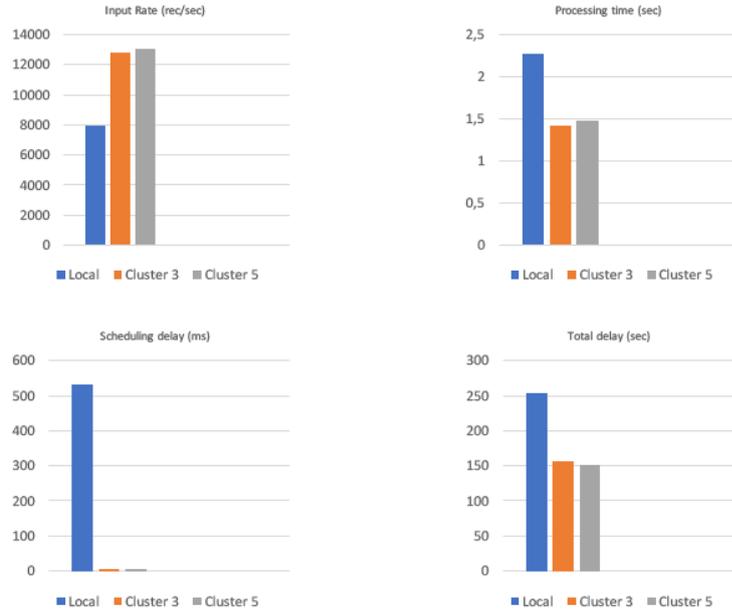


FIGURE 8.6: Input rate, Processing time, Scheduling delay and Total Delay of the 10 aggregation experiment

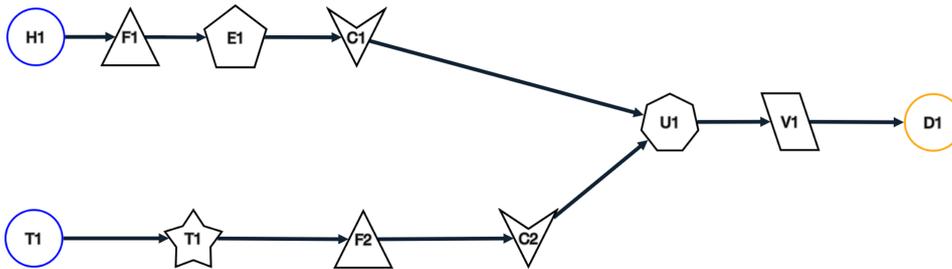


FIGURE 8.7: Humidex Factor calculation DAP

and latitude). A second set of sensors of type T_1 gathers the temperature every 15 minutes in Fahrenheit degree for each zone of the city. These sensors provide the needed information for computing the Humidex, but the sensor data need to be transformed, converted in coarser spatio-temporal granularities, integrated and enhanced in order to correctly compute the formula.

For the first four DAPs we have considered a single flow of 30 million events, whereas for the last one we have considered two flows of 30 millions events each.

8.3.1 Results

Figure 8.8 reports for each row the processing time and the size of the batch for different DAPs executed on a single node, a cluster of 5 nodes, and a cluster of 10 nodes.

As we can note from all the considered DAPs the processing time is deeply affected by the number of machines in which the DAP is executed. In the passage from one node to 5 nodes, the average processing times is reduced of at least one third. The reduction is less in the passage from the cluster of 5 to the one of 10 nodes, but we believe that this is due to the reduced number of processed events. Moreover, by increasing the complexity of the DAP (from the one in the first row to the last row) the processing time increases but not linearly with respect to the number of services that are included in the DAP. This is an important factor for dealing with DAPs of different complexity.

For what concern the computation of the Humidex factor, we can note that Spark is able to process 222.000 tuple per second. Moreover, the processing time improves deeply by considering clusters with 5 and 10 nodes. The local execution of a 10 seconds batch (with a range of events between 1.8 millions and 2.2 millions) requires an average time of 121 seconds, whereas in a cluster of 5 nodes requires 37 seconds, and 10 seconds in the cluster of 10 machines (thus an improvement of 12 times w.r.t. the local execution). Note that the scheduling delay is null for the cluster of 10 nodes (the average processing time – 3 ms. – is equal to the batch size), whereas this time is around 13 s. for the cluster of 5 nodes and 114 s. for the single node.

8.4 Semantic Virtualization Experiments

The data in the previous experiments were directly generated in the internal data model. For this reason no transformation need to be performed. In this experimental activity we performed a set of different experiments in order to understand how the system behave with the introduction of the transformation phase. The test executed can be defined within two main approaches:

- *Producer-Side (PS)*. Relying on this approach, data transformation in the internal format is realized before the transmission to the Apache Spark cluster (Figure 8.9(a)).
- *Consumer-Side (CS)*. Data are transmitted by the Kafka Client in the same format as they are generated. Data transformation in the internal format is performed directly on the Apache Spark cluster (Figure 8.9(b)).

Data, for this kind of experiments, is not randomly generated as in the previous case. We need to have information that is not already in the internal format, but is acquired, in csv format, from the (ARPA) (the Lombardy Regional Meteorological Agency)². Specifically, we get temperature and humidity data from some sensors located inside the city of Milan. Below an example of some temperature measurement.

²<http://www.arpalombardia.it/siti/arpalombardia/meteo/richiesta-dati-misurati/Pagine/RichiestaDatiMisurati.aspx>

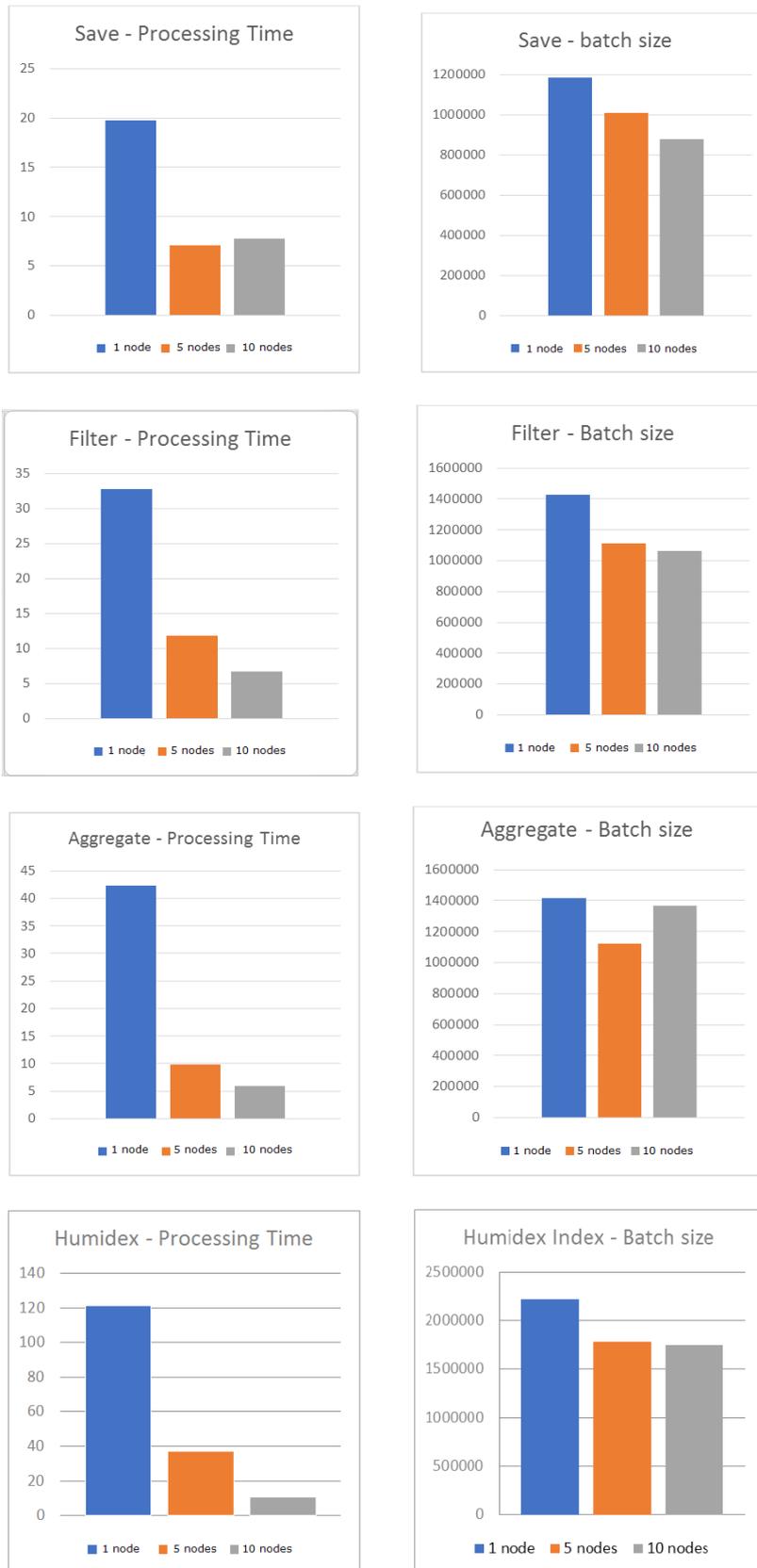


FIGURE 8.8: Processing time and batch size of different DAPs

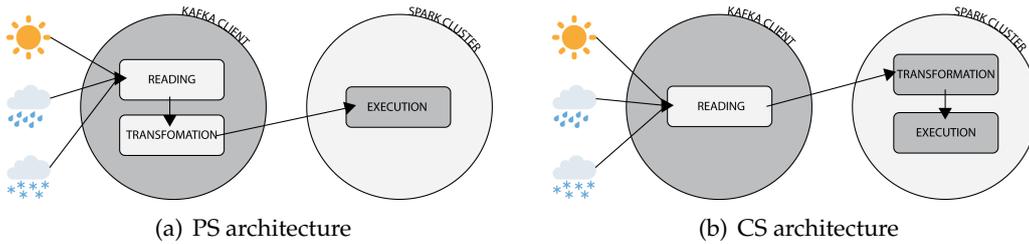


FIGURE 8.9: Architectures for the application of the transformation rules

```

1 45.466667, 9.216667, 2015/01/02 06:40, 0.6
2 45.466667, 9.216667, 2015/01/02 06:50, 0.6
3 45.466667, 9.216667, 2015/01/02 07:00, 0.5
4 45.466667, 9.216667, 2015/01/02 07:10, 0.4
5 45.466667, 9.216667, 2015/01/02 07:20, 0.5

```

CODE 8.2: Node representation

The first two attributes are the latitude and longitude of the sensor, the third is date and time of the measurement while the last attribute corresponds to the average temperature measured in Celsius degree in the last 10 minutes. Even if we considered the data produced in a long period of time, they are quite small. For this reason, the Kafka producer reads cyclically the same csv files.

8.4.1 Type of Experiments

The experiment for comparing the execution times on the transformation Producer-Side and Consumer-Side, have been conducted on a single machine and on a cluster of 5 machines, as shown in Figure 8.1. We here report the most representatives depending on the data acquisition services adopted (blocking and non blocking). Specifically, we detail the following three types of experiments:

- *Print*: 10 million records are ingested through the Kafka client in the system according to a specific topic. The Spark script receives the data and every 3 seconds prints the result on the console log (Figure 8.10 (a)). This is the easiest operation in which no elaboration of the data is required.
- *Aggregate*: the Kafka client ingests 10 million records to one topic. The Spark script receives the data and aggregates the records on a specific attribute every 6 seconds. The aggregation function is applied on the other attributes (Figure 8.10 (b)). This is an example of blocking service that requires to collect a certain number of tuples, from a single source, before the application of the transformation.
- *Join*: the number of ingested data through the Kafka client is 20 million records through two different topics (10 million records for each topic). The Spark

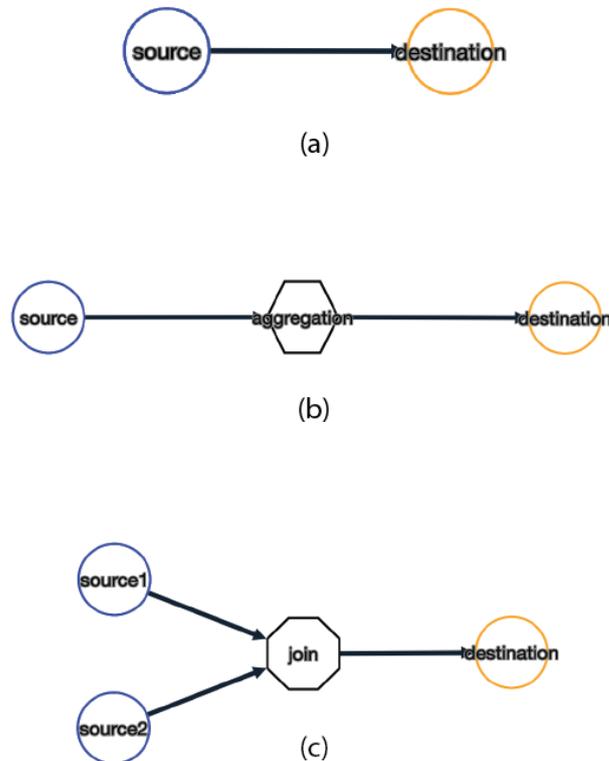


FIGURE 8.10: Type of experiments

script performs a Streaming Join on the records every 6 seconds (Figure 8.10 (c)). This is the most complicated experiment because tuples from different sources need to be collected and different transformation should be applied before performing the Join according to the specific time window.

In the presentation of results we use the following notation that is a combination of the approach used (Producer-Side or Consumer-Side) and the type of experiment (Print, Aggregate, or Join):

- *PSP* is the Producer-Side Print. Data is transformed in the internal format on the Kafka Producer-Side and the Apache Spark cluster only needs to print the incoming "raw" data, without elaborations.
- *CSP* is the Consumer-Side Print. Differently from the *PSP*, data is transmitted as arrive, without transformations. The Apache Spark cluster needs to gather the data transform them into the internal data format and then print the result.
- *PSA* is the Producer-Side Aggregate. The information produced by the sensors is transformed in the internal format before being transmitted to the Apache

Spark cluster. Data is then aggregate every 6 seconds.

- *CSA* is the Consumer-Side Aggregate. Raw data from sensor on the Apache Spark cluster is first of all transformed from the original format into the internal data format. Then, every 6 seconds, it is aggregated.
- *PSJ* is the Producer-Side Join. Data from two sensors is acquired and transformed by the Kafka Producer and is then joined on the Apache Spark cluster.
- *CSJ* is the Consumer-Side Join. This experiment performs a join of data of two sources but the information they produce must be transformed on the internal format before being elaborated.

In our tests the Apache Kafka Server runs in one of the worker of our cluster of machines. In the Print and Aggregate experiments the Kafka clients runs on the same machine of the Kafka Server while in the Join experiments, we need to have two different Kafka Producer that generate data on two different topics. For this reason one Kafka Producer runs on the same machine of the Apache Server and the second one runs on one of the cluster worker.

8.4.2 Results

These experiments aim at discovering the cost of performing the transformation before or after the transmission of the data. By means of the experimental activity we can point out some interesting observations. Before discussing the results on Processing Time, Scheduling Delay and Total Delay of each test we need to give a look to the diagram on Figure 8.11. This diagram reports the number of tuples that are transmitted every second in the Producer-Side architecture and in the Consumer-Side architecture by Apache Kafka. From the diagram is quite evident that in our setting the application of transformations "producer" side has negative effectiveness on the number of records produced every second. The average rate in the PS approach is around 24 thousand tuples while in the CS we have an average size of 55 thousand records. This means that the Consumer-Side approach generates more than the double of tuples per second. This is probably caused by the introduction of computation in order to transform the raw data into the internal data model.

For what concern the Join the average number of data per second is exactly the double in both approaches: 48 thousand tuples per second for the PS approach and 110 thousand tuples per second in the CS approach.

Processing Time. Figures 8.12(a) and 8.12(b) show the performance of the Producer-Side and Consumer-Side approach for the Print example. The interesting result is that the cluster execution has, in both approaches, worse performance than the local execution. Locally, the Processing Time, lasts 32 ms in the PS approach and 47 ms

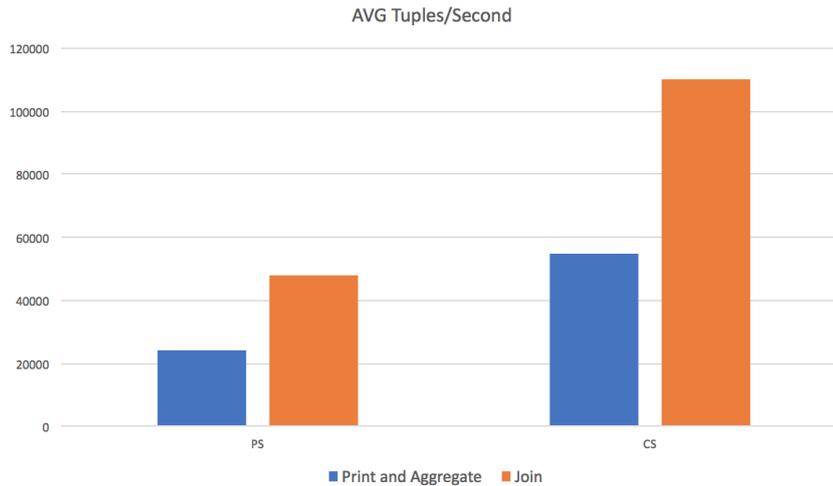


FIGURE 8.11: Average generated tuples per second for the three experiments

in the CS, while in the cluster execution the Processing Times are 46 ms for PS and 63 ms for CS. Spark script executes a simple operation. The time it takes to route the execution of the operation through the machines of the cluster worsens the performances. In this case we can see that in the CS execution the Processing Time are higher than the PS, but if we take in account the batch size presented in Figure 8.11 we can see that the amount of data to process is more than the double.

In the Aggregate and Join operations (blocking operators) we have better Processing Times. If, in the Aggregate tests (Figure 8.12(c) and Figure 8.12(d)) we have an improvement of more than a second (from 2,8 seconds to 1,5 seconds in the PS approach and from 6,6 seconds to 4,6 seconds in the CS approach), in the Join tests (Figure 8.12(e) and Figure 8.12(f)) we have a remarkable improvement (from 16,5 seconds to 5,70 seconds in the PS approach and from 20,1 to 7,9 seconds in the CS approach).

Scheduling Delay. The second feature to analyze is the Scheduling Delay. The tests provided us with interesting results, especially in the comparison of Producer-Side and Consume-Side.

In the Print experiments (Figures 8.13(a) and 8.13(b)) both, PS and CS approaches and both, local and cluster execution there are no remarkable differences. The average Scheduling Delay is around 1,5 ms for the PS approach and 1,8 ms for the CS approach. The main differences are shown in the other two tests. In the Aggregate case, Scheduling Delay in the PS approach is around 405 ms locally and 1,5 ms in the cluster while in the CS approach the local Scheduling Delay is more than 2 minutes and the local is close to 1 minute. If we take a look at the graph represented in Figures 8.13(c) and 8.13(d) we can see how the trend of the Scheduling Delay in the PS approach is similar to a constant in the cluster case and in the local case is more



FIGURE 8.12: Processing Times of the considered Data Acquisition Plans

floating, while in the CS approach the graph tends to grow constantly, especially in the local case.

The interesting point is given with the execution of the Join experiments. As shown on Figures 8.13(e) and 8.13(f) the Scheduling Delay of the local execution in the PS is smaller than the cluster execution in the CS approach (the highest value in our experiments are 390 seconds for the PS local approach and 580 seconds for the CS cluster approach). Another feature to notice is that in the cluster execution of the PS approach the Scheduling Delay is almost constant while in all the other cases the Scheduling Delay is represented as an increasing function.

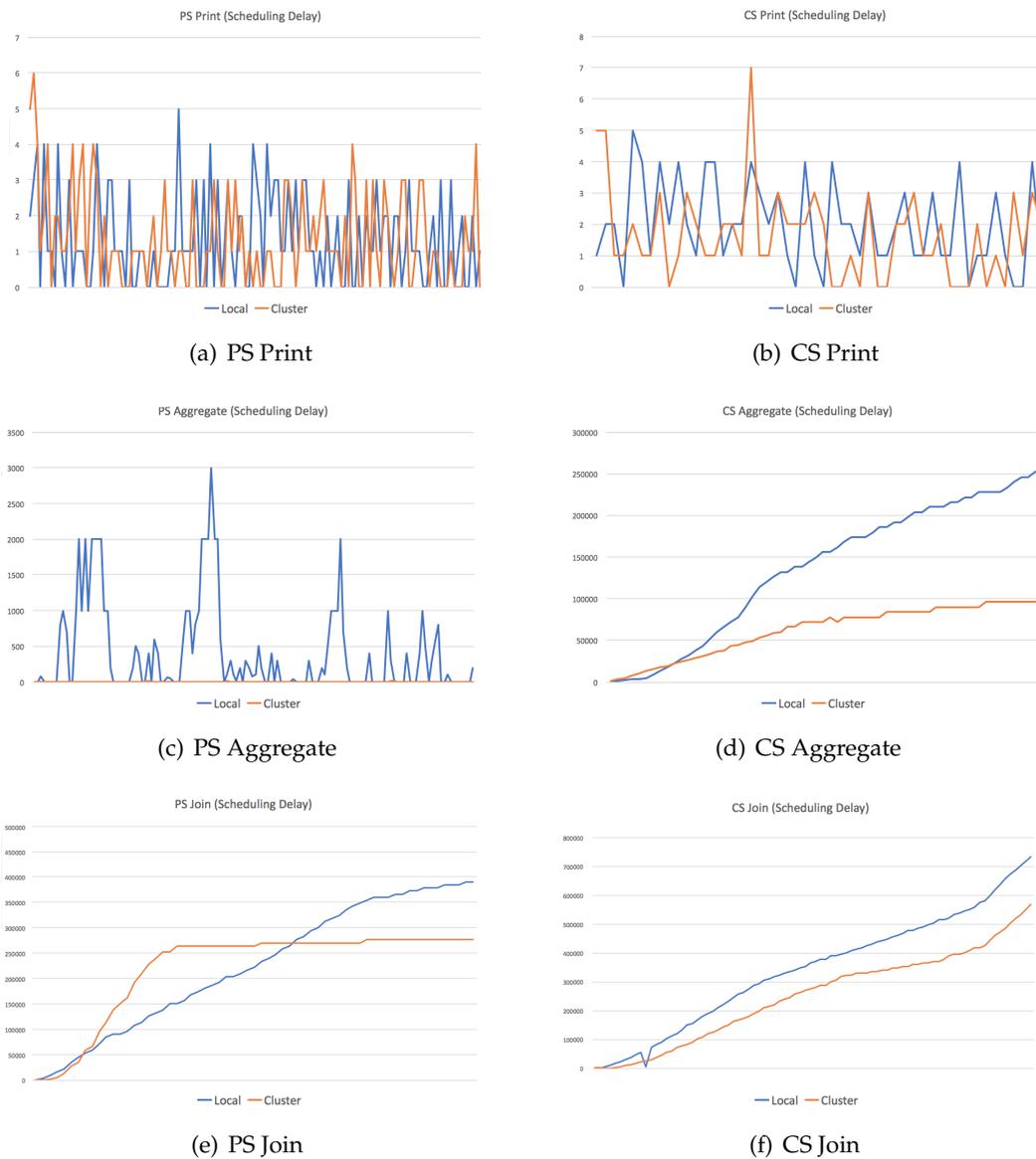


FIGURE 8.13: Scheduling Delay of the considered Data Acquisition Plans

Total Delay. Figure 8.14 provides details about the Total Delay for every experiment we conducted. As we can see the Print operation gives the best results in the Consumer-Side approach by taking more the half of the time of the Producer-Side approach either in the local and cluster mode. For what concern the Aggregation Total Delay is quite similar to every experiment we have conducted. The local execution takes the same time in both approaches (7 minutes and 30 seconds) while in the cluster execution there is an improvement of more than 1 minutes from the Producer-Side approach an the Consumer-Side approach.

The Join operator is the one that introduces the main differences between the two approaches. If we consider the difference of the local and cluster execution, in both

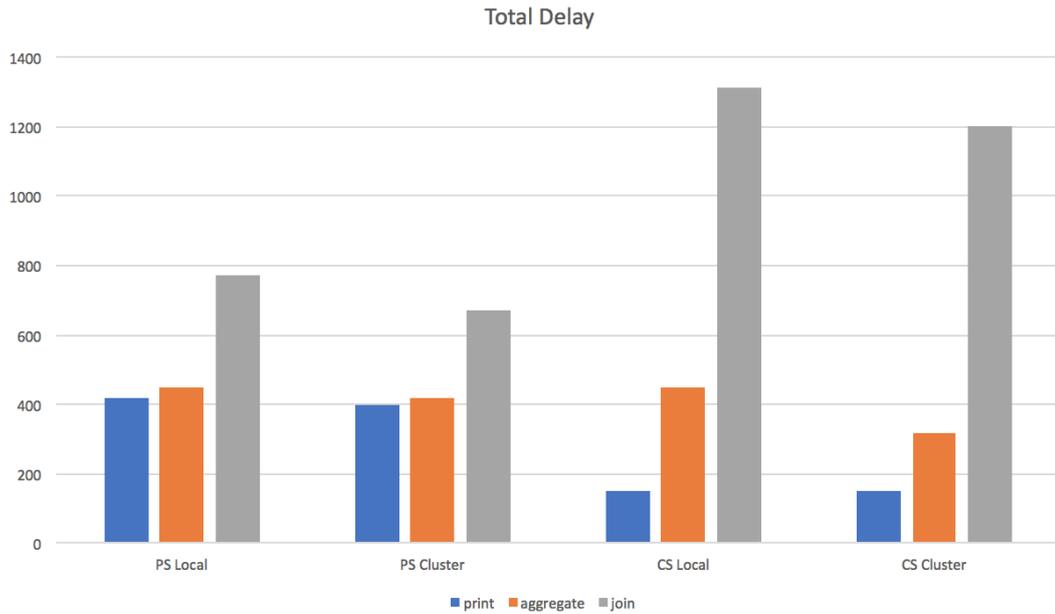


FIGURE 8.14: Total Delay for every experiment and for the different approaches

approaches the cluster execution gives the better results. For what concern the difference between the PS and CS approach we can see that also the execution of the Join locally is sensibly better than the execution in Cluster mode of the CS approach (770 seconds compared to 1200 seconds). The best performances are the ones provided by the execution in a Cluster with the PS approach. It takes 670 seconds to join 20 million tuples.

8.5 Concluding Remarks

The introduction of the transformation phase has definitely worsened the performances compared to the first experiments. However, in a real situation and as described in the whole thesis, the format of the data is heterogeneous and transformation from one format to another is often required. For what concern the two approaches we can highlight some interesting observations. Surely in the Producer-Side architecture the tuples are processed by Apache Spark not exactly in the same moment that they are produced. The elaboration performed with the transformation introduced a delay in the transmission of tuples. On the other hand the Consumer-Side approach has the advantage that the information are sent as they arrive to the Kafka Server but the larger size of the batch introduces a higher Scheduling Delay. This leads to a longer time to perform the same amount of informations, especially for the high computational services as Aggregation or Join and, if too many batches will be queued the system will come to a grinding halt eventually.

Conclusions and Future Research Directions

In this thesis we presented StreamLoader, a system developed with the aim of handling the heterogeneity of data produced by sensors in the field of Internet of Things. The thesis has been focused, in the first part, on the presentation of the issue of the Semantic Virtualization of sensors belonging to different cross-domain IoT platforms. We identified and introduced a flexible multi-granular Spatio-Temporal-Thematic data model according to which the schema of heterogeneous sensors belonging to cross-domain IoT platforms can be easily transformed by using standard wrapping tools. Then, a Semantic Virtualization process has been introduced according to which both the sensors and the schema of the events generated by the sensors are semantically characterized by means of ontology instances. This has been conceived by introducing a Domain Ontology, that is the conceptualization of a domain of interest, that extends concepts and ideas of some of the most important IoT ontologies already available. The process of Semantic Virtualization is carried out by taking into account the Spatio-Temporal-Thematic dimensions according to which the events generated by the sensors are observed and has the purpose to move towards the adoption of a common semantics of the sensor event streams.

In our system the description can be partial in order to deal with situations in which sensors are not equipped with the facilities or the properties for associating the Spatio-Temporal-Thematic dimensions specified in the model. Conditions are specified for guaranteeing the consistency of a sensor relying on the consistency of the Spatio-Temporal-Thematic dimensions of the produced events. It allows us also to enrich the description of the sensors with static information or with metadata taken from the concept of the Domain Ontology or external data sources.

The second part addressed the sound and consistent generation of Data Acquisition Plans by filtering, joining, aggregating and transforming events produced by the sensors and their execution on a real environment. A set of services has been proposed and described. The result of application of each service produces a new stream whose consistency can be evaluated w.r.t. the Domain Ontology for its semantic characterization. The Data Acquisition Plans are then automatically translated in a language of new generation for processing big data streams, specifically

Apache Spark Streaming. Then the execution is performed in a distributed environment in order to easily scale to the number of sensors and events generated by the sensors.

The proposed solution allow us to create "soft bridges" among the sensor belonging to cross-domain IoT platforms for the definition of Data Acquisition Plans required by the users. This means that if a user wishes to develop another analysis in the context of another Domain Ontology a new "soft bridge" can be easily specified. The whole system is supported by a visual environment that provides interfaces that support domain experts during the Semantic Virtualization phase and in the design of the DAP interfaces that allows them to specify services or configuration parameters on different levels without the use of specific programming languages.

The results obtained through this thesis can be the starting point for new research challenges. As future work we aim at investigating how to include different kinds of constraints into the Domain Ontology. We wish to include restrictions on the mandatory occurrence of properties and on the verification of logical formulas on the ontological instances. The introduction of these constraints will give the chance to introduce a new definition of consistency and to produce Data Acquisition Plan that are more meaningful.

Another research direction is the investigation of machine learning algorithms for supporting the user in the semi-automatic semantic labeling of the attributes belonging to the schema of new discovered sensors. This is a relevant feature to be considered when dealing with new sensors belonging to an external IoT Platform. The approach could also suggest possible annotation with a level of confidence of the quality of the suggestion. Moreover, it would be nice to exploit the semantic annotations of sensors for the identification of components of the Domain Ontology that need to be evolved in order to better describe a certain domain context.

Regarding the execution of the Data Acquisition Plan, we wish to introduce the possibility to translate the DAP in other Stream Processing Frameworks. From the plethora of systems available the most interesting and promising is surely Apache Flink. The aim of this work is to compare the different framework both from the point of view of expressiveness of the different platforms and from the point of view of scalability and efficiency of the generated scripts. Moreover, the current organization of the services in our graphical environment easily allows the introduction of new services that work on the event stream data model. For this reason, it would be nice to extend the set of services made available by StreamLoader with machine learning algorithms and user-defined applications that allow to execute sophisticated manipulation of sensor data streams and compare their execution in different stream processing frameworks.

At the current stage, once the script is generated and executed in the cluster of machines, it is not possible to monitor its execution. Therefore, an interesting research

direction is related to the extension of the system for monitoring the execution of the single services and of the entire DAP. This extension requires to modify the code of the generated scripts for keeping track of the processed events and to identify visualization approaches of the generated statistics.

Bibliography

- [1] D. Abadi et al. "The Aurora and Borealis Stream Processing Engines". In: *Data Stream Management* (2016), pp. 337–359.
- [2] K. Aberer, M. Hauswirth, and A. Salehi. *The Global Sensor Networks Middleware for Efficient and Flexible Deployment and Interconnection of Sensor Networks*. Tech. rep. Distributed Information Systems Laboratory LSIR, 2006.
- [3] M. B. Alaya et al. "Toward Semantic Interoperability in OneM2M Architecture". In: *IEEE Communications Magazine* 53.12 (2015), pp. 35–41.
- [4] C. Ambrosio and S. Widergren. "A Framework for Addressing Interoperability Issues". In: *IEEE Power Engineering Society General Meeting, PES '07*. 2007.
- [5] IoT Analytics. *List Of 640+ Enterprise IoT Projects*. URL: <https://iot-analytics.com/product/list-of-640-iot-projects/>.
- [6] Y. Arens, C. N. Hsu, and C. A. Knoblock. "Query Processing in the SIMS Information Mediator". In: *Readings in Agents* (1998), pp. 82–90.
- [7] G. Atemezeng et al. "Transforming Meteorological Data into Linked Data". In: *Semantic Web 4.3* (2013), pp. 285–290.
- [8] Z. B. Babovic, J. Protic, and V. Milutinovic. "Web Performance Evaluation for Internet of Things Applications". In: *IEEE Access* 4 (2016), pp. 6974–6992.
- [9] D. Bandyopadhyay and J. Sen. "Internet of Things: Applications and Challenges in Technology and Standardization". In: *Wireless Personal Communications* 58 (2011), pp. 49–69.
- [10] R. S. Barga et al. *Consistent Streaming Through Time: A Vision for Event Stream Processing*. Tech. rep. CERN, 2006.
- [11] P. Barnaghi et al. "Semantics for the Internet of Things: Early Progress and Back to the Future". In: *International Journal on Semantic Web and Information Systems* 8.1 (2012), pp. 1–21.
- [12] P. Bellini et al. "Km4City Ontology Building vs Data Harvesting and Cleaning for Smart-city Services". In: *Journal of Visual Languages and Computing* 25.6 (2014), pp. 827–839.
- [13] R. Bendadouche et al. "Extension of the Semantic Sensor Network Ontology for Wireless Sensor Networks: the Stimulus-Wsnnode-Communication Pattern". In: *5th International Workshop on Semantic Sensor Network, SSN12*. 2012.
- [14] S. Bendel et al. "A Service Infrastructure for the Internet of Things Based on XMPP". In: *IEEE International Conference on Pervasive Computing Community Workshop, PERCOM Workshop*. 2013.

- [15] M. Bermudez-Edo et al. "IoT-Lite: A Lightweight Semantic Model for the Internet of Things". In: *International IEEE Conferences on Ubiquitous Intelligence and Computing, Advanced and Trusted Computing, Scalable Computing and Communications, Cloud and Big Data Computing, Internet of People, and Smart World Congress (UIC/ATC/ScalCom/CBDCom/IoP/SmartWorld)*. 2016.
- [16] E. Bertino, S. Nepal, and R. Ranjan. "Building Sensor-Based Big Data Cyberinfrastructures". In: *IEEE Cloud Computing 2.5* (2015), pp. 64–69.
- [17] C. Bettini, S. Jajodia, and S. Wang. *Time Granularities in Databases, Data Mining, and Temporal Reasoning*. Springer, 2000.
- [18] O. Bibani et al. "A Demo of IoT Healthcare Application Provisioning in Hybrid Cloud/Fog Environment". In: *IEEE International Conference on Cloud Computing Technology and Science, CloudCom*. 2016.
- [19] M. Blackstock and R. Lea. "IoT Interoperability: A Hub-Based Approach". In: *2014 International Conference on the Internet of Things (IOT)*. 2014.
- [20] F. Bonomi, J. Milito R. Zhu, and S. Addepalli. "Fog Computing and its Role in the Internet of Things". In: *Federated Conference on Computer Science and Information Systems, Fed-CSIS '14*. 2014.
- [21] S. Borgo and C. Masolo. "Ontological Foundation of Dolce". In: Springer, 2010, pp. 279–295.
- [22] M. Botts et al. "Ogc® Sensor Web Enablement: Overview and High Level Architecture". In: *International Conference on GeoSensor Networks*. 2006.
- [23] A. Brito et al. "Scalable and Low-Latency Data Processing with Stream MapReduce". In: *2011 IEEE 3rd International Conference on Cloud Computing Technology and Science, CloudCom*. 2011.
- [24] A. Bröring et al. "Enabling IoT Ecosystems through Platform Interoperability". In: *IEEE Software* (2017).
- [25] H. Cai et al. "Service-Oriented Architecture". In: *Services Computing* (2007), pp. 89–113.
- [26] E. Camossi, E. Bertino, and M. Bertolotto. "Multi-Granular Spatio-Temporal Object Models: Concepts and Research Directions". In: *Second international conference on Object databases, ICOODB'09*. 2009.
- [27] E. Camossi et al. "Handling Expiration of Multigranular Temporal Objects". In: *Journal of Logic and Computation* 14.1 (2004), pp. 23–50.
- [28] S. Castano et al. "Ontology and Instance Matching". In: *Knowledge-Driven Multimedia Information Extraction and Ontology Evolution 6050* (2011), pp. 167–195.
- [29] R. Cattell. "Scalable SQL and NoSQL Data Stores". In: *ACM SIGMOD Record* 39.4 (2010), pp. 12–27.
- [30] M. A. Chaqfeh and N. Mohamed. "Challenges in Middleware Solutions for the Internet of Things". In: *3th International Conference on Collaboration Technologies and Systems, CTS'12*. 2012.

- [31] M. Compton et al. "The SSN Ontology of the W3C Semantic Sensor Network-Incubator Group". In: *Web Semantics: Science, Services and Agents on the World-Wide Web 17* (2012), pp. 25–32.
- [32] F. Corno, L. De Russis, and A. M. Roffarello. *A Semantic Web Approach to Simplifying TriggerAction Programming in the IoT*. URL: <https://ifttt.com/channels>.
- [33] M.F. Costabile et al. "Visual Interactive Systems for End-User Development: A Model-Based Design Methodology". In: *Systems, Man and Cybernetics, Part A: Systems and Humans, IEEE Transactions on 37.6* (2007), pp. 1029–1046.
- [34] P. De Bièvre. "The 2012 International Vocabulary of Metrology: "VIM"". In: *Accreditation and Quality Assurance 17.2* (2012), pp. 231–232.
- [35] J. Dean and S. Ghemawat. "MapReduce: Simplified Data Processing on Large Clusters". In: *6th Symposium on Operating System Design and Implementation, OSDI'04*. 2004.
- [36] G. Dimitrakopoulos. "Intelligent Transportation Systems Based on Internet-Connected Vehicles: Fundamental Research Areas and Challenges". In: *11th International Conference on ITS Telecommunications, ITST '11*. 2011.
- [37] M. Doyle. *Who's Going to Service All Those "Things" in the IoT?* URL: <https://www.ptc.com/en/product-lifecycle-report/whos-going-to-service-all-those-things-in-the-iot>.
- [38] L. van den Drink, J. Tandy, and P. Barnaghi. *Spatial Data on the Web Best Practices*. W3C Note. W3C, 2017.
- [39] J. Ellingwood. *An Introduction to Big Data Concepts and Terminology*. URL: <https://www.digitalocean.com/community/tutorials/an-introduction-to-big-data-concepts-and-terminology>.
- [40] ETSI. *Machine-to-Machine Communications (m2m); Mia, Dia and Midinterfaces*. URL: http://www.etsi.org/deliver/etsi_ts/102900_102999/102921/02.01.01_60/ts_102921v020101p.pdf.
- [41] J. Euzenat et al. "Ontology Alignment Evaluation Initiative: Six Years of Experience". In: *Journal of Data Semantics 15* (2011), pp. 158–192.
- [42] H. Farhangi. "The Path of the Smart Grid". In: *IEEE Power and Energy Magazine 8.1* (2010), pp. 18–28.
- [43] P. Feng and J. R. Hobbs. "Temporal Aggregates in OWL-Time". In: *AAAI Fall Symposium on Agents and the Semantic Web*. 2005.
- [44] P. Ferreira, R. Martino, and D. Domingos. "Iot-Aware Business Processes for Logistics: Limitations of Current Approaches". In: *Inforum Conference*. 2010.
- [45] I. Fette. *The WebSocket Protocol*. Tech. rep. Internet Engineering Task Force (IETF), 2011.
- [46] G. Fischer. "Social Creativity, Symmetry of Ignorance and Meta-design". In: *Knowledge-Based Systems Journal 13.7-8* (2000), pp. 527–537.
- [47] G. Fischer et al. "Meta-design: A Manifesto for End-user Development". In: *Commun. ACM 47.9* (2004), pp. 33–37.

- [48] G. Fortino et al. "BodyCloud: A SaaS Approach for Community Body Sensor Networks". In: *Future Generation Computing Systems* (2014), 35:62–79.
- [49] K. Frank et al. "Sensor-Based Identification of Human Stress Levels". In: *IEEE International Conference on Pervasive Computing and Communications Workshops, PerCom Workshops '13*. 2013.
- [50] M. Franz et al. "Cytoscape.js: a graph theory library for visualisation and analysis". In: *Bioinformatics* 32.2 (2016), pp. 309–311.
- [51] M. Friedewald and O. Raabe. "Ubiquitous Computing: an Overview of Technology Impacts". In: *Telematics and Informatics* 28.2 (2011), pp. 55–65.
- [52] E. Friedman and K. Tsoumas. *Introduction to Apache Flink: Stream Processing for Real Time and Beyond*. O'Reilly, 2016.
- [53] M. Ganzha et al. "Semantic Interoperability in the Internet of Things: An Overview from the INTER-IoT Perspective". In: *Journal of Network and Computer Applications* (2016).
- [54] M. Ganzha et al. "Semantic Technologies for the IoT - an INTER-IoT Perspective". In: *IEEE First International Conference on Internet-of-Things Design and Implementation, IoTDI*. 2016.
- [55] B. Gedik et al. "SPADE: the System S Declarative Stream Processing Engine". In: *2008 ACM SIGMOD International Conference on Management of data, SIGMOD'08*. 2008.
- [56] C. H. Goh. "Representing and Reasoning about Semantic Conflicts in Heterogeneous Information Sources." PhD thesis. MIT, 1997.
- [57] A. J. G. Gray et al. "A Semantically Enabled Service Architecture for Mashups over Streaming and Stored Data". In: *The Semantic Web: Research and Applications. ESWC 2011. Lecture Notes in Computer Science 6644* (2011).
- [58] T. Gruber. "A Translation Approach to Portable Ontology Specifications". In: *Knowledge Acquisition* 5.2 (1993), pp. 199–220.
- [59] T. Gruber. "Toward Principles for the Design of Ontologies Used for Knowledge Sharing". In: *International Journal of Human-Computer Studies* 43.5-6 (1995), pp. 907–928.
- [60] J. Gubbi et al. "Internet of Things (IoT): A Vision, Architectural Elements, and Future Directions". In: *Future Generation Computing Systems* 29.7 (2013), pp. 1645–1660.
- [61] B. Guo et al. "Opportunistic IoT: Exploring the Harmonious Interaction Between Human and the Internet of Things". In: *Journal of Network and Computer Applications* 36.6 (2013), pp. 1531–1539.
- [62] C. A. Gutwin, M. Lippold, and T. C. Graham. "Real-time Groupware in the Browser: Testing the Performance of Web-Based Networking". In: *ACM 2011 conference on Computer supported cooperative work, CSCW'11*. 2011.
- [63] A. Haller et al. "The SOSA/SSN Ontology: A Joint W3C and OGC Standard Specifying the Semantics of Sensors, Observations, Actuation, and Sampling". In: *Semantic Web Journal* (2018).

- [64] S. L. Hamilton et al. "Interoperability - a Key Element for the Grid and DER of the Future". In: *IEEE Power Engineering Society Transmission and Distribution Conference*. 2006.
- [65] D. M. Han and J. H. Lim. "Design and Implementation of Smart Home Energy Management Systems Based on ZigBee". In: *IEEE Transactions on Consumer Electronics* 56.3 (2010), pp. 1417–1425.
- [66] G. Hauber-Davidson and E. Idris. "Smart Water Metering". In: *Water* 33.3 (2006), pp. 56–59.
- [67] Healthcare Information and Management Systems Society (HIMSS). *What is Interoperability?* 2013. URL: <http://www.himss.org/library/interoperability-standards/what-is?navItemNumber=17333>.
- [68] Big IoT. *Deliverable 2.1 Analysis of Technology Readiness – and Annex*. URL: <http://big-iot.eu/media/deliverables/>.
- [69] S. Jabbar et al. "A Rest-Based Industrial Web of Things' Framework for Smart Warehousing". In: *The Journal of Supercomputing* (2016), pp. 1–15.
- [70] S. Jabbar et al. "Semantic Interoperability in Heterogeneous IoT Infrastructure for Healthcare". In: *Wireless Communications and Mobile Computing* (2017).
- [71] S. Karnouskos. "The Cooperative Internet of Things Enabled Smart Grid". In: *14th IEEE International Symposium on Consumer Electronics, ISCE '10*. 2010.
- [72] W. Keith Edwards and Grinter R. E. "At Home with Ubiquitous Computing: Seven Challenges". In: *International Conference on Ubiquitous Computing, Ubicomp 2001*. 2001.
- [73] M. Khan et al. "Context-Aware Low Power Intelligent Smart Home Based on the Internet of Things". In: *Computers & Electrical Engineering* 52 (2016), pp. 208–222.
- [74] J. Kinley. *The Lambda Architecture: Principles for Architecting Realtime Big Data Systems*. URL: <http://jameskinley.tumblr.com/post/37398560534/the-lambda-architecture-principles-for>.
- [75] M. Kirsche and R. Klauck. "Unify to Bridge Gaps: Bringing XMPP into the Internet of Things". In: *IEEE International Conference on Pervasive Computing Community Workshop, PERCOM Workshop*. 2012.
- [76] C. Knoblock et al. "Semi-Automatically Mapping Structured Sources into the Semantic Web". In: *9th Extended Semantic Web Conference*. 2012.
- [77] S. Kolozali et al. "A Knowledge-based Approach for Real-Time IoT Data Stream Annotation and Processing". In: *IEEE International Conference on Internet of Things, iThings 2014, Green Computing and Communications, GreenCom 2014, and Cyber-Physical-Social Computing, CPSCoM 2014*. 2014.
- [78] K. Kotis, A. Katasonov, and J. Leino. "Aligning Smart and Control Entities in the IoT". In: *Internet of Things, Smart Spaces, and Next Generation Networking* 7469 (2012), pp. 39–50.

- [79] M. Kranz. *IoT Meets Standards, Driving Interoperability and Adoption*. URL: <https://blogs.cisco.com/digital/iot-meets-standards-driving-interoperability-and-adoption>.
- [80] J. Kreps. *Questioning the Lambda Architecture*. URL: <https://www.oreilly.com/ideas/questioning-the-lambda-architecture>.
- [81] J. Laskowski. *DAGScheduler - Stage-Oriented Scheduler*. URL: <https://jaceklaskowski.gitbooks.io/mastering-apache-spark/spark-dagscheduler.html>.
- [82] J. Laskowski. *RDD Lineage - Logical Execution Plan*. URL: <https://jaceklaskowski.gitbooks.io/mastering-apache-spark/spark-rdd-lineage.html>.
- [83] D. Le-Phuoc. *Semantic Sensor Network Ontology*. W3C Recommendation. W3C, 2017.
- [84] D. Le-Phuoc et al. "The Linked Sensor Middleware - Connecting the Real World and the Semantic Web". In: *Semantic Web Challenge*. 2011.
- [85] D. Le Phuoc et al. "A middleware framework for scalable management of linked streams". In: *Web Semantics: Science, Services and Agents on the World Wide Web* 16 (2012), pp. 42–51.
- [86] M. Lee and J. D. Cho. "Logmusic: Context-Based Social Music Recommendation Service on Mobile Device". In: *ACM International Joint Conference on Pervasive and Ubiquitous Computing, UbiComp '14*. 2014.
- [87] A. H. Levis and L. W. Wagenhals. "C4ISR Architectures: I. Developing a Process for C4ISR Architecture Design". In: *System Engineering* 3.4 (2000), pp. 225–247.
- [88] G. Liang, J. Cao, and W. Zhu. "CircleSense: a Pervasive Computing System for Recognizing Social Activities". In: *11th IEEE International Conference on Pervasive Computing and Communications, PerCom '13*. 2013.
- [89] R. A. Light. "Mosquitto: Server and Client Implementation of the MQTT Protocol". In: *The Journal of Open Source Software* 2.13 (2017).
- [90] J. Liu et al. "Applications of Internet of Things on Smart Grid in China". In: *13th International Conference on Advanced Communication Technology: Smart Service Innovation through Mobile Interactivity, ICACT '11*. 2011.
- [91] M. Lv et al. "Detecting Traffic Congestions Using Cell Phone Accelerometers". In: *International Joint Conference on Pervasive and Ubiquitous Computing, UbiComp '14*. 2014.
- [92] S. Manna and N. Bhunia S. S. and Mukherjee. "Vehicular Pollution Monitoring Using IoT". In: *International Conference on Recent Advances and Innovations in Engineering, ICRAIE 2014*. 2014.
- [93] J. Manyika. "The Internet of Things: Mapping the Value Beyond the Hype". In: *McKinsey Global Institute* (2015).
- [94] J. Masterton and R. Fa. *A Method of Quantifying Human Discomfort Due to Excessive Heat and Humidity*. Ministere de l'Environnement, 1979.
- [95] E. Mena et al. "Observer: An Approach for Query Processing in Global Information Systems Based on Interoperability Between Pre-Existing Ontologies".

- In: *1st IFCIS International Conference on Cooperative Information Systems, CoopIS '96*. 1996.
- [96] N. Mitra and Y. Lafon. *SOAP Version 1.2 Part 0: Primer (Second Edition)*. W3C Recommendation. W3C, 2007.
- [97] Y. Mizunuma et al. "Twitter Bursts: Analysis of their Occurrences and Classifications". In: *8th International Conference on Digital Society, ICDS 2014*. 2014.
- [98] J.-J. Moreau et al. *Web Services Description Language (WSDL) Version 2.0 Part 1: Core Language*. W3C Recommendation. W3C, 2007.
- [99] J. P. Morrison. *Flow-Based Programming, 2nd Edition: A New Approach to Application Development*. CreateSpace, 2010.
- [100] H. Mueller et al. "From RESTful to SPARQL: A Case Study on Generating Semantic Sensor Data". In: *6th International Conference on Semantic Sensor Network, SSN13*. Vol. 1063. 2013, pp. 51–66.
- [101] P. Nesi et al. "Assisted Knowledge Base Generation, Management and Competence Retrieval". In: *International Journal of Software Engineering and Knowledge Engineering* 22.8 (2012).
- [102] N. Noury et al. "Monitoring Behavior in Home Using a Smart Fall Sensor and Position Sensors". In: *1st Annual International IEEE-EMBS Special Topic Conference on Microtechnologies in Medicine and Biology, MMB '00*. 2000.
- [103] N. F. Noy. "Semantic Integration: a Survey of Ontology-Based Approaches". In: *ACM Sigmod Record* 33.4 (2004), pp. 65–70.
- [104] National Security Agency (NSA). *NSA Releases First in Series of Software Products to Open Source Community*. 2014. URL: <https://www.nsa.gov/news-features/press-room/press-releases/2014/nifi-announcement.shtml>.
- [105] Open Mobile Alliance (OMA). *Open Mobile Alliance (OMA) Specification*. URL: http://technical.openmobilealliance.org/Technical/release_program/docs/NGSI/V1_0-20120529-A/OMA-TS-NGSI_Context_Management-V1_0-20120529-A.pdf.
- [106] D. Pavithra and R. Balakrishnan. "IoT Based Monitoring and Control System for Home Automation". In: *Global Conference on Communication Technologies, GCCT*. 2015.
- [107] M. Petre and A. F. Blackwell. "Children as Unwitting End-User Programmers". In: *Proc. of VL/HCC 2007*. 2007, pp. 239–242.
- [108] R. W. Picard. *Affective Computing*. MIT Press, 1997.
- [109] R. Pike et al. "Interpreting the Data: Parallel Analysis with Sawzall". In: *Scientific Programming Journal* 13.4 (2003), pp. 227–298.
- [110] G. Pozzani. *Temporal, Spatial, and Spatio-temporal Granularities*. 2009. URL: <http://www.di.univr.it/documenti/OccorrenzaIns/matdid/matdid955973.pdf>.
- [111] J. Pradilla, C. Palau, and M. Esteve. "Lightweight Sensor Observation Service (SOS) for Internet of Things (IoT)". In: *Processing of ITU Kaleidoscope Conference* (2015).

- [112] E. Rahm and P. A. Bernstein. "A Survey of Approaches to Automatic Schema Matching". In: *The VLDB Journal* 10.4 (2001), pp. 334–350.
- [113] V. Rajaraman et al. "Enabling Plug-n-Play for the Internet of Things with Self Describing Devices". In: *International Conference on Information Processing in Sensor Networks*. 2015.
- [114] N. Rao et al. "Design of Architecture for Efficient Integration of Internet of Things and Cloud Computing". In: *International Journal of Advanced Research in Computer Science* 8.3 (2017), pp. 392–396.
- [115] J. Rapoza. *SPARQL Will Make the Web Shine*. URL: <http://www.eweek.com/development/sparql-will-make-the-web-shine>.
- [116] Y. Raymond et al. *The Timeline Ontology*. URL: <http://mtools.sourceforge.net/timeline/timeline.html>.
- [117] M. A. Razzaque et al. "Middleware for Internet of Things: a Survey". In: *IEEE Internet of Things Journal* 3.1 (2016), pp. 70–95.
- [118] R. J. Robles et al. "A Review on Security in Smart Home Development". In: *International Journal of Advanced Science and Technology* 15 (2010), pp. 13–22.
- [119] K. Rose, S. Eldridge, and L. Chapin. *The Internet of Things (IoT): An Overview Understanding the Issues and Challenges of a More Connected World*. Internet Society, 2015.
- [120] Y. K. Row and T. J. Nam. "CAMY: Applying a Pet Dog Analogy to Everyday Ubicomp Products". In: *ACM International Joint Conference on Pervasive and Ubiquitous Computing, UbiComp '14*. 2014.
- [121] J. Rowley. "The Wisdom Hierarchy: Representations of the DIWK Hierarchy". In: *Journal of Information Science* 33.2 (2007), pp. 163–180.
- [122] E. A. Rundensteiner et al. "CAPE: A Constraint-Aware Adaptive Stream Processing Engine". In: *Stream Data Management* (2005), pp. 83–111.
- [123] P. Saint-Andre. *Extensible Messaging and Presence Protocol (XMPP): Core*. URL: <http://www.rfc-editor.org/info/rfc6120>.
- [124] T. Scholz et al. "An Integration Method for the Specification of Rule-Oriented Mediators". In: *International Symposium on Database Applications in Non-Traditional Environments, DANTE'99*. 1999.
- [125] A. Seaborn and E. Prud. *SPARQL Query Language for RDF*. W3C Recommendation. W3C, 2008.
- [126] *Semantic Sensor Network Ontology*. 2005. URL: <https://www.w3.org/2005/Incubator/ssn/ssnx/ssn>.
- [127] M. Sengupta et al. "Role of Middleware for Internet of Things: A Study". In: *International Journal of Computer Science & Engineering Survey* 2.3 (2011), pp. 94–105.
- [128] J. G. Shanahan and L. Dai. "Large Scale Distributed Data Science Using Apache Spark". In: *21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD'15*. 2015.
- [129] S. Shekhar, H. Xiong, and X. Zhou. *Encyclopedia of GIS*. Springer, 2017.

- [130] Z. Shelby, K. Hartke, and C. Bormann. "The Constrained Application Protocol (CoAP)". In: (2014).
- [131] Z. Sheng et al. "A survey on the ietf protocol suite for the internet of things: standards, challenges, and opportunities". In: *IEEE Wireless Communications* 20.6 (2013), pp. 91–98.
- [132] A. Sheth, C. Henson, and S. S. Sahoo. "Semantic Sensor Web". In: *IEEE Internet Computing* 12.4 (2008), pp. 78–83.
- [133] W. Shi et al. "Edge Computing: Vision and Challenges". In: *IEEE Internet of Things Journal* 3.5 (2016), pp. 637–646.
- [134] P. Shvaiko and J. Euzenat. "Ontology Matching: State of the Art and Future Challenges". In: *IEEE Transaction on Knowledge and Data Engineering* (2011).
- [135] A. Sixsmith. "A Smart Sensor to Detect the Falls of the Elderly". In: *IEEE Pervasive Computing* 3.2 (2004), pp. 42–47.
- [136] J. Soldatos et al. "OpenIoT: Open Source Internet-of-Things in the Cloud". In: *Interoperability and Open-Source Solutions for the Internet of Things* (2015), pp. 13–25.
- [137] S. Staab and R. Studer. *Handbook on Ontologies*. Springer, 2009.
- [138] H. Stuckenschmidt et al. "Enabling technologies for interoperability". In: *14th International Symposium of Computere Science for Environmental Protection*. 2000.
- [139] M. Taheriyan et al. "Learning the Semantics of Structured Data Sources". In: *Web Semantics: Science, Services and Agents on the World Wide Web* 37-38 (2016), pp. 152–169.
- [140] K. M. M. Thein. "Apache Kafka: Next Generation Distributed Messaging System". In: *International Journal of Scientific Engineering and Technology Research* 3.47 (2014), pp. 9478–9483.
- [141] M. Uschold and M. Gruninger. "Ontologies: Principles, Methods and Applications". In: *Knowledge Engineering Review* 11.2 (1996), pp. 93–155.
- [142] H. van der Veer and a. Wiles. *Achieving Technical Interoperability - the ETSI Approach*. Tech. rep. ETSI, 2008.
- [143] O. Vermesan and P. Friess. "Digitising the Industry: Internet of Things Connecting the Physical, Digital and Virtual Worlds". In: vol. *IoT Platforms Initiative*. River Publisher, 2016, pp. 265–291.
- [144] W3C. *SOSA, SSN, O&M Mapping Table*. 2017. URL: https://www.w3.org/2015/spatial/wiki/Mapping_Table.
- [145] H. Wache et al. "Ontology-Based Integration of Information - A Survey of Existing Approaches". In: *IJCAI-01 Workshop*. 2001.
- [146] S. J. Walker. "Big Data: A Revolution That Will Transform How We Live, Work, and Think". In: *International Journal of Advertising* 33.1 (2015), pp. 181–183.
- [147] C. Wang et al. "A General Sensor Web Resource Ontology for Atmospheric Observation". In: *IEEE International Geoscience and Remote Sensing Symposium, IGARSS 2011*. 2011.

- [148] C. Wang et al. "Studentlife: Assessing Mental Health, Academic Performance and Behavioral Trends of College Students Using Smartphones". In: *ACM International Joint Conference on Pervasive and Ubiquitous Computing, UbiComp '14*. 2014.
- [149] M. Weiser. "The Computer for the 21st Century". In: *SIGMOBILE Mobile Computing and Communication Review* 3.3 (1999), pp. 3–11.
- [150] G. Xiao et al. "User Interoperability with Heterogeneous IoT Devices Through Transformation". In: *IEEE Transactions on Industrial Informatics* 10.2 (2014), pp. 1486–1496.
- [151] B. Yan and G. Huang. "Supply Chain Information Transmission Based on RFID and Internet of Things". In: *2nd ISECS International Colloquium on Computing, Communication, Control, and Management, CCCM '09*. 2009.
- [152] W. Yong. *Deliverable 2.1: Analysis of Technology Readiness of Technologies and Platforms for the Internet of Things*. 2016. URL: <http://big-iot.eu/media/deliverables/>.
- [153] M. Yu et al. "A Posture Recognition-Based Fall Detection System for Monitoring an Elderly Person in a Smart Home Environment". In: *IEEE Transactions on Information Technology in Biomedicine* 16.6 (2012), pp. 1274–1286.
- [154] M. Zaharia et al. "Spark: Cluster Computing with Working Sets". In: *2nd USENIX conference on Hot topics in cloud computing*. 2010.
- [155] T. Zahariadis et al. "FIWARE Lab: Managing Resources and Services in a Cloud Federation Supporting Future Internet Applications". In: *IEEE/ACM 7th International Conference on Utility and Cloud Computing, UCC 14*. 2014.
- [156] K. Zettsu et al. "Complex Asthma Risk Factor Recognition from Heterogeneous Data Streams". In: *IEEE International Conference on Multimedia and Expo Workshops, ICMEW*. 2015.
- [157] K. Zettsu et al. "Exploring Spatio-Temporal-Theme Correlation Between Physical and Social Streaming Data for Event Detection and Pattern Interpretation from Heterogeneous Sensors". In: *IEEE International Conference on Big Data*. 2015.
- [158] G. Zhao et al. "A System for Pesticide Residues Detection and Agricultural Products Traceability Based on Acetylcholinesterase Biosensor and Internet of Things". In: *International Journal of Electrochemical Science* 10.4 (2015), pp. 3387–3399.
- [159] J. Zhao et al. "The Study and Application of the IOT Technology in Agriculture". In: *3rd IEEE International Conference on Computer Science and Information Technology, ICCSIT '10*. 2010.
- [160] J. Zheng et al. "The Internet of THings". In: *IEEE Communications Magazine* 49.11 (2011), pp. 30–31.
- [161] E. Zimanyi and G. Pozzani. "Defining Spatio-Temporal Granularities for Raster Data". In: *British National Conference on Databases, BNCOD 2010*. 2010.