Ruhan dos Reis Monteiro

A Real-Time Reasoning Service for the Internet of Things

Dissertação de Mestrado

Dissertation presented to the Programa de Pós–graduação em Informática, of PUC-Rio, in partial fulfillment of the requirements for the degree of Mestre em Informática.

Advisor: Prof. Markus Endler

Rio de Janeiro
September 2018

Ruhan dos Reis Monteiro

A Real-Time Reasoning Service for the Internet of Things

Dissertation presented to the Programa de Pós–graduação em Informática, of PUC-Rio, in partial fulfillment of the requirements for the degree of Mestre em Informática. Approved by the undersigned Examination Committee.

Prof. Markus Endler
Advisor
Departamento de Informática – PUC-Rio

Profª. Noemi de La Rocque Rodriguez
Departamento de Informática – PUC-Rio

Prof. Edward Hermann Haeusler
Departamento de Informática – PUC-Rio

Prof. Marcio da Silveira Carvalho
Vice Dean of Graduate Studies
Centro Técnico Científico – PUC-Rio

Rio de Janeiro, September 14th, 2018

Ruhan dos Reis Monteiro

Graduated in Information Systems by the Pontifical Catholic University of Rio de Janeiro (Rio de Janeiro, Brasil)

Bibliographic data

Monteiro, Ruhan dos Reis

A Real-Time Reasoning Service for the Internet of Things / Ruhan dos Reis Monteiro; advisor: Markus Endler. – Rio de janeiro: PUC-Rio, Departamento de Informática, 2018.

v., 85 f: il. color. ; 30 cm

Dissertação (mestrado) - Pontifícia Universidade Católica do Rio de Janeiro, Departamento de Informática.

Inclui bibliografia

1. Informática – Teses. 2. Middleware;. 3. Internet das Coisas;. 4. Processamento de Eventos Complexos;. 5. Inferência;. 6. Tempo Real;. I. Endler, Markus. II. Pontifícia Universidade Católica do Rio de Janeiro. Departamento de Informática. III. Título.

CDD: 004

To my parents, for their support
and encouragement.

Acknowledgments

I leave a special thanks to my family for all the lessons of love, fellowship, friendship, charity, dedication, understanding, and forgiveness that you give me with each new day.

To Prof. Markus Endler, for guidance, competence, professionalism and dedication. So many times that we met, and although in some of them I arrived discouraged, just a few minutes of conversation and a few words of encouragement and there I was, with the same spirit of the first day of school. Thanks for believe in me and the incentives.

To my beloved fiancee Suzana, for all love, affection, understanding and support in so many difficult moments of this walk. Thank you for staying by my side, even without the usual caresses. Thank you for the gift of each day, for your smile and for knowing how to make me happy.

Thanks to all LAC friends, especially Vitor for all the academic support and valuable tips. I also thank all the teachers who participated in my graduation at PUC. Without this technical basis I would not have been able to get here.

This study was financed in part by the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES) - Finance Code 001.

Abstract

Monteiro, Ruhan dos Reis; Endler, Markus (Advisor). **A Real-Time Reasoning Service for the Internet of Things**. Rio de Janeiro, 2018. 85p. Dissertação de mestrado – Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro.

The growth of the Internet of Things (IoT) has brought the opportunity to create applications in several areas, with the use of sensors and actuators. One of the problems encountered in IoT systems is the difficulty of adding semantic relations to the raw data produced by the sensors and being able to infer new facts from these relations. Moreover, due to the fact that many IoT applications are online and need to react instantly on sensor data collected by them, they need to be analyzed in real-time. Streams are a sequence of time-varying data elements that should not be stored forever and queried on demand. Streaming data needs to be consumed quickly through ongoing queries that continue to analyze and produce new relevant data, i.e. stream of output/result events. The ability to infer new semantic relationships over streaming data is called Stream Reasoning. We propose a semantic model and a mechanism for real-time data stream processing and reasoning based on Complex Event Processing (CEP), RDF (resource description structure) and OWL (Web Ontology Language). This work presents a middleware service that supports continuous reasoning on data produced by sensors. The main advantages of our approach are: (a) to consider time as a key relationship between information; (b) flow processing can be implemented using CEP; (c) is general enough to be applied to any data flow management system (DSMS). It was developed in the Advanced Collaboration Laboratory (LAC) and a case study in the field of fire detection is conducted and implemented, elucidating the use of real-time inference on streams.

Keywords

Middleware; Internet of Things; Complex Event Processing; Reasoning; Real-Time;

Resumo

Monteiro, Ruhan dos Reis; Endler, Markus. **Um Serviço de Raciocínio Computacional em Tempo Real para a Internet das Coisas**. Rio de Janeiro, 2018. 85p. Dissertação de Mestrado – Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro.

O crescimento da Internet das Coisas (IoT) nos trouxe a oportunidade de criar aplicações em diversas áreas com o uso de sensores e atuadores. Um dos problemas encontrados em sistemas de IoT é a dificuldade de adicionar relações semânticas aos dados brutos produzidos por estes sensores e conseguir inferir novos fatos a partir destas relações. Além disso, devido à natureza destes sistemas, os dados produzidos por eles, conhecidos como *streams*, precisam ser analisados em tempo real. *Streams* são uma sequência de elementos de dados com variação de tempo e que não devem ser tratados como dados a serem armazenados para sempre e consultados sob demanda. Os dados em *streaming* precisam ser consumidos rapidamente por meio de consultas contínuas que analisam e produzem novos dados relevantes. A capacidade de inferir novas relações semânticas sobre dados em *streaming* é chamada de inferência sobre *streams*. Nesta pesquisa, propomos um modo semântico e um mecanismo para processamento e inferência sobre *streams* em tempo real baseados em Processamento de Eventos Complexos (CEP), RDF (Resource Description Framework) e OWL (Web Ontology Language). Apresentamos um middleware que suporta uma inferência contínua sobre dados produzidores por sensores. As principais vantagens de nossa abodagem são: (a) considerar o tempo como uma relação-chave entre a informação; (b) processamento de fluxo por ser implementado usando o CEP; (c) é geral o suficiente para ser aplicado a qualquer sistema de gerenciamento de fluxo de dados (DSMS). Foi desenvolvido no Laboratório de Colaboração Avançada (LAC) utlizando e um estudo de caso no domínio da detecção de incêndio é conduzido e implementado, elucidando o uso de inferência em tempo real sobre *streams*.

Palavras-chave

Middleware; Internet das Coisas; Processamento de Eventos Complexos; Inferência; Tempo Real;

Table of contents

List of figures

List of tables

List of Abreviations

CEP	Complex Event Processing
GW	Gateway
IoT	Internet of Things
IoMT	Internet of Mobile Things
M-Hub	Mobile Hub
M-OBJ	Mobile Object
SDDL	Scalable Data Distribution Layer
SCA	SDDL Core Application

1 Introduction

The Internet of Thing (IoT) is a system of physical objects that can be discovered, monitored, controlled or interacted with by electronic devices that communicate over various networking interfaces and eventually can be connected to the wider internet. "Things", in IoT, can refer to a variety of devices such as automobiles with built-in sensors or even biochip transponders on farm animals. These devices collect useful data that can be processed by other systems, by demand or real-time.

IoT is the new impetus in the industrial sector and is empowering industrial engineering with sensors, software and data analysis to create smart automated systems. The driving philosophy behind IoT is that digital transformation, with intelligent machines, is more accurate and consistent than humans in data communication, and that data can help companies detect inefficiencies and problems sooner.

Major technology companies, such as Amazon, have invested in the emerging IoT market, putting the area in evidence. These intelligent so-called "Things" are then integrated into a network infrastructure, with their respective identifiers, physical attributes and interfaces. It is estimated that by the year of 2020, more than 30 billions objects are connected to the to the network [1].

Examples of potential applications include monitoring vehicles to optimize driving routes, intelligent highways with warming messages and diversions according to climate conditions, for example. In industry, we can use IoT to build a smart manufactory. In short, smart manufacturing is the use of IoT devices to improve the efficiency and productivity of manufacturing operations. Typically, this involves retrofitting sensors to existing manufacturing equipment, but new manufacturing equipment often comes with IoT sensors pre-installed.

The next biggest revolution in industry is being carried out by IoT, and it's called Industry 4.0. Industry 4.0 is an expression that encompasses some technologies for automation and data exchange and uses concepts of cyber-physical Systems, Internet of Things and Cloud Computing. This new revolution facilitates the viewing and execution of "Intelligent Factories" with their

[1]https://www.idc.com/infographics/IoT

modular structures, cyber-physical systems monitor physical processes, create a virtual copy of the physical world and make decentralized decisions. With the Internet of Things, cyber-physical systems communicate and cooperate with each other and with humans in real-time, and through cloud computing, both internal and intra-organizational services are offered and used by value chain participants.

However, so far in current IoT systems, sensing and actuation is mostly done at the lowest data level. In contrast, many IoT applications demand higher level situation awareness. To achieve this, semantic models for data stream analysis are necessary. Semantic models are formally defined concepts and relations on which reasoning engines can operate to derive new information and knowledge about a system and its environment. The main problem is that current semantic models (designed for the Semantic Web) are not suitable for efficient and real-time reasoning. Current data analysis for IoT systems is either done off-line or lacks any semantic-based reasoning. Real-world IoT systems must be able to interpret and reason about uncertain sensor observations to effectively operate in the physical world.

There is thus an increasing need to build scalable systems to support such applications. The complex nature of these systems and their rapid evolution, coupled with the huge volume of streaming data and the need for real-time processing, raise many computational challenges that have not been addressed in prior work.

1.1 Problem Statement

Since IoT nodes (sensors and actuators) produce low level data, it is a challenge to extract higher level knowledge, which would facilitate the understanding of complex situations and allow to address them properly. Suitable knowledge models and processing approaches are required to make it possible to deduce new knowledge for IoT applications in an efficient manner. Structuring the data semantically and relating them to an ontology, brings the possibility to contextualize the data, relate them to the environment's stimuli and consequently extract a higher level information or even implicit knowledge that can be used to automate some kind of work.

Knowledge bases and reasoning techniques are good options for handling and managing the acquisition of new knowledge and deduction. The related Semantic Web technologies are widely accepted as the data model for semantically representing static information. However, the approaches developed by the Semantic Web community are designed for a static processing over knowledge data and one of the main characteristics of an IoT system is the need of a fast response over an environment stimuli. IoT systems often refer to real-time data analysis systems, like monitoring a smart manufacturing, a smart home or even an autonomous vehicle. These kind of application needs a response in seconds, or, in the case of an autonomous vehicle, in milliseconds. So, besides having a semantic annotated data with a knowledge representation, we need to manage this knowledge in a spectrum of a real-time system.

1.2 Objective and Contribution

The main objective of this dissertation is to propose a processing model for IoT that deals with the challenges presented in the previous section. For which, we created a Real-Time service that collects the data from the Things, relates them to a semantic model and derive higher level semantic annotated data and implicit knowledge. We propose here an approach for the discovery and analysis of IoT nodes and later we use semantic and reasoning techniques, over real-time generated data, in order to discover new implicit facts that are more tangible for a system actor.

As a contribution, this dissertation addresses 1) the discover of IoT nodes placed in the environment, 2) the generation of semantic annotated data, 3) semantic reasoning over streams, creating a Stream Reasoning Service and 4) a systematic evaluation if this real-time approach is able to scale in relation to the volume of data collected from sensors and also the complexity of the knowledge model.

1.3 Outline

The sequence of this dissertation is organized as follows:

– **Chapter 2 - Fundamentals**: In this chapter we present the fundamental concepts and technologies that were used to build this work.

– **Chapter 3 - Related Works**: This chapter presents and discusses some works related to Stream Reasoning.

– **Chapter 4 - Stream Reasoning Service**: In this chapter we introduce our approach and implementation to deal with the Stream Reasoning for IoT.

– **Chapter 5 - A fire detection application**: In this chapter we present an IoT application, in the area of fire detection, that makes use of our service.

- **Chapter 6 - Performance Analysis**: We analyze the performance of our Stream Reasoning Service on several different aspects.
- **Chapter 7 - Discussion**: We discuss the results obtained in the service implementation and performance analysis.
- **Chapter 8 - Conclusion**: The work is summarized and future research is suggested.

The research has been carried out in the scope of the ESMOCYP cooperation project between PUC-Rio, Federal University of Maranhão and University of Stuttgart.

2
Fundamentals

This chapter presents the fundamental concepts and technologies used in this dissertation. Section 2.1 introduces the concept of Complex Event Processing, which is a fundamental part of this work. Section 2.2 discusses the Semantic Web, from which we use RDF and OWL to represent the semantic modeling. Then, Section 2.3 explains the concept of Description Logics and Reasoning, which is used to obtain implicit knowledge about an ontology. In Section 2.4, we present the concept of Stream Reasoning, a discipline that joins sections 2.1, 2.2 and 2.3. Finally, section 2.5 presents the ContextNet, which was responsible for all the discovery and communication with the sensors and actuators.

2.1
Complex Event Processing

Complex Event Processing (CEP) is a technology for dynamically processing near-real-time event data flows. It was proposed in the mid-1990s by David Luckham (3). In contrast to the DBMS (Database Management System) paradigm, where data is stored for a later query, CEP stores continuous queries and performs a data flow on them. In other words, CEP allows the continuous analysis of an event data flow through stored queries.

Classical database systems and data warehouses are concerned with what happened in the past. In contrast, CEP is about processing events upon their occurrence, with the goal to detect what has just happened or what is about to happen. For example, an event may represent a sensor reading, a stock price change, a complied transaction, a new piece of information, a content update made available by a Web service and so forth. In all these situations, it is reasonable to compose simple (atomic) events into derived (complex) events, in order to structure the course of affairs and describe more complex dynamic matters. CEP deals with real-time recognition of such derived events, i.e., it processes continuously arriving events with the aim of identifying occurrences of meaningful derived events (according to predefined event patterns or event operations).

A stream of events within a CEP engine is a sequence of events created and sent by producer elements (3). Events are processed according to defined queries and the result is sent to consumer elements. In the IoT spectrum, the producing events are the sensors that collect the raw data from the environment, from users or from components/nodes of the communication and computing system. The figure 2.1 represents the basic architecture of a CEP engine.

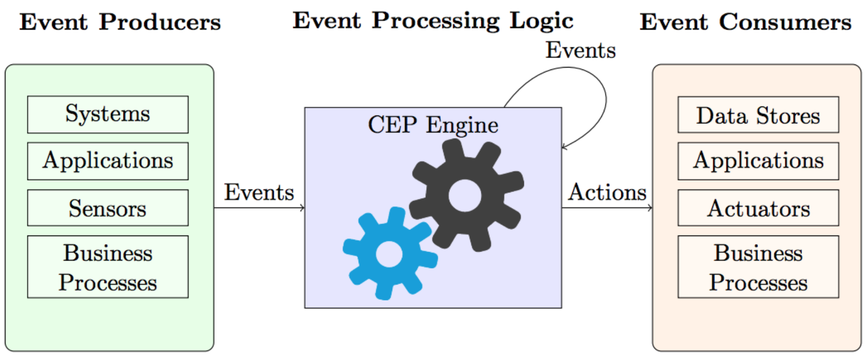

Figure 2.1: CEP engine architecture (63)

An event producer is an entity that introduces events into a system which implements an CEP architecture. An event producer is also known as an event source. The producer listens to an environment that is attached to, and provides events from that environment to an attached Event Processing Agent (EPA). For example, an event producer can be attached to a physical sensor so that the sensor detects a change, the producer creates an object that represents the change and emits it as an event.

An event consumer is an entity that receives events from a system which implements an EPA. An event consumer consumes events and uses them for real-time analytics or further computation.

An EPA is a software module that processes events (19). An EPA takes events as input and, by applying an CEP operation, it outputs complex events. In (20), the authors give EPAs the following classification:

- **Filter agent** - filters out irrelevant events with respect to a filtering condition. For example, filters out temperature events whose temperture is bellow 30 celsius. The goal is to increase performance by discarting irrelevant events.
- **Pattern detection agent** - detects an event pattern based on conditions (e.g., temporal, spatial). For instance, an agent may detect the temperature increasing of x% whitin a particular time.

– **Transformation agent** - transforms input events according to transformations operations.

Transformation agents can be classified based on the cardinality of their inputs and outputs.

– **Translate agent** - takes each incoming event object and operates on it independently of preceding or subsequent event objects. It performs a single event in, single event out kind of operation.
– **Split agent** - takes a single incoming event and emits a stream of multiple event objects.
– **Aggregate event** - takes a stream of incoming event objects and produces an output event that is a function of the incoming events.
– **Compose agent** - takes two streams of incoming event objects and operates on them to produce a stream of output events. This is similar to the join operator in relational algebra, except that it joins streams of events rather than tables of data.

CEP generated events are denominated as complex events because they can produce another events to be processed or reinserted in the CEP machine. The engine allows the composition of events, in which intermediate events can be used to define other complex events at a higher level. For example, an event called Fire Alarm can be built from the presence of high temperature and low humidity.

In most Stream processing engines like Storm [1] and S4 [2], users write code to create the operators, wire them up in a graph and run them. Then the engine runs the graph in parallel using many computers. In contrast, in CEP engines, users can write queries using a higher level language such as an SQL like query language. Also CEP has build in operators such as time windows and temporal event sequences.

The queries are defined by a language denominated CQL (Continuous Query Language), which is very similar to SQL (Structured Query Language) from relational database systems. The Esper implementation defines a language denominated EPL (Event Processing Language). To illustrate, at Listing 2.1, we have a query that looks for events in which their temperature is higher than a certain level. This query searches for events contained in a 5-second window, and, with these events, does a mean calculation.

[1]http://storm.apache.org/
[2]http://incubator.apache.org/projects/s4.html

```
SELECT avg(sensorValue)
FROM Event.win:time(5 sec)
```

Listing 2.1: CEP query example

Another very important concept in CEP are windows. Windows allow to define a search scope on a data stream, such as an interval of time or amount of events. Having defined a window of x seconds, it is possible to detect, for example, if an event happens or not within a predefined interval.

There are two kinds of CEP windows, batch and time windows. Batch windows have fixed length size and waits until a certain number of events or time units (defined in the window) is reached. On the other hand, a time window processes only the events that have happened between the last x units of time.

There are several implementations of CEP available, such as Esper [3], Microsoft StreamInsight [4] and Apache Flink [5]. In this work we chose Esper for being open-source and distributed under the GNU GPL license. Esper follows an object-oriented pattern with dynamic data types, which allows you to easily map an event to an object in a language such as Java, for example. In addition Esper has an available Android implementation, called Asper (26), making possible the use of CEP on smartphones.

Complex event processing is a key enabler in IoT settings and Smart Cyber-physical systems (CPS) as well. Processing dense and heterogeneous streams from various sensors and matching patterns against those streams is a typical task in such cases (21). The majority of these techniques rely on the fact that representing the IoT system's state and its changes is more efficient in the form of a data stream, instead of having a static, materialized model. Reasoning over such stream-based models fundamentally differs from traditional reasoning techniques and typically require the combination of model transformations and CEP (22).

2.2 Semantic Web

The term "Semantic Web" was coined by Tim Berners-Lee (1), the inventor of the World Wide Web and director of the World Wide Web Consortium ("W3C"), which oversees the development of proposed Semantic Web standards. He defines the Semantic Web as "a web of data that can be processed directly and indirectly by machines". Some technologies used to

[3]http://www.espertech.com/esper/
[4]https://msdn.microsoft.com/pt-br/library/ee362541(v=sql.111).aspx
[5]https://flink.apache.org/

describe the Semantic Web are the Resource Description Framework (RDF) and the Web Ontology Language (OWL).

On the Semantic Web (SemWeb), computers do the browsing (and searching, and querying, and...) for us. The SemWeb enables computers to seek out knowledge distributed throughout the Web, mesh it, and then take action based on it. Take an analogy: the current web is a decentralized platform for distributed presentations, while the SemWeb is a decentralized platform for distributed knowledge. Resource Description Framework (RDF) is the W3C standard for encoding knowledge.

2.2.1 Resource Description Framework

RDF is a standard model for data interchange. RDF has features that facilitate data merging even if the underlying schemas differ, and it specifically supports the evolution of schemas over time without requiring all the data consumers to be changed (34).

RDF extends the linking structure of the Web to use URIs to name the relationship between things as well as the two ends of the link (this is usually referred to as a "triple"). Using this simple model, it allows structured and semi-structured data to be mixed, exposed, and shared across different applications.

The foundation is breaking knowledge down into a labeled, directed graph. Each edge in the graph represents a fact, or a relation between two things. The edge in the example from the node vincent donofrio labeled starred in to the node the thirteenth floor represents the fact that actor Vincent D'Onofrio starred in the movie "The Thirteenth Floor." A fact represented this way has three parts: a subject, a predicate, and an object. The subject is what's at the start of the edge, the predicate is the type of edge (its label), and the object is what's at the end of the edge.

The RDF metadata model is based upon the idea of making statements about resources in the form of a subject-predicate-object expression, called a triple in RDF terminology. The subject is the resource, the "thing" being described. The predicate is a trait or aspect about that resource, and often expresses a relationship between the subject and the object. The object is the object of the relationship or value of that trait.

2.2.2 Web Ontology Language

An ontology defines the terms used to describe and represent a domain, that is, an ontology is a description of concepts and relationships that can be used by people or software agents that want to share information within a domain. For these characteristics, ontologies are one of the key technologies in the emerging Semantic Web.

The Web Ontology Language (OWL) is a Semantic Web language designed to represent rich and complex knowledge about things, groups of things, and relations between things (64). OWL is a computational logic-based language such that knowledge expressed in OWL can be exploited by computer programs, e.g., to verify the consistency of that knowledge or to make implicit knowledge explicit. OWL documents, known as ontologies, can be published in the World Wide Web and may refer to or be referred from other OWL ontologies. OWL is part of the W3C's Semantic Web technology stack [6]. According to Harmelen and McGuinness (23), an OWL has three incremental sub-languages:

– OWL Lite: is a sub-language of OWL DL that uses only some characteristics of the OWL language and has more limitations than OWL DL or OWL Full.
– OWL DL: is used by users who want maximum expressiveness, with completeness (all conclusions are guaranteed to be computable) and decidability (all computations will end in a finite time) computational. It includes all constructions OWL language, but these constructs can only be used under certain restrictions. DL stands for description logics (DL), a research area that studies a particular piece of first-order logic.
– OWL Full: is used by users who want maximum expressiveness with no computational guarantee. OWL Full and OWL DL support the same set of OWL constructs, although with a little difference. While OWL DL imposes restrictions on the use of RDF and requires disjunction of classes, properties, individuals, and data values, OWL Full allows to mix OWL with RDF Schema and does not require the disjunction of classes, properties, individuals and data values. That is, a class can be both a class and an individual.

[6]https://www.w3.org/OWL/

2.2.3 SPARQL

SPARQL is the standard query language and protocol for Linked Open Data on the web or for semantic graph databases (also called RDF triplestores) (17). SPARQL, short for “SPARQL Protocol and RDF Query Language”, enables users to query information from databases or any data source that can be mapped to RDF. The SPARQL standard is designed and endorsed by the W3C and helps users and developers focus on what they would like to know instead of how a database is organized. In addition, a SPARQL query can also be executed on any database that can be viewed as RDF via a middleware. For example, a relational database can be queried with SPARQL by using a Relational Database to RDF (RDB2RDF) mapping software (35).

RDF data can also be considered a table with three columns – the subject column, the predicate column, and the object column. The subject in RDF is analogous to an entity in a SQL database, where the data elements (or fields) for a given business object are placed in multiple columns, sometimes spread across more than one table, and identified by a unique key. In RDF, those fields are instead represented as separate predicate/object rows sharing the same subject, often the same unique key, with the predicate being analogous to the column name and the object the actual data. Unlike relational databases, the object column is heterogeneous: the per-cell data type is usually implied (or specified in the ontology) by the predicate value. Also unlike SQL, RDF can have multiple entries per predicate; for instance, one could have multiple "child" entries for a single "person", and can return collections of such objects, like "children".

Thus, SPARQL provides a set of analytic query operations such as JOIN, SORT, AGGREGATE for data whose schema is intrinsically part of the data rather than requiring a separate schema definition. However, schema information (the ontology) is often provided externally, to allow joining of different datasets. In addition, SPARQL provides specific graph traversal syntax for data that can be thought of as a graph.

The example in Listing 2.2 demonstrates a simple query that leverages the ontology definition foaf ("friend of a friend"). Specifically, the following query returns names and emails of every person in the dataset:

This query joins together all of the triples with a matching subject, where the type predicate, "a", is a person (foaf:Person), and the person has one or more names (foaf:name) and mailboxes (foaf:mbox). The result of the join is a set of rows – ?name and ?email.

```
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
SELECT ?name
       ?email
WHERE
  {
    ?person  a          foaf:Person  .
    ?person  foaf:name  ?name  .
    ?person  foaf:mbox  ?email  .
  }
```

Listing 2.2: SPARQL query example

Forging data with URIs allows data to be unambiguously referenced across applications and overcomes the constraints posed by local search. Consequently, additional application-specific APIs can be developed and can refer to that data.

These design choices – enabling queries over distributed sources on non-uniform data – are not accidental. SPARQL is designed to enable Linked Data for the Semantic Web. Its goal is to enrich data by linking it to other global semantic resources, thus sharing, merging and reusing data in a more meaningful way.

2.3 Description Logics and Reasoning

Description logics (DL) (49) are a family of formal knowledge representation languages. Many DLs are more expressive than propositional logic but less expressive than first-order logic. In contrast to the latter, the core reasoning problems for DLs are (usually) decidable. There are general (59), spatial (52, 53, 54), temporal (55, 56, 60), spatio-temporal (51, 61, 62), and fuzzy descriptions logics (57, 58), and each description logic features a different balance between DL expressivity and reasoning complexity by supporting different sets of mathematical constructors (65).

Knowledge representation system based on DLs consists of two components - T-Box and A-Box. The T-Box describes the basic types of concepts (i.e. terminology), i.e., the ontology in the form of concepts and roles definitions (e.g. Person, Sensor), while the A-Box is an assertion component — a fact associated with a terminological vocabulary within a knowledge base (e.g. james, sensor1). Concepts describe sets of individuals, roles describe relations between individuals. Both can be represented using OWL and RDF formats (50).

A reasoning engine is a piece of software able to infer logical consequences from a set of asserted facts or axioms. Every reasoner works with some set of axioms and an axiom describes some logical fact. The capabilities of a reasoner depend on the expressiveness of the kind of logic that the reasoner uses and the axioms provided for the reasoner and logic to work against.

Reasoners are sometimes referred to as inference engines because while, as stated above, reasoners work with asserted facts and can also use the rule of logic to deduce theorems. Theorems are indirectly deduced facts. Theorems are deductions which can be proven by constructing a chain of reasoning by applying axioms. Basically, a reasoner and an inference engine are the same thing [7]. The inference rules are commonly specified by means of an ontology language, like OWL, and often a DL language. Many reasoners use first-order predicate logic to perform reasoning; inference commonly proceeds by forward chaining and backward chaining (43).

Most of the IoT systems lack of semantic representation of their environment, nodes/devices, and users, that are directly related to the probed sensor data and the issued activation commands. By adding a semantic model to the IoT application and applying reasoning techniques, we can derive new facts based on the current state of the system. For example, consider that a person is in the range of a proximity sensor and another person is also near to the same sensor. By adding a semantic annotation that this same sensor is connected to a room and put these facts in a reasoning engine, we can infer a logical consequence that the two people are near each other or even they are in the same room.

2.4 Stream Reasoning

Stream reasoning is defined as the capacity of generating a stream of conclusions by means of reasoning over terminological or assertional axioms (4). Stream reasoning combines reasoning and stream processing techniques. Such a combination enables handling data continuously produced from a large amount of sources, processing several streams on-the-fly, and implementing real-time services. In general, stream reasoning meets the requirements of processing dynamic, heterogeneous, and scalable data for IoT (5). A stream reasoning system must cope with several things (16):

– **Heterogeneous formats and access protocols:** Streams can appear in different formats, from text streams from Twitter[8] to relation data over binary protocols, such as data streams originated by sensor networks.

[7]http://xbrl.squarespace.com/journal/2015/7/30/understanding-the-utility-of-a-reasoner-or-inference-engine.html

[8]www.twitter.com

– **Semantic Modeling:** A stream is observed through a window, which can be a span in time or a number of elements. By their very nature, streams of data can be inspected only while they flow. If information is not captured and immediately summarized (aggregated, for example), then information reconstruction may be impossible. Given that aggregation can perform lossy data compression, stream reasoning will require methods to determine which inferences are possible even after summarization and which must be performed before summarization.

  Data streams are naturally time-stamped, but the time validity of static information sources is normally not stated. Thus, merging data streams with static information can create hybrid data that must be carefully managed. Also needed are vocabularies to state future validity of information.

– **Scale:** Scale is an issue for stream reasoning due both to the high throughput of incoming data from sensors and to the need to link streaming data with large static knowledge bases. But for many applications, a limited amount of streaming data and knowledge are sufficient for a specific stream-reasoning task. In these cases, the streaming data should be sampled, abstracted, and approximated.

– **Continous Processing:** Stream reasoning requires continuous processing, because queries, once registered, remain continuously active while data flows into the stream-reasoning system.

– **Real-Time Constraints:** Stream-reasoning systems must provide answers, of knowledge-based queries, before they become useless.

Stream reasoning is a natural advance of stream processing. The most advanced stream processing systems were developed in the context of Data Stream Management Systems (DSMSs) and Complex Event Processors (CEPs). DSMSs transform data streams in time-stamped relations and process them with well known techniques such as relational algebras (24). DSMSs allow the construction of systems able to compute aggregations and statistics (e.g. averages and Pearson correlation) over streaming data. On the other hand, Complex Event Processing Systems (CEPs) look for patterns in the streams to identify when complex events occur (3) and focus on the derivation of (complex) events from the inbound stream of data with patterns of events (e.g., sequences). CEP, DSMS, Knowledge Representation (KR) and SemWeb supplied the ingredients to Stream Reasoning (16).

Extending DSMS and CEP to do reasoning tasks is one of the first issues researched in stream reasoning (8). In SemWeb this was addressed by adding the reasoning techniques to SPARQL (6). Entailment regimes brought an inference process in SPARQL (7). They affect the basic graph pattern evaluation by extending the matching definition to take the edges and the nodes that can be logically inferred from those explicitly stated into account.

The stream reasoning reference model described in (8) and illustrated in Figure 2.2, shows us that it is possible to take the inference at different moments. Inference can take place before or after the window (giving, respectively, stream and window-level entailments), or still after that the window content is merged (giving graph-level entailment).

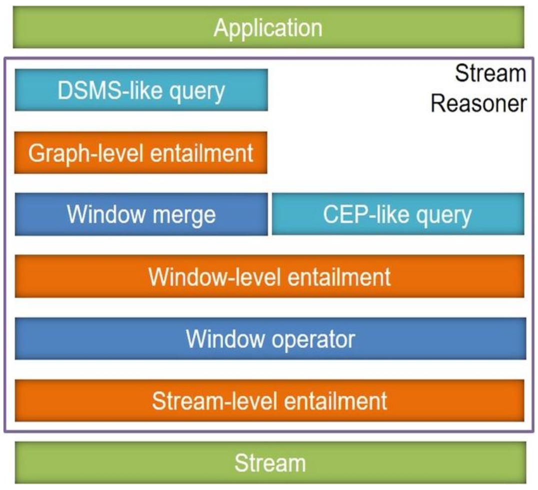

Figure 2.2: Stream Reasoning Reference Model (8)

The *window operators* create a time-dependent finite views over the streams, namely windows, over which processors perform the tasks. Window contains a portion of the input streams, i.e. a set of timestamped data items, that represents the data needed to solve the task at the current time instant. Several types of window operators exist, defined in CEP and DSMS research (25).

In the *graph-level entailment* the inference process occurs after the window merges. That means a window captures a portion of the stream and a merge operation creates a collection from the stream item contents. This enables the execution of typical DSMS-like queries, e.g. aggregations and filters, but make not possible to evaluate CEP-like queries, since temporal annotations are lost and it is not possible to verify if temporal constraints are satisfied.

CEP performs the processing over time annotated data, since engines use them to determine if temporal constraints defined in the event patterns are satisfied. The stream time mapped behaviour is modelled through *landmark windows*. A initial time instant is fixed and the window expands over time to capture portions of the stream.

DSMSs perform operations that do not require to process time annotations after that windows have been computed, like aggregations and filters. The *window merge* is an operation that moves from temporal data to atemporal one. The *sliding window* operator creates a window with a fixed width of units or data items. The operator slides the window over time, capturing the most recent part of the stream.

The *window-level entailment* applies the inference process on the non-merged stream items captured by the window operators declared in the query. Differently from graph-level entailment, which works on graphs (ontologies), the window-level entailment applies the inference process to a window, i.e. a finite sequence of time-stamped data items. This type of entailment overcomes the main drawback of the *window-level entailment* that is the process uses only a subset of the information available in the stream: it considers the data item contents but not the relative temporal annotations.

Stream-level entailment consideres a larger portion of the stream than the one defined by the user through window operator. In this case the reasoning is made on the top of a landmark window, which is a window that captures the stream from an initial time instant (e.g. when the source starts to supply the data when the engine starts to monitor the stream) up to now.

Stream reasoning is one approach for querying and reasoning over continuous distributed data streams. With a streaming query engine, simultaneous queries can be passed to a reasoner as an input. Thus, new knowledge can be inferred and RDF graphs can be updated on-the-fly (5). Stream reasoning techniques over a time-based data model where data items can be annotated with time-stamps, either occurrence time or validity time period. Defining a suitable time processing model is probably one of the main contributions of stream reasoning technologies to IoT.

2.5 ContextNet

ContextNet is a scalable middleware for the Internet of Mobile Things (27). It is composed of a) SDDL, a suite of cloud-based services for communication and processing, interacting through DDS, and b) Mobile-Hub, a smartphone based component that discovers, connect with and acts as an internet proxy for Bluetooth-enabled smart objects. ContextNet also provides CEP engines for sensor data stream processing both in SDDL and Mobile-Hub, on smartphones, at the IoT edges.

2.5.1 SDDL

The most basic layer of the ContextNet architecture is its communication middleware, named Scalable Data Distribution Layer (SDDL), which connects stationary nodes of a wired "core" network with all the mobile nodes. SDDL employs two communication protocols: DDS's (Distribution Service for Real-Time Systems) RTPS (37) for the wired communication within the ContextNet core network, and the Reliable UDP protocol (RUDP) for the inbound and outbound communication between the core network and the mobile nodes.

The DDS [9] is a standard from the OMG, which specifies a high performance, robust and scalable middleware architecture for real-time data distribution, with Quality of Service (QoS) contracts between producers and consumers of data (e.g. best effort or reliable communication, data persistency and several other message delivery optimizations, etc.).

The Reliable UDP (RUDP) [10] protocol is the basis for the Gateway-mobile node interaction. It implements some TCP functionality on the top of UDP, and has been customized to handle intermittent connectivity, Firewall/NAT traversal and robustness to changes of IP addresses and network interfaces.

As part of the core network based on DDS, two types of SDDL nodes have distinguished roles: a) the Gateway (GW) that defines a unique Point of Attachment (PoA), for connection with the mobile nodes and b) The PoA-Manager is responsible for two things: to periodically distribute PoA-List to the mobile nodes, and to eventually request some mobile nodes to switch to a new Gateway/PoA. The PoA-List is always a subset of the group of all available Gateways in the SDDL, and the order in the list is relevant, i.e. the first element points to the preferred Gateway/PoA.

In the SDDL, every mobile node and every Gateway have a unique identifier (ID). While RUDP messages carry only the mobile node's ID, the ID of the GW currently serving the mobile node is automatically attached to any message (or location update) entering the SDDL core network. By this, any corresponding node can learn which is the mobile's current GW, and most messages addressed to a mobile node will thus carry also a Gateway ID, allowing them to be directly routed to the corresponding Gateway. However, if the mobile node becomes suddenly unreachable/disconnected, its most recent Gateway, will notify this to all other nodes in the SDDL core network, by which the Gateway ID will be omitted in future messages to the mobile node.

[9] http://portals.omg.org/dds/
[10] https://tools.ietf.org/html/draft-ietf-sigtran-reliable-udp-00

However, even in this case where the current Gateway of a mobile node is not specified, messages to the mobile node will be delivered, because they will be received by all Gateways.

2.5.2 Mobile-Hub

The concept of the Mobile Hub (M-Hub) (28) is independent of the protocol/technology used for Internet connectivity and communication with the cloud. The current implementation is based on the utilization of SDDL, which connects mobile nodes with IP-based wireless data connection to stationary nodes in a wired core network, the SDDL Core. The mobile nodes become the gateways (hubs) of communication for providing Internet connectivity to Smart Objects, as shown in Figure 2.3.

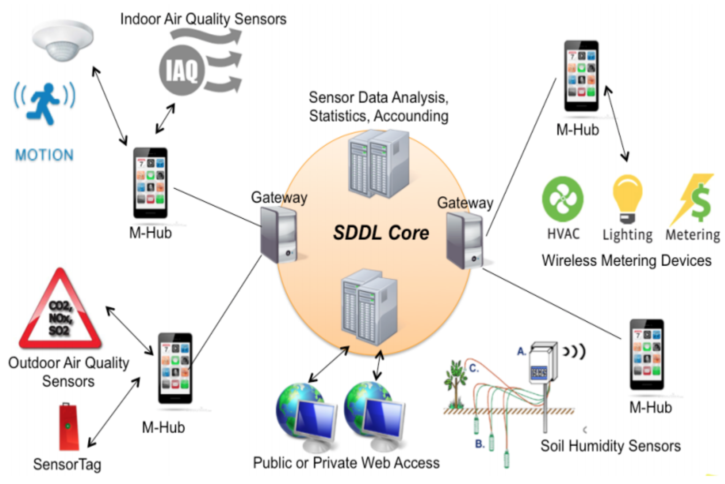
Indoor Air Quality Sensors
IAQ
MOTION
M-Hub
Sensor Data Analysis, Statistics, Accounding
Gateway
Gateway
SDDL Core
M-Hub
HVAC
Lighting
Metering
Wireless Metering Devices
CO2, NOx, SO2
Outdoor Air Quality Sensors
M-Hub
SensorTag
Public or Private Web Access
A.
B.
C.
M-Hub
Soil Humidity Sensors

Figure 2.3: Example of M-OBJs sensing using the SDDL and M-Hubs

As a key feature, M-HUB enable us to enrich the Mobile Objects (M-OBJ's) data streams with contextual information, obtained from its own sensors, such as current geographic position, temperature or humidity. M-Hub, which is represented by Figure 2.4, is multi-threaded and consists of the following services and managers:

- **Location Service** - is responsible for sampling the M-Hub's current position and attaching it to a message that is sent by the M-Hub to the Gateway.
- **S2PA Service** - uses the S2PA library (Short-Range Sensor, Presence and Actuation API) and is responsible for periodically doing scanDevies in all the supported WPAN technologies, registering discovered the

M-OBJ's IDs and theie capabilities in the SensServie Registry, and transcoding sensor data and commands from the specific SF protocol format to Java objects to be transmitted to the GW, and vice-versa.

– **Connection Service** - manages a *msg buffer* of ready-to-send messages.

– **MEPA Service** (29) - enables a CEP engine in the M-Hub. The MEPA Service is subscribed to all the messages that are sent from the S2PA and Connection services, since the former collects the data from the M-OBJs, and the latter receives commands from the cloud to modify the behavior of the MEPA Service itself (e.g. deploy a new CEP rule). Every time an event pattern is detected (which leads to the generation of a new complex event), it will be published to any interested component that could be the Connection service, or another rule in the MEPA Service.

– **Energy Manager** - controls the device's current energy level (LOW, MED, HIGH). From time to time sample's the device's battery level and checks if it is connected to a power source.

– **Handover Manager** - interacts with other M-Hubs (detected nearby) so as to proactively share information and parameters about M-OBJs, and ultimately swap responsibility for handing-out or handling-in M-OBJs, with these M-Hubs.

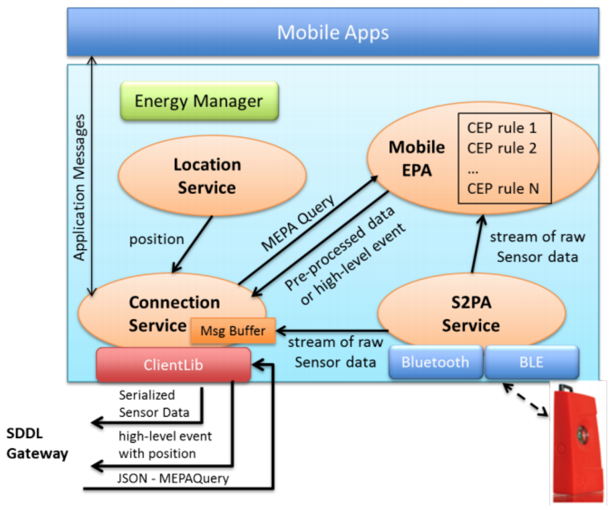

Figure 2.4: M-Hub architecture

2.6 Summary

This chapter presented the fundamental topics that underprints this dissertation, the Complex Event Processing (CEP), Semantic Web and its related technologies, Description Logics/Reasoning, Stream Reasoning and, finally, the ContextNet middleware.

First we presented the Complex Event Processing concept, a real-time data stream processing paradigm. CEP provides logical operators, primitives and time windows to react and process event streams in real-time. These concepts are used to specify continuous queries, that analyze, process and produces higher level events. Each query is executed by an Event Processing Agent (EPA), which continuously receives incoming events and outputs derived facts that are delivered to the event consumers.

In sequence, the Semantic Web and its related technologies brought the opportunity the structure the web data and relate them to semantic models. Several frameworks and languages were proposed. The Resource Description Framework (RDF) defines a general method for the conceptual description or information modeling based on triples, the Web Ontology Language (OWL), which is a language to define and instantiate ontologies and SPARQL, which is a query language that is able to retrive and manipulate data storage in RDF format.

The Description Logics language, together with reasoning techniques, brings the possibility to infer logical facts from a set of asserted facts or axioms. Having a semantically structured data based on an ontology, we can use the Reasoning Engines to discover new facts that are implicit to the model and make them explicit, thus generating facts of higher level complexity.

The Stream Reasoning concept is a combination of technologies and techniques presented by the Semantic Web and CEP. The Semantic Web works on a static knowledge that rarely changes. CEP, in turn, processes heterogeneous low level data in real-time, applying operators and pattern rules that generates other types of events. Stream Reasoning was born to mitigate the limitations of these two parts, enabling real-time processing of semantic models and continuously inferring new facts.

Finally, we presented the ContextNet middleware, responsible for the discovery and processing of IoT M-OBJs data. In addition, ContextNet also enables data processing in the cloud through the SDDL Core.

3
Related Work

A market overview on real-time complex event processing systems and tools is provided in (9). Some of the existing CEP systems can integrate and access external reference data sources. However, these systems do not provide any inference on external KBs and do not consider reasoning on relationships of events to other non-event concepts. In (18, 19), the authors propose a semantic model for for data stream processing and real-time reasoning based on the concepts of Semantic Stream and Fact Stream as natural extension of Complex Event Processing and RDF.

Some stream reasoning languages and processing approaches are also proposed. Stream reasoning approaches like (10) are proposed for reasoning on RDF stream. These engines typically make use of extended SPARQL-based query languages over continuous data streams. Continuous SPARQL (C-SPARQL) language (11, 12), CQELS-QL (13) and SPARQLstream (38). Such languages share the idea of extending SPARQL with sliding windows. C-SPARQL extends the FROM clause in order to support sliding windows, while CQELS-QL pushes sliding windows in the GRAPH clause. morph-stream, an implementation of SPARQLstream, adopts an OBDA-like approach: it processes relational streams by transforming the query from SPARQLstream to one to be registered to a DSMS engine such as Esper. Other approaches are also proposed, like STARQL (39), IMaRs (40), Streaming Knowledge Bases (41), Sparkwave (42), INSTANTS (44) and The Streaming Linked Data (45).

C-SPARQL (Continuous SPARQL) is one of the first contributions in the area of stream reasoning. It is a language for continuous queries over streams of RDF data. C-SPARQL extends SPARQL for querying RDF streams. An RDF stream is defined as an ordered sequence of pairs, where each pair is constituted by an RDF triple and its timestamp t: $< Subject_i, Predicate_i, Object_i >$

C-SPARQL supports timestamped RDF triples, continuous queries over RDF streams, and integration with both background data and streams. In C-SPARQL, a time window always selects the most recent items from a stream, and can be either count-based (a fixed number of items) or time-based (a variable number of items occurring in a fixed time interval).

STARQL (39) proposes a framework to access and query heterogeneous sensor data through ontologies. STARQL is structured as a two-layer framework, composed of an Ontology Language, to model the data and its schema, and an Embedded Constraint Language, to compose the queries. STARQL offers window operators, clauses to express event matching and a layer to integrate static and streaming data. STARQL uses a sequence of time-annotated ontologies to make inference taking into account the temporal annotations of the streaming data.

ETALIS (Event TrAnsaction Logic Inference System) (14) is a CEP-based stream reasoning engine. This query model processes streams where data items are annotated with two timestamps (i.e., time intervals). ETALIS defines two declarative rule-based languages, ETALIS Language for Events (ELE) and Event Processing SPARQL (EP-SPARQL). The former language is more expressive than the latter, even if it is less usable. A common point is that complex events are derived from simpler events using deductive prolog rules. EP-SPARQL supports backward temporal reasoning over RDFS, continuously evaluating the query over the whole stream received by the engine. Similar to other solutions, EP-SPARQL also provides windowing operators for isolating portions of the streams. However, EP-SPARQL is a solution that inherits the language constructs and processing model of CEP systems. EP-SPARQL focuses more on detection of RDF triples in a specific temporal order. Another important difference of EPSPARQL with respect to other languages such as C-SPARQL, is in the data model and consists in the way time is associated to RDF triples. While C-SPARQL associates one single timestamp to each triple.

IMaRS (Incremental Materialization for RDF Streams) is developed on top of C-SPARQL and it focuses on materialization (40). IMaRS utilizes an incremental reasoning approach and the specific data and processing models of C-SPARQL to compute the expiration time of streaming RDF triples based on the windows of deployed queries. By annotating each RDF triple with its expiration time and utilizing a hash table to index triples by their expiration time, IMaRS reduces the amount of computation that needs to be performed to update the results of reasoning. However, IMaRS relies on the strong assumption that the expiration time of each triple can be pre-computed, which limits its applicability.

Streaming Knowledge Bases is built on top of the TelegraphCQ DSMS (66) and provides reasoning using a subset of RDFS and OWL over streaming RDF triples (41). This approach allows to pre-compute and store inferences to reduce the overall computational effort, and consequently, the delay, during the evaluation of the queries.

Sparkwave (42) is a system designed for high performance and on-the-fly reasoning over RDF data streams. Sparkwave poses several limitations to the size of the background knowledge, which has to fit into the main memory of a single machine. Moreover, it operates with a pre-loaded RDF schema and provides limited reasoning functionality. Sparkwave implements a variant of the RETE algorithm (43), in which a preprocessing phase is used to materialize derived information before performing pattern matching. The portions of data considered during the processing are isolated through traditional windowing mechanisms, similar to those used by DSMSs and C-SPARQL.

INSTANS, The Incremental eNgine for STANding Sparql (44), takes a different perspective on RDF Stream Processing. Users model their task as multiple interconnected SPARQL 1.1 queries and rules. Next, INSTANS performs continuous evaluation of incoming RDF data against the compiled set of queries, storing intermediate results into a Rete-like structure. When all the conditions are matched, the result is instantly supplied. In this sense, INSTANS does not require continuous extensions to RDF or SPARQL.

The Streaming Linked Data (SLD) framework (45) wraps the C-SPARQL engine and it adds new features. SLD offers a set of adapters that transcode relational data streams in streams of RDF graphs (e.g. a stream of micro-posts as an RDF stream using the SIOC vocabulary (46), or a stream of weather sensor observation using the Semantic Sensor Network vocabulary (47)), a publish-subscribe bus to internally and externally exchange RDF streams (following the Streaming Linked Data Format (48)), facilities to record and replay RDF streams, and extendible layer to plug components that decorate RDF streams (e.g. adding sentiment annotations to micro-posts).

3.1 Summary

Stream reasoning enables associating reasoning tasks to time windows describing data validity and therefore producing time-varying inferences. Stream reasoning introduces new query and reasoning models, based on time model and continuous queries, enable on-the-fly processing of streaming data for IoT. We notice that stream processing technologies for IoT are still in the beginning. Research on stream reasoning has mainly focused on query processing.

Most of the systems proposed for Stream Reasoning are based on the concepts that we saw in the previous chapter, Semantic Web with the use of RDF, OWL and SPARQL together with CEP. In addition, these engines usually provide a SPARQL-based processing language, making some extensions to enable continuous processing over data streams.

However, the current developed solutions do not fulfill requirements of IoT. Querying, reasoning and handling uncertainties on big IoT data streams is still a challenge. Which data structures to adopt, and how to better exploit the limited size of main memory, how to reduce the expensive communication between reasoning nodes are challenges which remain unsolved.

4
Stream Reasoning Service

In this chapter, we present the architecture and implementation of the Stream Reasoning Service. This approach involves the use of the M-Hub, where CEP rules can be written and that will be executed in the Hub-serving devices, producing RDF triples. As well as C-SPARQL, which is responsible for executing continuous queries on RDF streams.

4.1
Conceptual Architecture

The architecture and its implementation must be able to handle two functional requirements, 1) the discovery of M-OBJs that are placed in the environment, 2) generate a real-time higher level contextualized information based on the data collected from the environment in which the M-OBJs are. Also, the implemented service must cope with some non-functional requirements: 1) Scaling: The service must have a high throughput of incoming data from sensors and to the need to link streaming data with large static knowledge bases; 2) Continuous Processing: Since IoT data is continuously generated, there must be a continuous evaluation of this data; 3) Real-Time: IoT systems must have a rapid reaction to the environment changes. To achieve these points, the service's architecture will define two levels of transformation, where the first level transforms the event/stream into a stream of RDF triples (fact units) using CEP. The second level takes these facts, and runs a continuous query in a stream reasoning system that can deduce new facts. The architecture's overview is illustrated in Figure 4.1.

We chose C-SPARQL for the continuous query module because it supports the processing of RDF triples streams in a SPARQL-like language, includes the possibility to merge static A-Box and dynamic RDF data and supports integration with ontologies described in RDF, RDFS or OWL. Moreover, it is open source, easily extendable and well documented. Its source code can be downloaded at Github [1].

Initially, the first level of processing receives a) an unique identifier (UUID) of the smart Thing's sensor and b) the related raw data that reveals

[1]https://github.com/streamreasoning/CSPARQL-engine

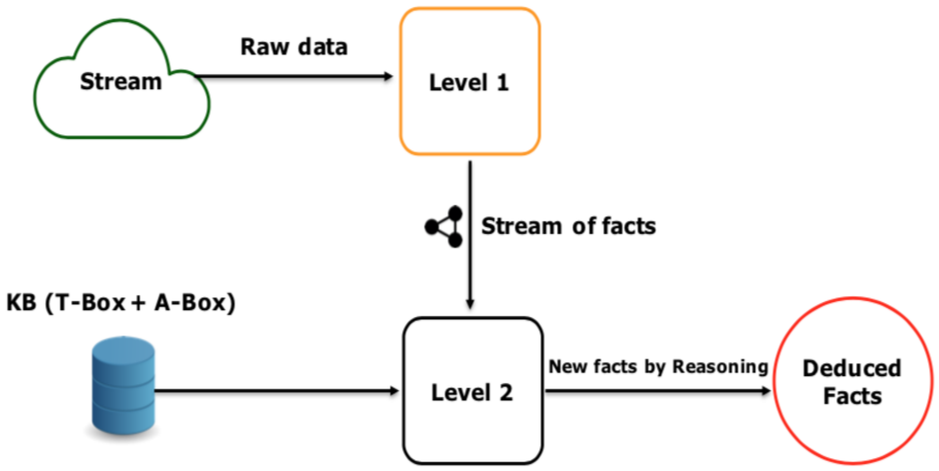

Figure 4.1: Stream Reasoning Service Conceptual Architecture

if the thing is subject to some action, experiencing a state change or any other transformation. For example, if the Thing is moving, the CEP engine will receive the UUID and the ever changing values of geographic coordinates (latitude, longitude). Otherwise, when monitoring temperature sensors, it will receive the UUID of the sensor and the temperature value. Then, by matching a sequence of data against some pattern in a pre-defined time window, the CEP layer is able to produce a semantically annotated stream of triples (facts) like <UUID, has-type, HighTemperature>.

The second level of processing takes as input the facts generated by the first level and runs a continuous stream reasoning engine (e.g. C-SPARQL). The engine uses a Knowledge Base (KB) which provides a background knowledge (conceptual and assertional, T-Box and A-Box of an ontology) about the events and other resources of the application domain, such as entities, relations, states. This means that events can be detected based on reasoning on their type hierarchy and their relationship to other objects in the application domain. For example, we have the fact that the temperature sensor with UUID(x) measured a high temperature value and, as we mentioned before, the first level of processing generates the triple <UUID, has-type, HighTemperature>. Adding the background knowledge that the sensor is connected – i.e. being worn by a person (Bill) and this person is currently a patient, and, by doing a ontology reasoning, leads to the conclusion that Bill has high temperature. Also, we can extend this by adding a rule that high temperature is a sub class of fever, so the reasoner could infer that the patient Bill has fever.

One of the benefits of using background knowledge with complex event processing is that we can have a higher expressiveness than if just using

CEP. Expressiveness means that an event processing system can now precisely express more complex events or states of entities, which are only revealed by combining the witnessed events with the background knowledge, the knowledge of the actors, the environment and the system. Returning to our example, adding another relationship to the KB that the temperature sensor belongs to a room numbered 123 and remembering that the sensor is being worn by Bill, it would be possible to precisely infer and inform a nurse or a doctor that patient Bill in room 123 has fever.

4.2 Implementation

In this section we describe the Stream Reasoning Service for the IoT. The service consists of three parts, 1) CEP layer provided by the MEPA Service in M-Hub, 2) SDDL Core Application (SCA), which is the backstage service implemented and hosted in a SDDL Core node, responsible for integrating the two processing layers and 3) A reasoning module, which makes use of C-SPARQL.

4.2.1 Complex Event Processing using M-Hub

The first level, where we have the transformation of raw sensor data by the CEP rules, can be achieved with the use of the M-Hub, since the M-Hub is both the direct acceptor of sensor data from (embedded or Bluetooth) sensors, and the element capable of making CEP computations on the stream of raw sensor data.

Once the M-Hub is connected with a gateway, it will start collecting information from the M-OBJs. Such information is in JSON[2] format and is of three types: LocationData that samples the current location of the M-Hub, SensorData which is the raw sensor values obtained from the M-OBJs as well the EventData, which describes events obtained from processing over the sensor data. Some examples are illustrated in Listing 4.1.

```
{
  "tag": "LocationData",
  "uuid": "b06de58d-6a20-44f9-8cd4-83f074c2edd6", // M-Hub UUID
  "latitude": -22.98137128,
  "longitude": -43.23421961,
  "battery": 50,
  "charging": false,
  "timestamp": 1442169467
```

[2]https://www.json.org/

```
}

{
  "tag": "SensorData",
  "uuid": "b06de58d-6a20-44f9-8cd4-83f074c2edd6",   // M-Hub UUID
  "source": "00000000-0000-0000-0001-bc6a29aecef5", // M-OBJ UUID
  "action": "found|connected|disconnected|read",
  "signal": -48,
  "sensor_name": "Temperature",
  "sensor_value": [80.0, 83.0],
  "latitude": -22.98137128,
  "longitude": -43.23421961,
  "timestamp": 1442169467
}

{
  "tag": "EventData",
  "uuid": "b06de58d-6a20-44f9-8cd4-83f074c2edd6", // M-Hub UUID
  "label": "AVGTemp",
  "data": {"value": 80.0},
  "latitude": -22.98137128,
  "longitude": -43.23421961,
  "timestamp": 1442169467
}
```

Listing 4.1: M-Hub messages

To obtain the data from the applied CEP rules, the SCA must send a MEPAQuery to the M-Hub. Listing 4.2 shows the JSON message that SCA sends to M-Hub.

```
{
 "MEPAQuery": {
  "type":"add|remove|start|stop|clear|get",
  "label":"AVGTemp",
  "object":"rule|event",
  "rule":"SELECT avg(sensorValue[1]) as value FROM
      SensorData(sensorName='Temperature')
      .win:time_batch(10 sec)",
  "target":"local|global"
 }
}
```

Listing 4.2: MEPAQuery

The *rule* field in the JSON message from Listing 4.2 represents the CEP rule that the M-Hub will apply to the sensors' data. The rule indicates that, for a time window of 10 seconds, it should compute the average temperature

```
{
  "tag": "EventData",
  "uuid": "b06de58d-6a20-44f9-8cd4-83f074c2edd6", // M-Hub UUID
  "label": "AVGTemp",
  "data": {"value": 80.0},
  "latitude": -22.98137128,
  "longitude": -43.23421961,
  "timestamp": 1442169467
}
```

Listing 4.3: EventData

readings. Listing 4.3 shows the M-Hub results of matching the rule pattern, shown in Listing 4.2, with the stream of sensor data.

Figure 4.2 illustrates the flow of messages in our first processing level, considering only the M-Hub and the SCA integration. In message {1}, SCA sends the MEPAQuery. Message {2} is the sensors' raw data that M-Hub receives. In {3} we have the sensors' data with the applied CEP rules {1} that are sent back to the SCA, that is listening for M-Hub messages. Once the Message 3 is received by the SCA, it will be transformed into a RDF triple.

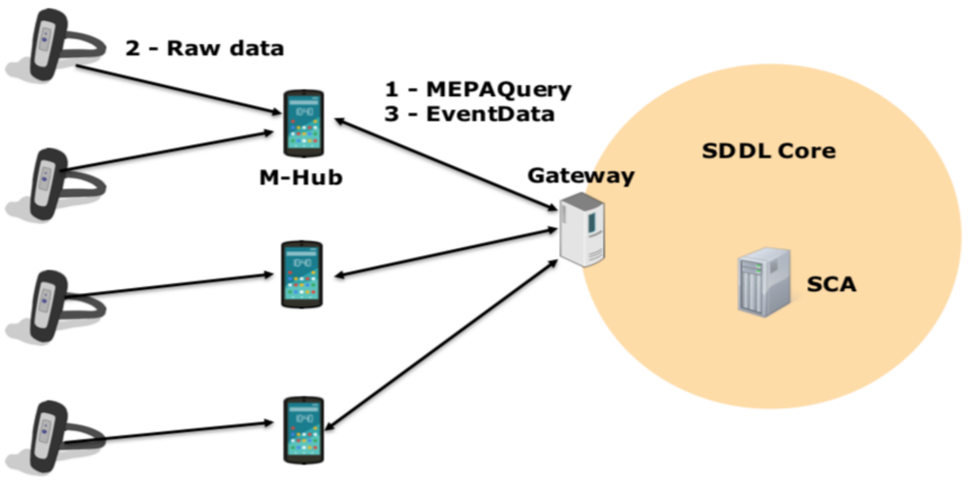

Figure 4.2: M-Hub and SCA integration

The RDF triple is related to the data coming from the EventData response and the underlying ontology. Let's consider an example that we are monitoring the temperature of a room and this room has a temperature sensor, which is connected with M-Hub. To establish the relation between each M-Hub and the room, we will use an external service, which we will call Relations DB. It's important to point out that the service's developer must know the ontology in order to translate the EventData message to a RDF triple. So, from the *uuid* key in the EventData message, which is originally the M-Hub's

UUID, we get the Room-UUID, representing our RDF subject. Also, we have an implicit mapping in the SCA that tells that temperature *80.0* means a very high temperature, resulting in the *VeryHigh* RDF object. Finally, we can map that *AVGTemp* represents the *has-temperature* predicate. As a consequence of this 3 points, we can generate a triple <Room-UUID, has-temperature, VeryHigh>. Figure 4.3 illustrates the process.

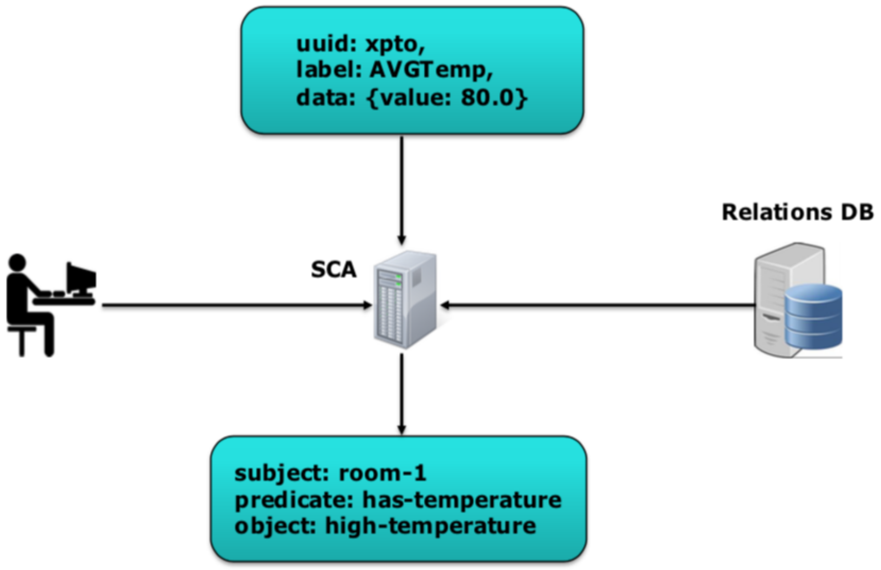

Figure 4.3: Converting EventData to RDF

4.2.2 Reasoning using C-SPARQL

As stated earlier, the second level of processing takes the RDF triples generated so far and performs a C-SPARQL query. The C-SPARQL reasoning engine has access to both A-Box and T-Box data, and can deduce new facts based on the ontology and the data. For external access to the Stream Reasoning Service, we have defined a Restful API which will be explained in sequence.

To be able to make inferences based on an ontology and their individuals, we first need to define an A-Box, a T-Box and a C-SPARQL query in the Reasoning Service. The Stream Reasoning engine (C-SPARQL) is responsible for integrating these three points. In addition, in order to send the reasoning result to the consumers (e.g. users of a dashboard, mobile users or an actuator device), we need to know their UUIDs. With these four fundamental pieces of information, we defined an endpoint called *register*, which receives the A-Box, T-Box, C-SPARQL query and the clients' UUIds, thus making it possible to search for inferred facts and send them to consumers. The registration phase should happen in the SCA start up. Upon initiating, the SCA calls the endpoint passing the required parameters. After the registration, the SCA can start sending RDF triples to the service. For this, we defined another endpoint, called *stream*, which receives as attributes a RDF triple, containing the *subject*, predicate and *object*. This phase occurs when the SCA receives the EventData message from the M-Hub and transforms it into a RDF triple. After this transformation, the endpoint is continuously called and the triples are inserted into the Reasoning Service. Figure 4.4 illustrates the process.

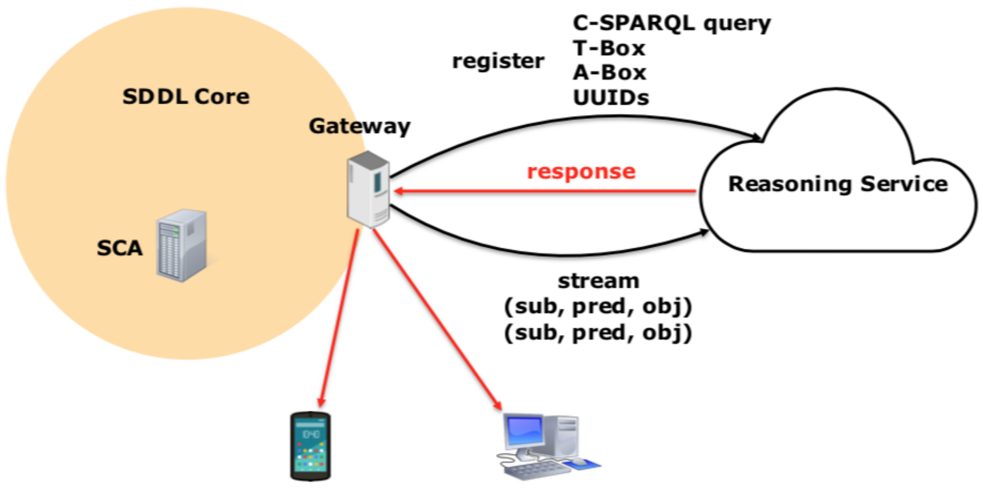

Figure 4.4: Reasoning Service API workflow

Returning to the example of the previous section where the CEP service generated the triple with the format <Room-UUID, has-temperature, Very-High>, assuming the following T-Box described in Listing 4.4 and the A-Box in Listing 4.5.

```
<!-- http://www.streamreasoning.com/OnFire -->

    <owl:Class rdf:about="http://www.streamreasoning.com/OnFire">
        <owl:equivalentClass>
            <owl:Class>
                <owl:intersectionOf rdf:parseType="Collection">
                    <rdf:Description rdf:about="http://www.
    streamreasoning.com/Room"/>
                    <owl:Restriction>
                        <owl:onProperty rdf:resource="http://www.
    streamreasoning.com/has-temperature"/>
                        <owl:hasValue rdf:resource="http://www.
    streamreasoning.com/#VeryHigh"/>
                    </owl:Restriction>
                </owl:intersectionOf>
            </owl:Class>
        </owl:equivalentClass>
    </owl:Class>

    <!-- http://www.streamreasoning.com/Room -->

    <owl:Class rdf:about="http://www.streamreasoning.com/Room"/>

    <!-- http://www.streamreasoning.com/Temperature -->

    <owl:Class rdf:about="http://www.streamreasoning.com/
    Temperature"/>
```

Listing 4.4: T-Box definition

```
<!-- http://www.streamreasoning.com/#VeryHigh -->

    <owl:NamedIndividual rdf:about="http://www.streamreasoning.com
    /#VeryHigh">
        <rdf:type rdf:resource="http://www.streamreasoning.com/
    Temperature"/>
    </owl:NamedIndividual>

    <!-- http://www.streamreasoning.com/#room-1 -->

    <owl:NamedIndividual rdf:about="http://www.streamreasoning.com
    /#room-1">
        <rdf:type rdf:resource="http://www.streamreasoning.com/
    Room"/>
        <has-temperature rdf:resource="http://www.streamreasoning.
    com/#VeryHigh"/>
    </owl:NamedIndividual>
</rdf:RDF>
```

Listing 4.5: A-Box definition

The T-Box, Listing 4.4, describes two classes, *Room* and *Temperature*. The individual for *Room* is *room-1* and for *Temperature* is *VeryHigh*, both are defined in the A-Box, Listing 4.5. Additionally we have the *OnFire* axiom, which is derived when there is a triple with the <Room-UUID, has-temperature, VeryHigh> format. This axiom represents a room on fire. Using C-SPARQL, we can assemble a query that repeatedly searchs for rooms that may be on fire. The query described in Listing 4.6.

The REGISTER QUERY clause on line 1 registers the continuous query named "onFireRoom" at the C-SPARQL engine. The query considers a sliding

```
1 REGISTER QUERY onFireRoom AS
2 PREFIX :<http://www.streamreasoning.com/>
3 PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
4 SELECT ?room
5 FROM STREAM <http://streamreasoning.com/streams/> [RANGE 1s STEP 1
     s]
6 FROM <http://streamreasoning.org/data>
7 WHERE {
8      ?room a :OnFire
9 };
```

Listing 4.6: C-SPARQL Query

window of 1 second that slides 1 second (line 5) and receives the RDF stream (clause FROM STREAM at line 5). The clause FROM at line 6 opens the RDF graph containing the static data (A-Box), which considers the individuals. The line 8 matches the room which is a *OnFire* room. Finally, line 4 constructs the RDF triples that are streamed out for down stream analysis.

Let's assume that the first level (CEP) continuously generates the triple <room-1, has-temperature, VeryHigh> and this triple is inserted by the endpoint *stream* into the continuous query engine. The query, described above, is executed in a time window of 1 second, which means that we consider only the last 1 second streamed data. So, every 1 second the reasoning engine takes a snapshot of the window, combines the window data with the static A-Box and the T-Box, and performs the reasoning process. When searching for rooms that may be on fire (e.g. the room has VeryHigh temperature), due to the streamed data, the A-Box and the T-Box, the query must generate the result *room-1*, since the SELECT clause is projecting the room that may be on fire.

4.3 Summary

In this chapter we have described an architecture for real-time stream processing using Stream Reasoning techniques in the scope of IoT, followed by its implementation.

As we have seen, the architecture consists of two processing levels, where the first level is responsible for the data pre-processing using CEP, which is possible by the use of the M-Hub. At the second level, we have the Stream Reasoning service with a semantic model defined by the T-Box, the A-Box and the RDF stream from the first level. The Reasoning Service outputs a new stream of facts that are deduced. The SDDL Core Application (SCA) works as a controller, responsible for activating the two levels and working as a translator, where an event originating from the M-Hub is translated into a

RDF triple, based on the model defined by the ontology.

The proposed service implementation addresses the main points that were defined at the beginning of this chapter. The M-Hub is responsible for the discovery and processing of the M-OBJs. The SCA and the Stream Resoning service contextualizes the data by adding a semantic model to them. Subsequently, this same Stream Reasoning service is responsible for deducting new facts and sending them to the actuators. All this processing is done in real time, since the data is continuously generated and treated by the two levels of processing.

5
A fire detection application

Fire detection is among the various areas that can benefit from interconnecting and monitoring sensors in a building. We have various kinds of fire detectors and suppression systems to help prevent against fire. The IoT is helping these products become more intelligent and connected. With IoT, now safety alerts can be sent to hundreds of people fast and effectively. Several leading fire safety companies, have already launched IoT-enabled fire detectors.

The most popular connected smoke detector on market is offered by Nest Labs[1], the leading smart home automation products. The company, now a part of Alphabet Inc.[2], offers Nest Protect smoke and carbon monoxide detector. The Nest Protect detectors are able to communicate with the Nest thermostat and can shut off the furnace in the event of a fire or carbon monoxide. The detectors can be accessed anywhere using mobile apps. In the event of an alarm, the detectors sound a local alarm as well as send notifications on the mobile phone.

Sensors have been widely used for fire detection (30, 31, 32, 33). Silva et al. (30) proposed a work for fire detection in mines by using wireless sensor networks called WMSS. For determining the hazardous factor in the mines, they used gas sensors and designed a wireless sensor network which collects and analyses the gas level in mines. The work proposed in (31) used Zigbee-based wireless sensors for fire detection in forests. They used temperature sensors to establish the intensity of fire in a forest. They used a CC2430 chip in their hardware design for network nodes.

Similarly, Buratti et al. (32) also designed a framework for forest fire detection. In their work, they used a model for fire detection using different clustering schemes and communication protocols. They performed the simulation for validation and evaluation of their work. W. Tan et al. (33) implemented a work for forest fire detection. They used multi-sensor and wireless IP cameras to avoid false alarms. Their system also connected to the internet via gateways for uploading the data to the cloud.

In subsequent sections, we describe a hypothetical application for fire

[1]https://nest.com/ca/
[2]https://abc.xyz/

detection, based on the temperature and humidity of the environment. The service developed and described in the previous section was used. As M-OBJs, we use the *SensorTags* from Texas Instruments [3]. These sensors have BLE interface and can be discovered using M-Hub. Figure 5.1 shows the M-Hub with 4 *SensorTags*, each with 6 sensors (temperature, accelerometer, humidity, ...). The *SensorTags* have a range up to 50m, then for the M-Hub to be able to discover and connect to the tags, it needs to be within this distance. The M-Hub must be an Android smartphone or tablet, since its implementation is only available for these devices.

This kind of application can be applied to a smart manufacturing or a smart home for example. We expect, with the modeling that will be described below, to achieve a real-time system that responds to an environmental stimulus, e.g. high temperature, in seconds, thus generating an fast alert for some kind of automation or actuator.

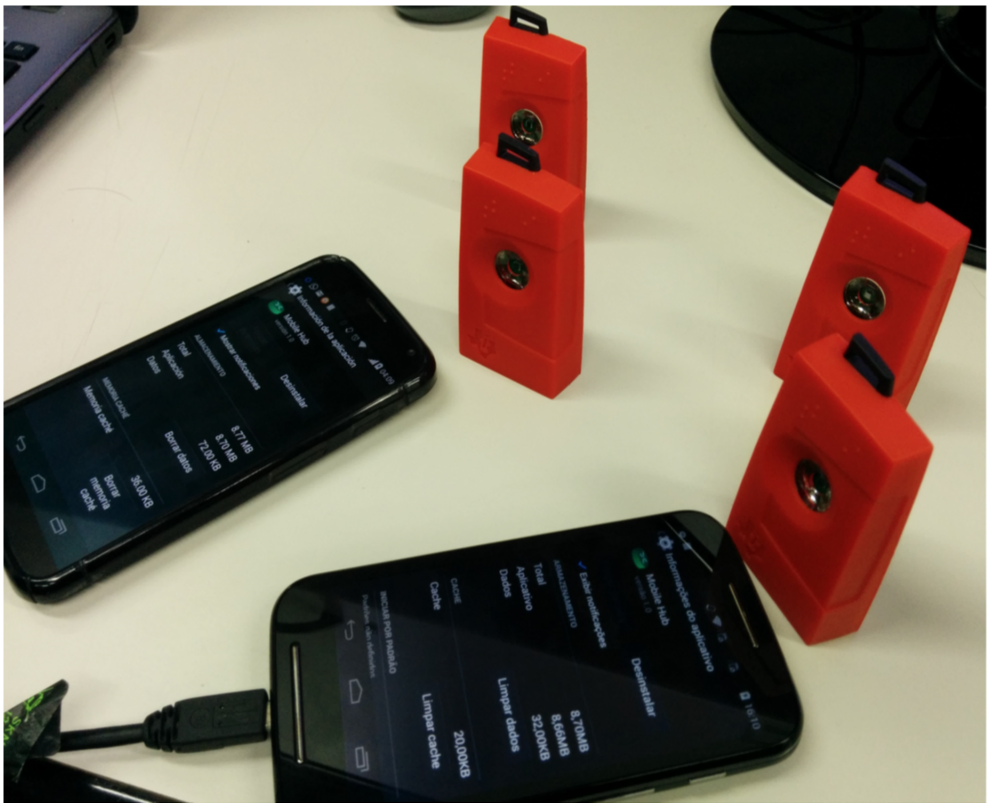

Figure 5.1: *SensorTags* and M-Hubs

The CEP query used for the first processing level (M-Hub) is described in Listing 5.1. The query gets the average temperature data collected by the M-Hub from the M-OBJs in a 10 seconds time window. As we saw in Chapter 4, the M-Hub's response brings the M-Hub's UUID, in JSON's *uuid*

[3]http://www.ti.com/

```
SELECT avg(temperature.sensorValue[0]) as avgTemperature
FROM SensorData(sensorName='Temperature').win:time (10 sec) as
    temperature
```

Listing 5.1: CEP Average temperature query

property. To get the UUID of the related room, let's assume there is a database service that gives this information. Also, the *predicate* will be filled according to the value receive and the type of CEP rule (AVGTemp). If the value is greater than 80 Celsius, the triple <Room-UUID, has-very-hot-sensor, M-Hub-UUID> is generated by the SCA and sent to the *stream* endpoint in the Reasoning Service. This triple format serves as input for all the scenarios that are described in this chapter.

An important note is that the ontology writer, since he is not necessarily a systems developer, does not need to know implementation details, such as using ContextNet, M-Hub, or C-SPARQL. Because of this, in modeling the ontology, we can consider that a room has sensors, not M-Hubs, since the second case is a specific detail of the implementation. So, even if our actual implementation we have a room related to one or more M-Hubs, we will consider that the M-Hub is represented by the *Sensor* class of the ontology.

To better understand our use case, let us consider the following characteristics:

1. Each SensorTag can detect both temperature and humidity
2. Each Room has 3 Sensors (M-Hubs)
3. Each Hall has 10 Rooms
4. Each Floor has 1 Hall

5.1 Scenario 1: Fire detection in a room

The first, and simplest, scenario is described by an ontology where we have two classes, *Room* and *Sensor*, that are defined in the T-Box. In addition, we have the *has-very-hot-sensor* relationship, linking these two entities. The A-Box for this scenario describes the relations between the rooms and sensors. Figure 5.2 illustrates the ontology.

Additionally, we define an axiom that represents a room in danger. This axiom, in OWL, is specified by the *DangerousRoom* class. Below we have the axiom described in Protégé DL Query [4] format:

[4]https://protegewiki.stanford.edu/wiki/DLQueryTab

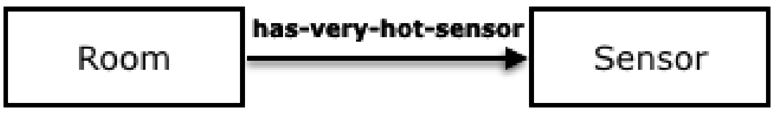

Figure 5.2: First scenario ontology

```
REGISTER QUERY dangerousRoom AS
PREFIX :<http://www.streamreasoning.com/>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
SELECT ?room
FROM STREAM <http://streamreasoning.com/streams/> [RANGE 1s STEP 1
    s]
FROM <http://streamreasoning.org/data>
WHERE {
    ?room a :DangerousRoom
};
```

Listing 5.2: DangerousRoom C-SPARQL query

– *Room* **and** *has-very-hot-sensor* **some** *Sensor*

The axiom indicates that if there is a triple with the <Room-UUID, has-very-hot-sensor, M-Hub-UUID> format then the *DangerousRoom* class is inferred for the related Room-UUID. The **some** keyword in DL means *at least one*, so the DL query indicates that ***A Room that has at least one very hot Sensor*** is a *DangerousRoom*.

As we saw earlier, in order to find the *DangerousRoom*, a C-SPARQL, described in Listing 5.2, must be defined in the Reasoning Service. The query and, consequently, the Reasoning Service, outputs the Room's UUID that might be in danger. Figure 5.3 shows the ontology mapping with the inferenced class.

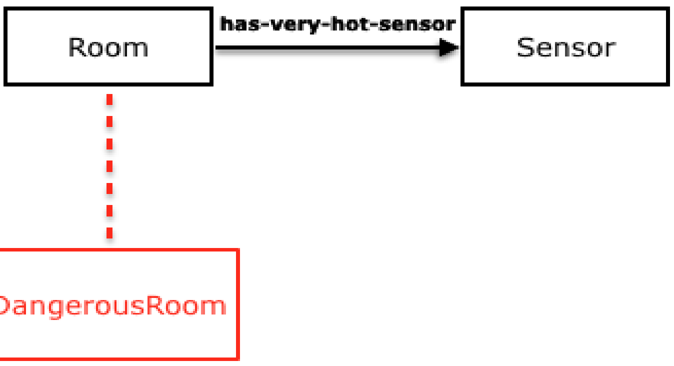

Figure 5.3: First scenario ontology with DangerousRoom axiom

5.2 Scenario 2: Fire detection in a building hall

The second scenario can be seen as an extension of the first, where the concept of hall is added, which is defined by the class of the same name, *Hall*. To relate the *Hall* and *Room* classes, the *has-hall* property was created. For this scenario, the A-Box, besides having the relations of the first scenario, adds the relations inherent to the *Hall* and *Room*.

In addition to the concept of *DangerousRoom*, we also add an axiom representing a hall in danger, which is named *DangerousHall*. Below we have its definition, where we first present the simplified version, using the class inferred in the first scenario and, secondly, the expanded version, where the axiom that defines *DangerousHall* encompasses the *DangerousRoom* axiom. The axiom means that ***A Hall that has at least one Room that has at least one very hot Sensor*** is a *DangerousHall*. Figure 5.4 illustrates the updated ontology with the new additions.

– *Hall* **and** *has-room* **some** *DangerousRoom*

– *Hall* **and** *has-room* **some** (*Room* **and** *has-very-hot-sensor* **some** *Sensor*)

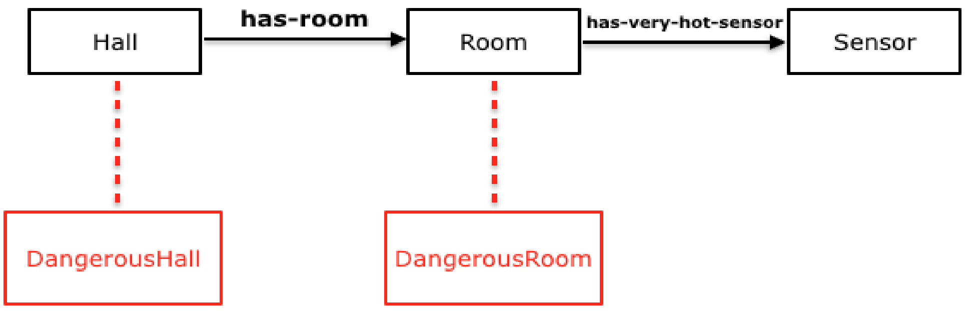

Figure 5.4: Second scenario ontology with DangerousHall axiom

In order to find the *DangerousHall* inference, we had to change the previous C-SPARQL query. The new one is described in Listing 5.3. Differently from the query in the first scenario, this one seeks for halls that might be in danger and outputs the Hall's UUID.

```
REGISTER QUERY dangerousHall AS
PREFIX :<http://www.streamreasoning.com/>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
SELECT ?hall
FROM STREAM <http://streamreasoning.com/streams/> [RANGE 1s STEP 1
    s]
FROM <http://streamreasoning.org/data>
WHERE {
    ?hall a :DangerousHall
};
```

Listing 5.3: DangerousHall C-SPARQL query

5.3 Scenario 3: Fire detection in a building floor

Extending the scenario further, we added the concept of Floor, represented by the class of the same name. To relate the *Floor* and *Hall* classes, we created and used the *has-floor* property. For this scenario, the A-Box adds the relations between floors and halls, in addition to the relations between halls, rooms and sensors.

We also add an axiom representing a floor in danger, called *DangerousFloor*. Below we have its definition, where we first present the simplified version, using the class inferred in the first scenario and, secondly, the expanded version, where the axiom that defines *DangerousFloor* encompasses the *DangerousHall* axiom. The axiom means that ***A Floor that has at least one Hall that has at least a Room that has at least one very hot Sensor*** is a *DangerousFloor*. Figure 5.5 illustrates the updated ontology with the new additions.

- *Floor* **and** *has-hall* **some** *DangerousHall*
- *Floor* **and** *has-hall* **some** (*Hall* **and** *has-room* **some** (*Room* **and** *has-very-hot-sensor* **some** *Sensor*))

In order to find the *DangerousFloor* inference, we defined a new C-SPARQL query, which is described in Listing 5.4. This one seeks for floors that might be in danger and outputs the Floor's UUID. Bellow we have the query used to find this inference.

5.4 Scenario 4: Fire detection by proximity

Proximity fire detection goes through all of the scenarios described in the previous sections. In this section we will present additions to the

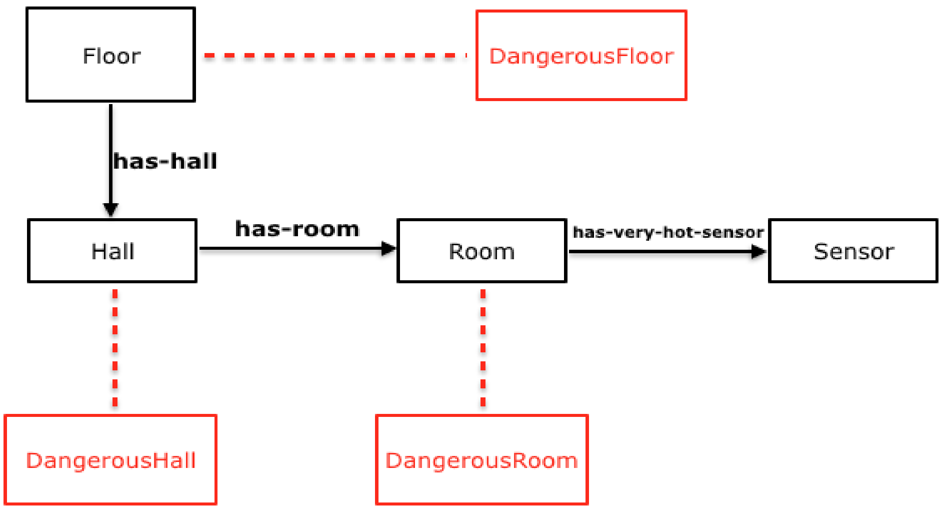

Figure 5.5: Third scenario ontology with DangerousFloor axiom

```
REGISTER QUERY dangerousFloor AS
PREFIX :<http://www.streamreasoning.com/>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
SELECT ?floor
FROM STREAM <http://streamreasoning.com/streams/> [RANGE 1s STEP 1
    s]
FROM <http://streamreasoning.org/data>
WHERE {
    ?floor a :DangerousFloor
};
```

Listing 5.4: DangerousFloor C-SPARQL query

ontology to make the inference from the temperature measurement in another environment, possible.

First we will describe how the detection is made considering only the *Room* entity. To add the notion of proximity, we define a new property in the ontology that relates a room to another, we call this property *is-near*. Recalling that for this scenario it was not necessary to make any alteration in the first level of processing, the CEP. The modifications were all done in the T-Box and A-Box. Figure 5.6 illustrates the updated ontology.

With these modifications, it is possible to infer a new class type, which is called the *AlertRoom*. An *AlertRoom* is defined by ***A Room that is near another Room which has a Sensor that is very hot***. As we present in the other scenarios, below we have two ways in which we can define this axiom. First, the simplest form, already encompassing the concept of *DangerousRoom* and, second, the form that we do not consider the axiom defined previously.

– *Room* **and** *is-near* **some** *DangerousRoom*

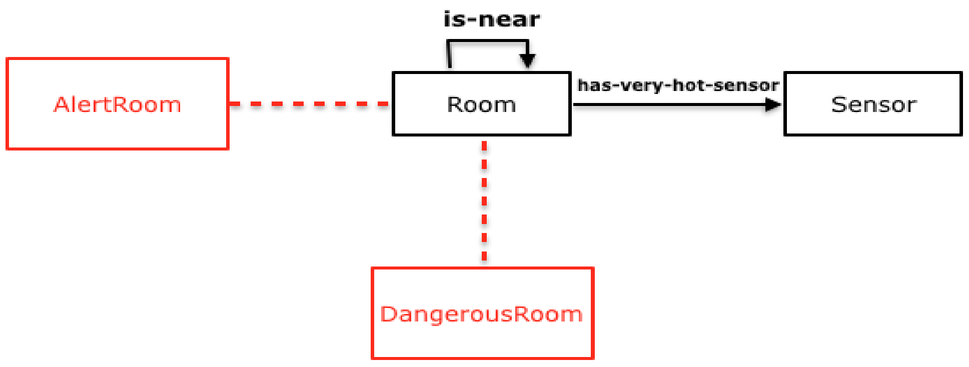

Figure 5.6: Adding room's proximity relation

```
REGISTER QUERY alertRoom AS
PREFIX :<http://www.streamreasoning.com/>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
SELECT ?room
FROM STREAM <http://streamreasoning.com/streams/> [RANGE 1s STEP 1
    s]
FROM <http://streamreasoning.org/data>
WHERE {
    ?room a :AlertRoom
};
```

Listing 5.5: AlertRoom C-SPARQL query

– *Room* **and** *is-near* **some** (*Room* **and** *has-very-hot-sensor* **some** *Sensor*)

To find the *AlertRoom* inference, we defined a continous query, which is described in Listing 5.5, that seeks for rooms that might be near to rooms that are in danger. The query outputs the Room's UUID.

To gradually increase the ontology's complexity, we apply the concept of fire detection by proximity to a *Hall*. For this scenario, we will use the same property that was defined before, *is-near*, but, in this case, the relation is between two halls.

Adding the concept of a hall that can be near to another hall, we can do the inference like we did in the previous scenario. Considering that a hall could be in danger and this hall is near to another one, we can say that the other hall is in alert. To define this type of inference, the *AlertHall* entity was created. An *AlertHall* means ***A Hall that is near to another Hall which has a Room that has a very hot Sensor***. As in the other scenarios, the two types of axioms that we can define are presented below, as well the Figure 5.7 illustrates the updated ontology.

– *Hall* **and** *is-near* **some** *DangerousHall*

```
REGISTER QUERY alertHall AS
PREFIX :<http://www.streamreasoning.com/>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
SELECT ?hall
FROM STREAM <http://streamreasoning.com/streams/> [RANGE 1s STEP 1
    s]
FROM <http://streamreasoning.org/data>
WHERE {
    ?hall a :AlertHall
};
```

Listing 5.6: AlertHall C-SPARQL query

– *Hall* **and** *is-near* **some** (*Hall* **and** *has-room* **some** (*Room* **and** *has-very-hot-sensor* **some** *Sensor*))

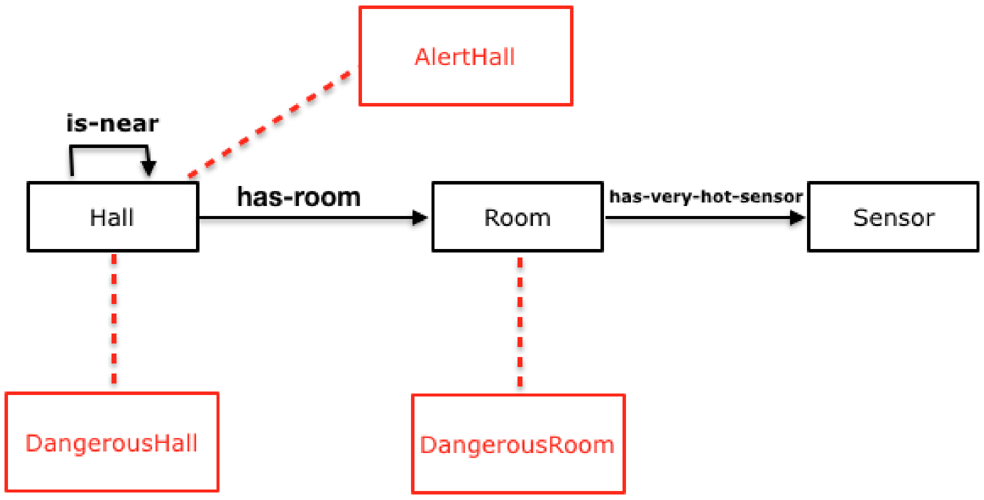

Figure 5.7: Adding hall's proximity relation

To find the *AlertHall* inference, we defined a continuous query, which is described in Listing 5.6, that seeks for halls that might be near to halls that are in danger. The query outputs the Hall's UUID.

The last scenario to be presented represents a Floor that may be on alert. In this scenario, we use the same property presented in all others that detect fire by proximity, the *is-near* property. However, in this case, this property relates two floors. The proximity fire detection can occur in such a way that ***An Alert Floor is a floor that is near a floor that has a hallway which in turn has a room that has a very hot Sensor***. The two types of axioms that we can define are presented bellow. Figure 5.8 illustrates the updated ontology.

– *Floor* **and** *is-near* **some** *DangerousFloor*

```
REGISTER QUERY alertFloor AS
PREFIX :<http://www.streamreasoning.com/>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
SELECT ?floor
FROM STREAM <http://streamreasoning.com/streams/> [RANGE 1s STEP 1
    s]
FROM <http://streamreasoning.org/data>
WHERE {
    ?floor a :AlertFloor
};
```

Listing 5.7: AlertFloor C-SPARQL query

– *Floor* **and** *is-near* **some** (*Floor* **and** *has-hall* **some** (*Hall* **and** *has-room* **some** (*Room* **and** *has-very-hot-sensor* **some** *Sensor*)))

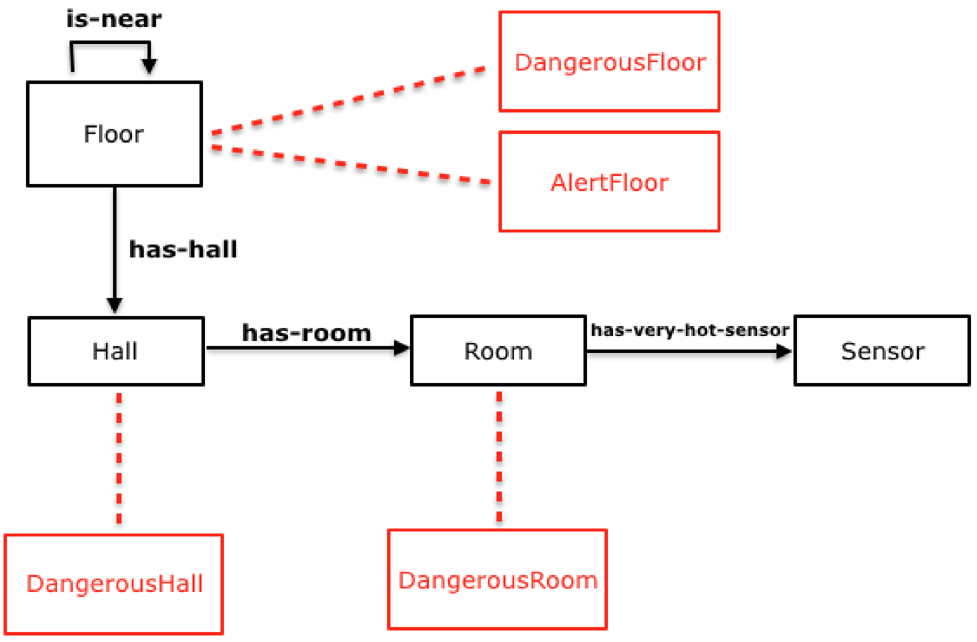

Figure 5.8: Adding floor's proximity relation

To find the *AlertFloor* inference, we defined a C-SPARQL query described in Listing 5.7, which seeks for floors that might be near to floors that are in danger. The query outputs the Floor's UUID.

5.5 Summary

In this chapter we proposed an application in the scope of IoT for fire detection. This application makes use of the architecture and implementation which were described in chapter 4.

By using the M-Hub together with the *SensorTags*, we were able to monitor the temperature measurements. With these measurements, we can infer if the environment has the temperature out of normality, thus generating a semantically annotated information for the Stream Reasoning Service. Enriching the environment information data with the ontology and the static relations (e.g. sensor x is placed in room y), we can infer new facts based on the streaming data, for example, if a room, hall or floor is on fire.

As we have seen, we can divide our ontology according to what is being monitored. We showed that we can gradually increase the entities presented in the ontology, so that in the first definition we have only the concept of rooms and sensors. In the last ontology, we defined concepts for monitoring, as well as rooms, halls and floors. In addition, we have shown that it is possible to monitor an environment that does not necessarily have a coupled sensor, which can be done with the proximity relationship.

6
Performance Analysis

To determine the service performance we focused on scalability, that is, on studying the performance of reasoning and data delivery when the amount of connected IoT nodes and the data volume sizes are varied. Semantic representations are known to have a significant effect on resource usage (36). Hence, different ontology models and methods to make inferences were considered as one of the most important feature to be tested. Heterogeneous and continuously provided distributed data is assumed as a general characteristic of IoT systems, thus, the experiments focus on reasoning in real-time with distributed data providers and data volumes in configurable distributed environment. To do this, we vary the size of the A-Box, increase and decrease the complexity of the ontology, and for some cases, we change the frequency with which the data is generated by the sensors.

For the analysis, we will use time as the measure of the service performance. The initial time will be the instant that the data is generated by the *SensorTag*. The final time will be counted as the instant of the Reasoning Service (second processing level) output. We will use the average of 3 measurements to get the service's response time, eliminating the outliers.

One of the key features of IoT systems is that this kind of application continuously produce new data and the data to be processed must always be the most recent. In our tests, due to the difficulty of simulating ambient conditions such as temperature and humidity, we chose to simulate sensor data that is processed by the M-Hub. With this, we can also simulate the rate in which the raw data is generated. For the tests, we set 1 second rate of new data and simulate the ambient conditions in exactly 1 specific room. For example, with a rate of 1 second, the triple <room-1, has-very-hot-sensor, sensor-1> is generated.

Our experimental setup, which includes the M-Hub simulator, the SCA and the Reasoning Service, are all distributed in the same network connected by a WiFi 802.11 router. The M-Hub simulator and the Reasoning Service uses a Core i7 processor with 16GB of RAM, hosted on a macOS 10.13. The SCA is hosted in a Ubuntu 16.04 virtual machine with Core i7 processor and 4 GB RAM. The experiments were developed using Java, in version 8.

6.1 A-Box Size

In this first performance test we will consider only the variation of the number of individuals (e.g. floors, halls, rooms and sensors) contained in the A-Box. Each experiment increments the A-Box size gradually, so that we can see the evolution of the response time. It is important to remember that each room has 3 sensors. Then, by varying the number of rooms, consequently the number of sensors also varies.

In the first experiment (Figure 6.1), which measures the scenario that looks for rooms that may be in danger, the individuals that were inserted in the A-Box refer to the number of rooms and sensors being observed.

In the second experiment (Figure 6.2), measures the scenario that looks for rooms that may be in alert. In this case, the base test must start at least with 2 rooms because the *is-near* relation is about connecting 2 rooms.

In the third experiment (Figure 6.3), we collapsed the Hall and Floors scenarios due to the case that a building can have exactly one floor with one hall. In this experiment, the x axis refers to the number for floors that we are observing. Remembering that a floor has 1 hall and a hall has 10 rooms. In this case, the experiment starts with 1 floor, 1 hall, 10 rooms and 30 sensors.

The fourth experiment (Figure 6.4) is an extension of the third experiment adding the *is-near* property. In this experiment, due to proximity, the test must start with the minimum of 2 floors.

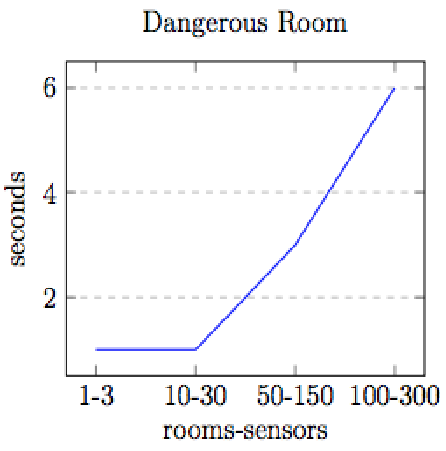

Figure 6.1: First experiment DangerousRoom inference performance

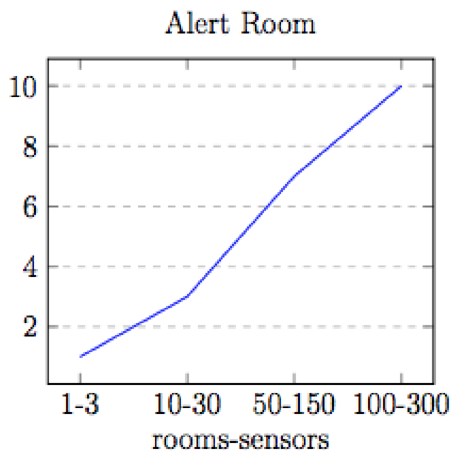

Figure 6.2: Second experiment AlertRoom inference performance

The experiments involving the change in A-Box size showed a similar result. It is important to note that a difference of up to 1 second for one experiment and another we can consider that time is equal. This happens because we work with a time window of 1 second in C-SPARQL, so there

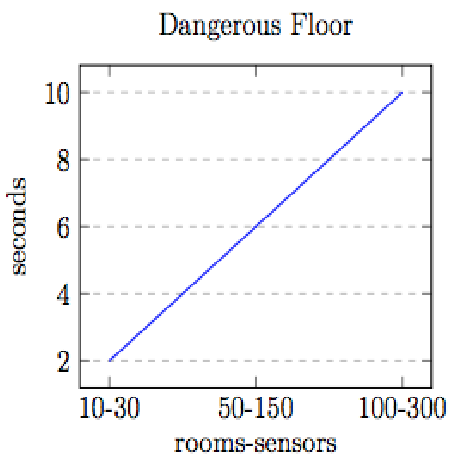

Figure 6.3: Third experiment DangerousFloor inference performance

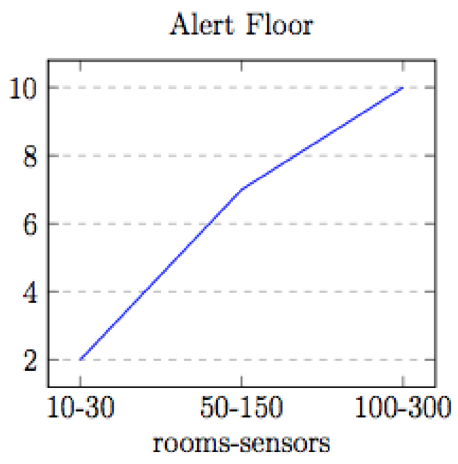

Figure 6.4: Fourth experiment AlertFloor inference performance

may be some delay that affects the comparison between experiments. Another important point is that all experiments had acceptable results for a real-time application. Applications of this type require response in seconds, which we obtained in all experiments. We can also observe that the result, comparing experiments 1 and 2, following 3 and 4, are practically the same. This leads us to believe that the addition of the concept of proximity did not influence the increase of complexity, and consequently of time, of the scenarios. One negative point is that for values greater than 400 individuals, the reasoning engine could not complete the processing, staying for more than 20 minutes in lock. This was due to an internal failure of C-SPARQL.

6.2 T-Box Complexity

In this section we will change the T-Box complexity in several ways. As in the previous section, we have divided in some experiments which have the same size variation of the A-Box. In the **fifth experiment** we will modify the ontology and simplify the first level of processing (CEP). In the **sixth experiment** we will keep the change in the CEP but we will increase even more the complexity. Finally, in the **seventh experiment**, we will add humidity data and compare the results with all the other experiments.

6.2.1 Sensor Subclass

In the fifth experiment we modified the first level of processing by removing the external service that retrieves the room-sensor relation. With this, the generated triple is also modified to the format <M-Hub-UUID,

rdf:type, VeryHotSensor>, which means that ***The sensor with the related UUID has the type VeryHotSensor***. *VeryHotSensor* is a specialization of *Sensor* and means that a Sensor can be a ***Very Hot Sensor***. With this small change in the RDF stream format, the base ontology must also undergo changes to achieve the same inference from the scenarios presented previously. The axioms *DangerousRoom* and *AlertRoom* have also been modified. The A-Box and the C-SPARQL query had no changes. Figure 6.5 illustrates the updated ontology and bellow we have the updated axioms.

– **DangerousRoom**: *Room* **and** *has-sensor* **some** *VeryHotSensor*

– **AlertRoom**: *Room* **and** *is-near* **some** *DangerousRoom*

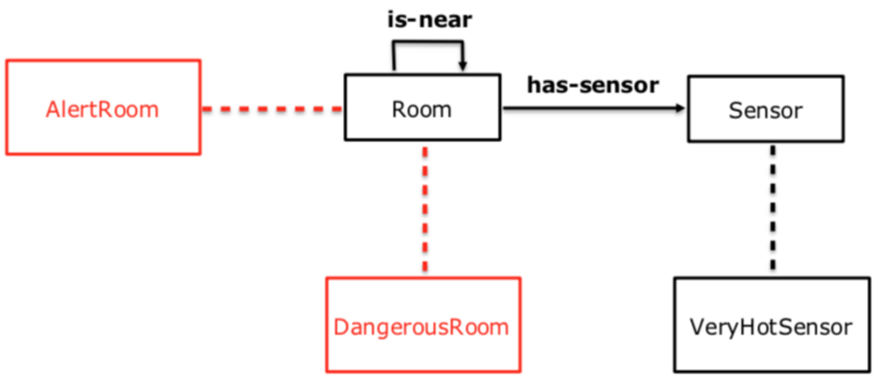

Figure 6.5: Fifth experiment - Adding VeryHotSensor subclass to the ontology

The changes were also applied in the *DangerousFloor* and *AlertFloor* axioms. Figure 6.6 illustrates the updated ontology and bellow we have the updated axioms.

– **DangerousFloor**: *Floor* **and** *has-hall* **some** (*Hall* **and** *has-room* **some** (*Room* **and** *has-sensor* **some** *VeryHotSensor*))

– **AlertFloor**: *Floor* **and** *is-near* **some** *DangerousFloor*

To find out if there has been any change in the performance of the service, we plotted the time in seconds represented by the y-axis and the variation of the A-Box size represented by the x-axis. Figures 6.7, 6.8, 6.9 and 6.10 represent the charts.

From the results, we see that the performance had few differences compared to the experiments presented in the previous section. This occurred because the ontology did not undergo major modifications, and consequently, the inference steps also did not have a significant change.

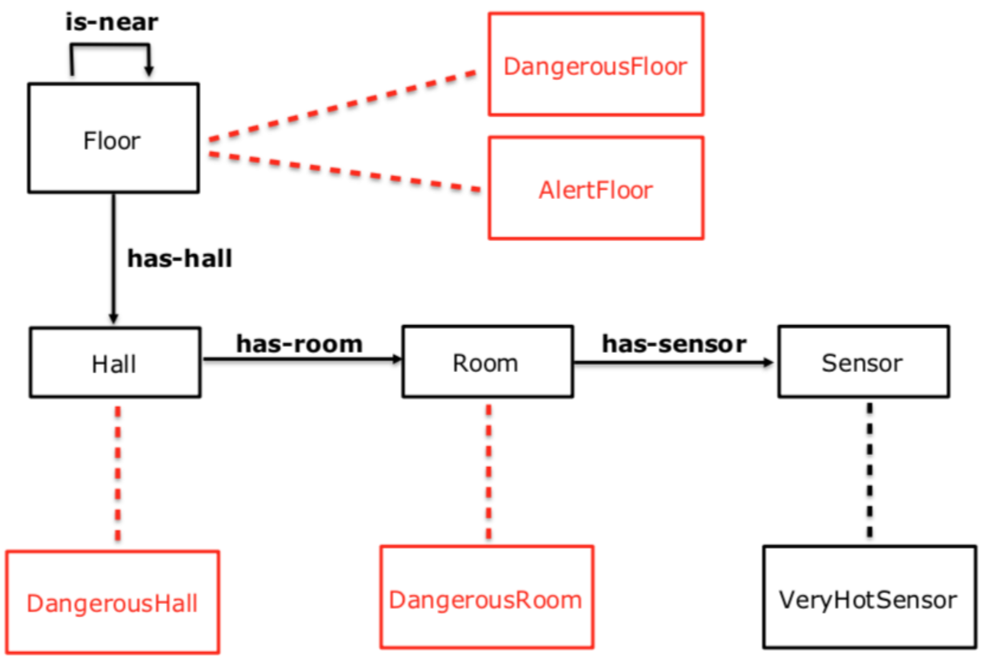

Figure 6.6: Fifth experiment - Adding VeryHotSensor subclass complete ontology

6.2.2 Adding Device Specialization

In the sixth experiment, we made a profound change in the ontology by adding more levels of verification. We add the concept of *Device*. A room can be connected to one or more devices.

The *Device* can have various types of expertise, such as *EnergyDevice* or *SensingDevice*. For this experiment we will consider only the *SensingDevice* specialization. A *SensingDevice* can observe a certain characteristic of interest, such as temperature or humidity. In case of temperature, it can be high, low or normal.

With these modifications, looking for, for example, a *DangerousRoom*, is a little more complicated. As described above, a *Room* is connected to one or more Devices and to be able to detect a high temperature, this Device must be a *SensingDevice* and also observe a Temperature which is high.

To fit the changes in the ontology, we need to change the RDF triple generated by the SCA. In this experiment, we will assume, as in the previous experiment, that there is no external service that does the room-sensor mapping. Based on the ontology, the RDF generated must be in the format <M-Hub-UUID, obverses, high-temperature>. Figure 6.11 shows the updated ontology and below we see the description of the updated axioms for *DangerousRoom* and *AlertRoom*.

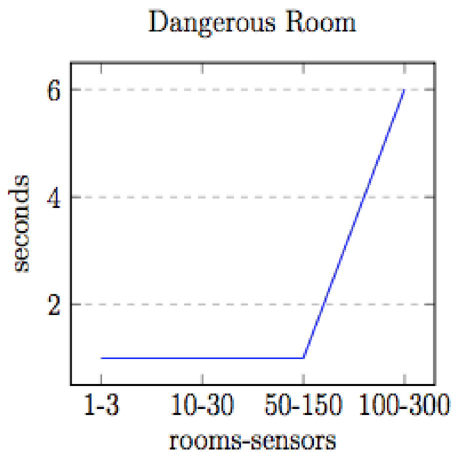

Figure 6.7: Fifth experiment DangerousRoom inference performance

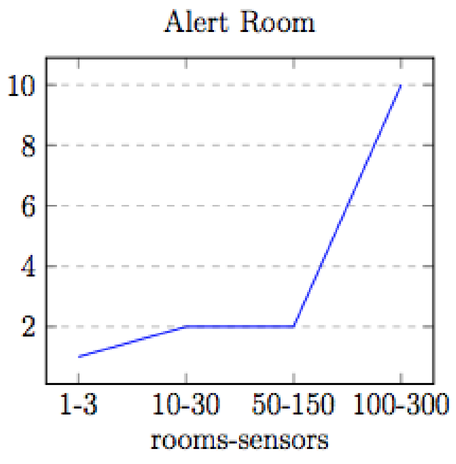

Figure 6.8: Fifth experiment AlertRoom inference performance

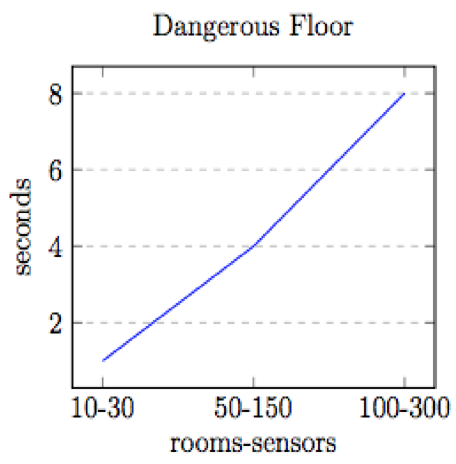

Figure 6.9: Fifth experiment DangerousFloor inference performance

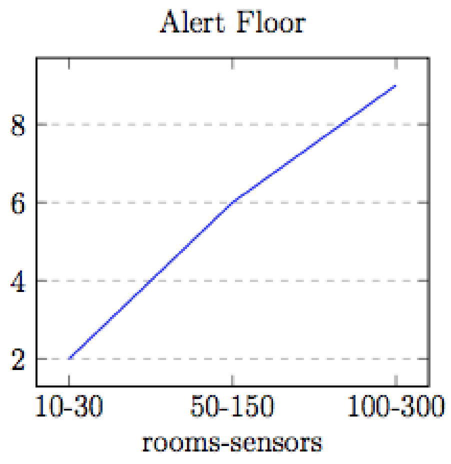

Figure 6.10: Fifth experiment AlertFloor inference performance

- **DangerousRoom**: *Room* **and** *has-device* **some** (*Device* **and** *observes* **value** *high-temperature*)
- **AlertRoom**: *Room* **and** *is-near* **some** *DangerousRoom*

It is important to point out that the axioms seem to be simple, but there are implicit verifications that are made, such as subclasses' instance checking, which in this case have the subclasses *SensingDevice* and *Temperature*.

As in the previous experiment, the *DangerousFloor* and *AlertFloor* were also modified to undergo the ontology's modification. Bellow we have the updated axioms and Figure 6.12 illustrates the updated ontology with this two more entities.

- **DangerousFloor**: *Floor* **and** *has-hall* **some** (*Hall* **and** *has-room* **some** (*Room* **and** *has-device* **some** (*Device* **and** *observes* **value** *high-temperature*)))

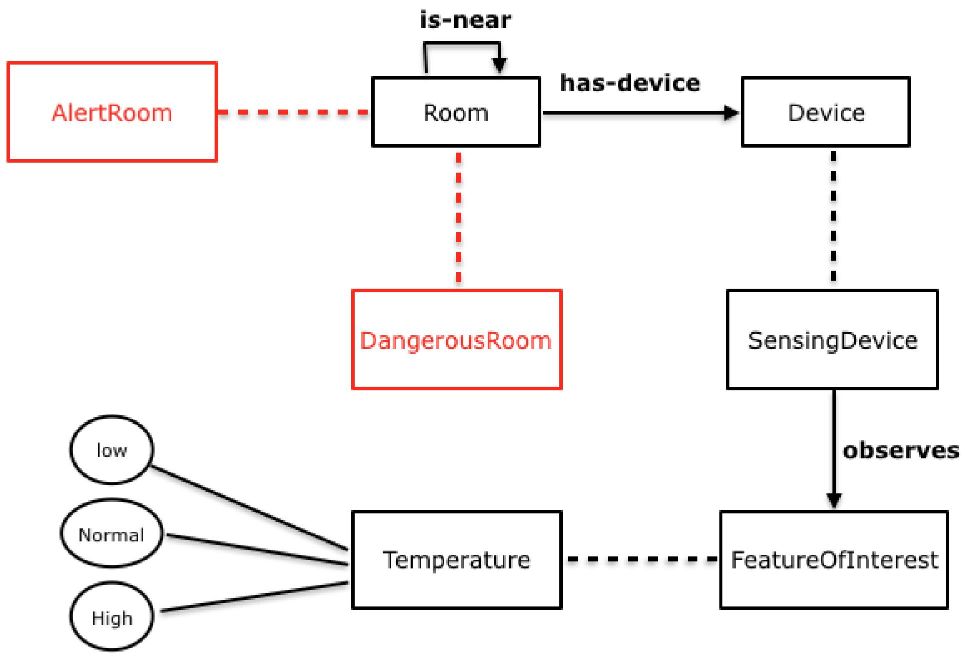

Figure 6.11: Sixth experiment - Adding device specialization to the ontology

Experiment	1	2	3	4	5.1	5.2	5.3	5.4	6.1	6.2	6.3	6.4
Time in Seconds	6	10	10	10	6	10	8	9	11	23	14	20

Table 6.1: Experiments comparison

– **AlertFloor**: *Room* **and** *is-near* **some** *DangerousRoom*

For this experiment we tried to compare, with the cases already tested, to the maximum individuals, e.g. 400. The comparison can be seen in Table 6.1.

The comparison of this experiment with the others shows that the addition of more complexity to the ontology has a great impact on the service's performance, even compromising the real-time analysis. We still consider that the size of this ontology is not very large, so a point to be pondered is whether it is worth adding a lot of complexity to the ontology in an unnecessary way because we have seen that for simpler ontology we get equal results with better performance.

6.2.3 Adding Humidity

In previous experiments we were only considering temperature. However, in this experiment we will add another data coming from the *SensorTag*, the humidity. In addition, we will compare it with all the other experiments already performed, in order to compare the performance with and without the humidity

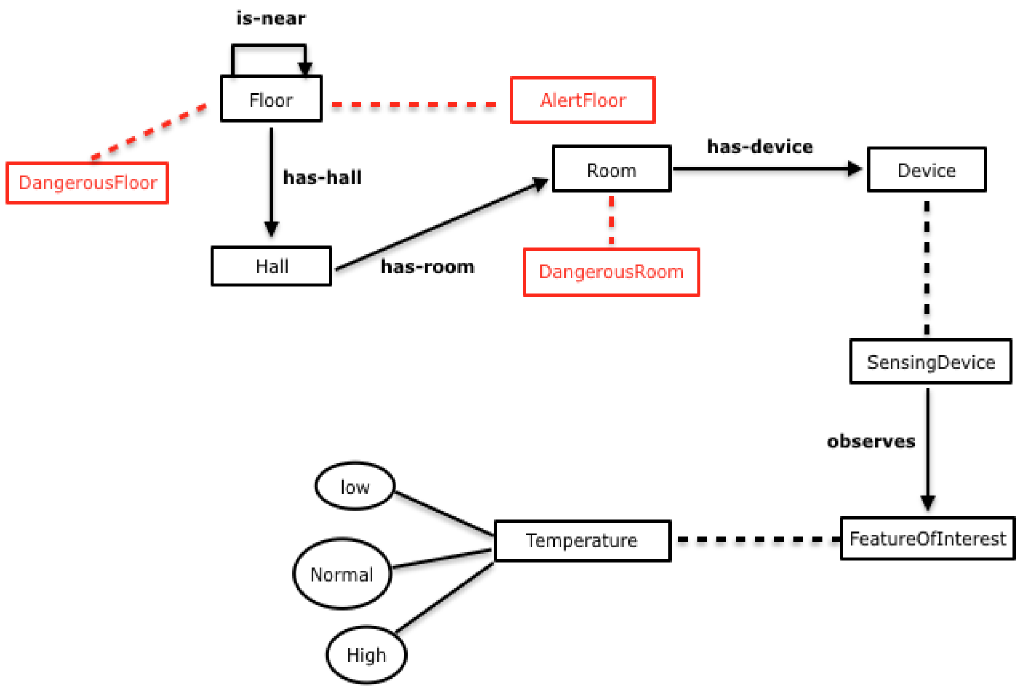

Figure 6.12: Sixth experiment - Adding device specialization complete ontology

```
SELECT avg(temperature.sensorValue[0]) as avgTemperature,
         avg(humidity.sensorValue[0]) as avgHumidity
FROM SensorData(sensorName='Temperature').win:time (10 sec) as
    temperature, SensorData(sensorName='Humidity').win:time (10 sec
    ) as humidity
WHERE temperature.source = humidity.source
```

Listing 6.1: CEP query considering temperature and humidity

data. It is worth mentioning that we will use humility below 19% as being considered low.

In order to obtain the sensors' humidity, it is necessary to make a small modification in the CEP query that is applied by the M-Hub, which is described in Listing 6.1. For optimization purposes, the CEP query matches both the temperature and humidity data in the same response, through a normal join. It is worth mentioning that the WHERE clause assures us that the two information comes from the same sensor. This query is used by all the experiments that will be described bellow.

Extending the **first to fourth experiments**, we added the notion of *very-dry-sensor* relation between *Room* and *Sensor* entities. With this addition, we must generate another triple that matches this kind of relation. So, besides the triple with the <Room-UUID, has-very-hot-sensor, M-Hub-UUID> format, we also generate the triple <Room-UUID, has-very-dry-sensor, M-

Hub-UUID>. Another import point is that, to preserve the simulated sensors data rate of 1 second, and, considering that we are sending two kinds of triple, we will sending with a rate of 500 milliseconds between each triple, thus maintaining the rate of 1 second considering the temperature's and humidity's triples. The A-Box and the C-SPARQL query remains the same. Figures 6.13, 6.14, 6.15 and 6.16 illustrates the performance comparison and bellow we have the updated *DangerousRoom* and *DangerousFloor* axioms adding the humidity concept:

– **DangerousRoom**: *Room* **and** (*has-very-hot-sensor* **some** *Sensor*) **and** (*has-very-dry-sensor* **some** *Sensor*)

– **DangerousFloor**: *Floor* **and** *has-hall* **some** (*Hall* **and** *has-room* **some** ((*Room* **and** *has-very-hot-sensor* **some** *Sensor*) **and** (*Room* **and** *has-very-dry-sensor* **some** *Sensor*)))

By extending the **fifth experiment**, we added the *VeryDrySensor* entity as a specialization of Sensor. With this modification, like in the previous experiment, we had to generate another RDF triple. So, besides the triple with the format <M-Hub-UUID, rdf:type, VeryHotSensor>, we also generate the triple <M-Hub-UUID, rdf:type, VeryDrySensor>. The A-Box and the C-SPARQL query remains the same. Figures 6.17, 6.18, 6.19 and 6.20 illustrates the performance comparison and bellow we have the updated *DangerousRoom* axiom adding the humidity concept:

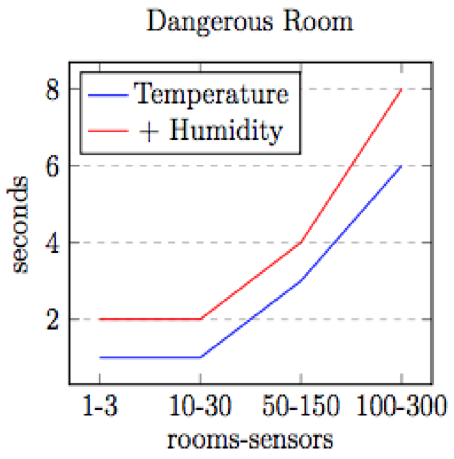

Figure 6.13: First experiment DangerousRoom inference performance adding humidity

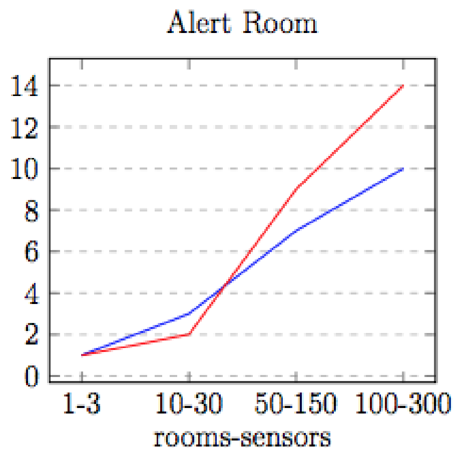

Figure 6.14: Second experiment AlertRoom inference performance adding humidity

Adding humidity to the **sixth experiment** means adding another specialization of the class *FeatureOfInterest*. This new specialization is called

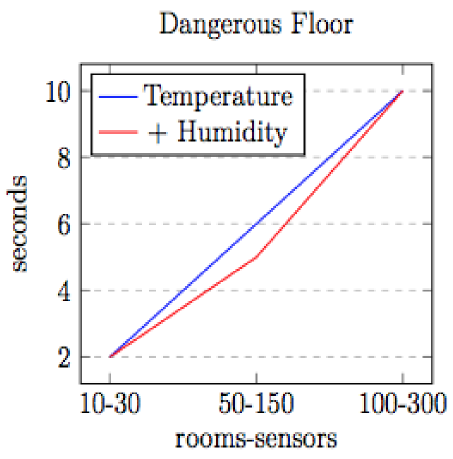

Figure 6.15: Third experiment DangerousFloor inference performance adding humidity

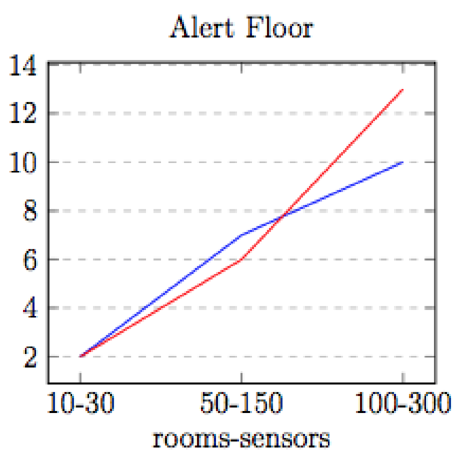

Figure 6.16: Fourth experiment AlerFloor inference performance adding humidity

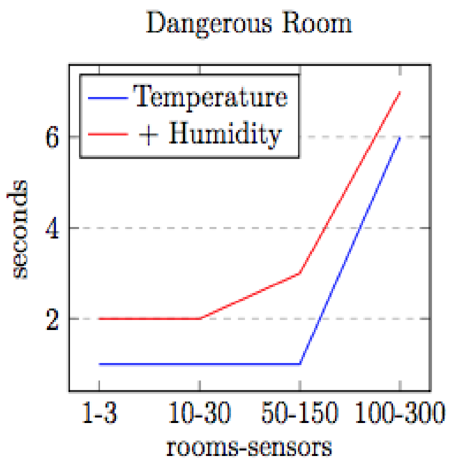

Figure 6.17: Fifth experiment DangerousRoom inference performance adding humidity

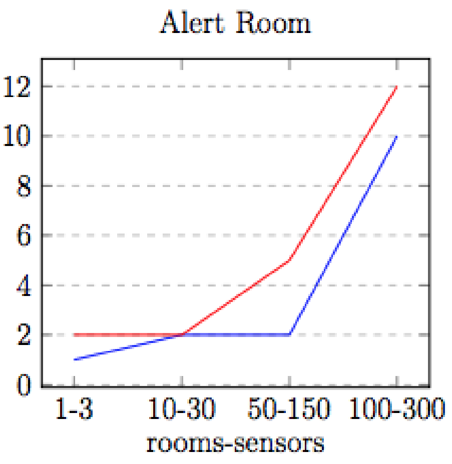

Figure 6.18: Fifth experiment AlertRoom inference performance adding humidity

Humidity and can have the values *low-humidity*, *normal-humidity* and *high-humidity*. With this addition, we must pass another RDF triple format, like in the experiments above. Besides passing the triple <M-Hub-UUID, observes, high-temperature>, we will pass <M-Hub-UUID, observes, low-humidity>. Bellow we have the updated ontology. The A-Box and the C-SPARQL query remains the same.

- **DangerousRoom**: *Room* **and** *has-device* **some** (*Device* **and** *observes* **value** *high-temperature* **and** *observes* **value** *low-humidity*)
- **DangerousFloor**: *Floor* **and** *has-hall* **some** (*Hall* **and** *has-room* **some** (*Room* **and** *has-device* **some** (*Device* **and** *observes* **value** *high-temperature* **and** *observes* **value** *low-humidity*)))

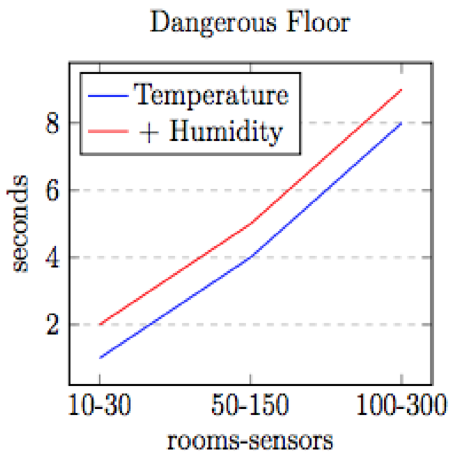

Figure 6.19: Fifth experiment DangerousFloor inference performance adding humidity

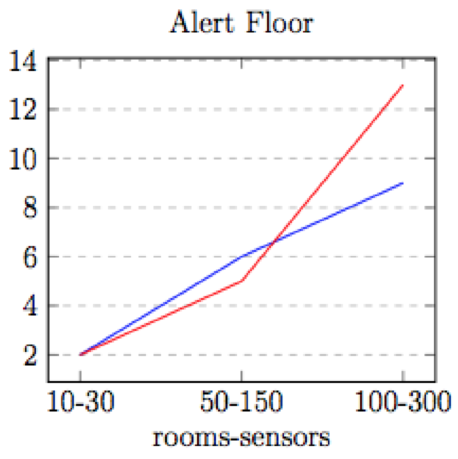

Figure 6.20: Fifth experiment AlertFloor inference performance adding humidity

Experiment	Temperature (seconds)	+ Humidity (seconds)
1	6	8
2	10	14
3	10	10
4	10	13
5.1	6	7
5.2	10	12
5.3	8	9
5.4	8	13
6.1	11	13
6.2	23	15
6.3	14	16
6.4	20	25

Table 6.2: Experiments comparison with humidity

According to Table 6.2 we can observe that the service response time increased in almost all scenarios with the addition of the humidity data. Again, we have to consider whether it is worth adding more data to be checked, when we can only use the temperature to do this type of verification. Of course, the more data to be checked, the more certain the service response will be.

6.3 Removing axioms from T-Box

In this section, ***the seventh experiment***, we will show that it is possible to infer the same data that we obtained in previous scenarios by removing all axioms from the ontology. To do this, we will take the ontology from scenarios 1 to 4 and change it to remove the axioms *DangerousRoom*, *AlertRoom*, *DangerousFloor*, *AlertFloor* and get service response time again.

It is important to note that there have been no changes to the A-Box. In addition, we will use the triple format RDF <Room-UUID, has-very-dry-sensor, M-Hub-UUID> and <Room-UUID, has-very-hot-sensor, Sensor-UIUD>.

For this experiment, we need another way of being able to make the inference. We can transfer this processing directly to the C-SPARQL query, as we will see below. In the updated queries, the WHERE clause must explicitly specify all the steps to make the inference. The queries that were used for this scenario are described in Listing 6.2, 6.3, 6.4, 6.5, 6.6, 6.7, 6.8 and 6.9 and Table 6.3 shows the comparison between all the experiments, with 400 individuals.

```
REGISTER QUERY dangerousRoom-temperature AS
PREFIX :<http://www.streamreasoning.com/>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
SELECT ?room
FROM STREAM <http://streamreasoning.com/streams/> [RANGE 1s STEP 1
    s]
FROM <http://streamreasoning.org/data>
WHERE {
    ?room :has-very-hot-sensor ?sensor
};
```

Listing 6.2: DangerousRoom considering only temperature

```
REGISTER QUERY dangerousRoom-temperature-humidity AS
PREFIX :<http://www.streamreasoning.com/>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
SELECT ?room
FROM STREAM <http://streamreasoning.com/streams/> [RANGE 1s STEP 1
    s]
FROM <http://streamreasoning.org/data>
WHERE {
    ?room :has-very-dry-sensor ?sensor
    ?room :has-very-hot-sensor ?sensor
};
```

Listing 6.3: DangerousRoom considering temperature and humidity

```
REGISTER QUERY alertRoom-temperature AS
PREFIX :<http://www.streamreasoning.com/>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
SELECT ?room2
FROM STREAM <http://streamreasoning.com/streams/> [RANGE 1s STEP 1
    s]
FROM <http://streamreasoning.org/data>
WHERE {
     ?room2  :is-near  ?room1
     ?room1  :has-very-hot-sensor  ?sensor
};
```

Listing 6.4: AlertRoom considering only temperature

```
REGISTER QUERY alertRoom-temperature-humidity  AS
PREFIX :<http://www.streamreasoning.com/>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
SELECT ?room2
FROM STREAM <http://streamreasoning.com/streams/> [RANGE 1s STEP 1
     s]
FROM <http://streamreasoning.org/data>
WHERE {
     ?room2  :is-near  ?room1
     ?room1  :has-very-hot-sensor  ?sensor
     ?room1  :has-very-dry-sensor  ?sensor
};
```

Listing 6.5: AlertRoom considering temperature and humidity

```
REGISTER QUERY dangerousFloor-temperature  AS
PREFIX :<http://www.streamreasoning.com/>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
SELECT ?floor
FROM STREAM <http://streamreasoning.com/streams/> [RANGE 1s STEP 1
     s]
FROM <http://streamreasoning.org/data>
WHERE {
     ?floor   :has-hall  hall
     ?hall    :has-room  ?room1
     ?room1   :has-very-hot-sensor  ?sensor
};
```

Listing 6.6: DangerousFloor considering only temperature

```
REGISTER QUERY dangerousFloor-temperature-humidity  AS
PREFIX :<http://www.streamreasoning.com/>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
SELECT ?floor
FROM STREAM <http://streamreasoning.com/streams/> [RANGE 1s STEP 1
     s]
FROM <http://streamreasoning.org/data>
WHERE {
     ?floor   :has-hall  hall
     ?hall    :has-room  ?room1
     ?room1   :has-very-hot-sensor  ?sensor
     ?room1   :has-very-dry-sensor  ?sensor
};
```

Listing 6.7: DangerousFloor considering temperature and humidity

```
REGISTER QUERY alertFloor-Temperature AS
PREFIX :<http://www.streamreasoning.com/>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
SELECT ?floor2
FROM STREAM <http://streamreasoning.com/streams/> [RANGE 1s STEP 1
    s]
FROM <http://streamreasoning.org/data>
WHERE {
    ?floor2 :is-near ?floor1
    ?floor1 :has-hall hall
    ?hall   :has-room ?room1
    ?room1  :has-very-hot-sensor ?sensor
};
```

Listing 6.8: AlertFloor considering only temperature

```
REGISTER QUERY alertFloor-temperature-humidity AS
PREFIX :<http://www.streamreasoning.com/>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
SELECT ?floor2
FROM STREAM <http://streamreasoning.com/streams/> [RANGE 1s STEP 1
    s]
FROM <http://streamreasoning.org/data>
WHERE {
    ?floor2 :is-near ?floor1
    ?floor1 :has-hall hall
    ?hall   :has-room ?room1
    ?room1  :has-very-hot-sensor ?sensor
    ?room1  :has-very-dry-sensor ?sensor
};
```

Listing 6.9: AlertFloor considering temperature and humidity

From the obtained results at Table 6.3, we noticed that the removal of axioms from the ontology and the transfer to the C-SPARQL query had a positive effect on service performance. We were able to obtain, with this change, the shortest times considering all the scenarios. With these results, we may consider, depending on the complexity of our application, to use axioms or transfer the intelligence to the query to search for inferences.

Experiment	Temperature (seconds)	+ Humidity (seconds)
1	6	8
2	10	14
3	10	10
4	10	13
5.1	6	7
5.2	10	12
5.3	8	9
5.4	8	13
6.1	11	13
6.2	23	15
6.3	14	16
6.4	20	25
7.1	4	4
7.2	2	3
7.3	6	4
7.4	5	4

Table 6.3: All experiments comparison

6.4 Summary

In this chapter we presented a systematic evaluation of the service's performance. The performance was measured according to the processing time from the data generation by the M-OBJ to the Reasoning Service's output. In order to obtain more complete metrics, we developed several experiments. The experiments served to measure the reliability of the service as well as the level of scalability.

In the first experiments, ranging from 1 to 4, we varied only the number of individuals contained in the A-Box, following the base ontology proposed in section 5. In this experiment, we managed to reach a total of 400 individuals in the A-Box, with a maximum response time of 10 seconds. For the proposed application, this response time is in accordance with a real-time system, allowing a fast response of some actuator.

In the subsequent experiment we added a specialization to the *Sensor* class, called the *VeryHotSensor*. In addition, the generated triple the first level has also been modified, causing the reasoning process to do the relationship between the room and the sensor, since the streamed triple does not bring this information. These modifications did not reflect so much on the service performance because the response time remained practically the same. The conclusion that we can obtain is that these small modifications in the ontology and in the streamed triple format did not affected the performance.

Experiment 6 brought a great change to the ontology, whereas the generated triple did not have major modifications in relation to the previous experiment. In this experiment, we based the ontology on the Semantic Sensor Network Ontology (SSN)[1], but using only a small idea of it. With this modification, we could notice that the response time in the service increased a lot in relation to the other experiments, reaching up to 25 seconds. This happened because the reasoning process has to do many more steps. With this, we can get to the conclusion that a very complex ontology, emphasizing that we do not use the complete SSN, brings us a significant loss in performance.

Also, we added the humidity measurement from the *SensorTag*. For this to be possible, we proceed to generate an extra triple, which contains this information. In addition, we altered the ontology to reflect this change. For the performance tests, we revisited all the previous experiments and made the time measurements again. The results showed a slight increase in response time, which was expected, since the reasoning process started to have one more step of inference.

Finally, in the last experiment, we wanted to show that it is possible to reach the same answer without the use of axioms in the ontology. With this, the ontology started to have a much smaller complexity, passing its complexity to the query, since it starts to have explicitly all the steps of inference. The results showed a drastic reduction in response time because the reasoning process was greatly simplified. This experiment showed us that, in order to achieve a better performance, we must consider how complex the ontology should be.

[1]https://www.w3.org/TR/vocab-ssn/

7
Discussion

According to the experiments presented in the previous chapter, we showed that we can model the fire detection application in several ways. This is not only useful for this application, since the addition, mainly, of a knowledge base together with an ontology brings us a range of possibilities of what regards modeling systems.

Adding more complexity to the ontology can become a problem for real-time systems. This is corroborated by our sixth scenario, where we added more specializations to the possible devices that the room can have. With that, the response time of the service increased greatly, and in some cases, up to 25 seconds. For more critical systems, where the response time must be at the millisecond level, this time is unacceptable and may even cause major disasters. A way that can be used to improve performance is, as we saw in the seventh experiment, is by removing all the axioms from the ontology. This really brings about a performance gain, but all the complexity of the ontology is brought into the query, which becomes very detailed and specific to a certain type of application. Keeping the intelligence of inference in ontology, through axioms, enables us to write more abstract queries without knowing deeply the way inferences are made.

The use of CEP, possible through integration with M-Hub, along with reasoning, which is done through C-SPARQL, brings us the possibility of being able to transfer some of the processing to one or the other. This is good because we can overlook C-SPARQL, for example, for very complex scenarios, which need a very fast response, and keep processing more at the CEP level, which tends to be faster because its architecture was born to be real-time. However, one point in favor of the use of ontologies is that they can describe classes and relations, adding more expressiveness to the data, while the CEP is responsible for the processing of raw data, not having direct access to the model described by the ontology. This process is described in experiments 1-4 and 5, where we eliminate the external service that relates the room-sensor and transfers the check to the ontology. Despite this, the performance did not have a great impact. But, for very large A-Boxes, it can lead to a problem, because it is one more step that we are doing in the inference. Therefore, we should consider whether the processing should be more in CEP part or on the ontology reasoning. A bad choice can compromise the performance and effectiveness of the application.

One of the negative points that were observed with the use of C-SPARQL was that from a certain number of instances in the A-Box, the service stops responding. The maximum we could process, as can be seen in the all the experiments, was on the order of 400 individuals in the A-Box, which we consider not to be very high, since a database of a normal application can easily have thousands or even millions of tuples. Another important point is that for 400 individuals C-SPARQL starts to behave intermittently, so that outliers in the performance measures begin to appear. In these cases, we disregard the outliers and consider only the results that are within normalcy. One of the solutions that can be applied to our service is that we can divide the processing by floors, for example, where each floor would contain a service being executed, thus limiting the size for each service to a maximum of 10 rooms and 30 sensors, which means 40 individuals in the A-Box.

Another way that the CEP can generate a high temperature event, is not only considering the gross temperature, but its variation within a time window. In CEP this is possible by using a pattern matching, that is, a match of an event sequence. This variation can bring us even more interesting facts, such as an indicator that a room is heating up.

Finnaly, we understand that different IoT applications has different requirements. Some applications handle frequently updating data and target low response time, but without complex ontologies and reasoning. However, other applications do not require low response time, but demand complete and expressive reasoning. It is very difficult to achieve a single solution for all the IoT applications, since IoT has a enormous actuation field. Special data models, query languages, reasoning approaches need to be developed to address different requirements accordingly. It remains an open issue to match reasoning ability and expressive power required in different applications.

8
Conclusions

The current IoT approach, especially in the field of sensor data analysis and process automation, provides a number of research opportunities and challenges. An enormous amount of sensors and actuators are present in diverse environments, such as in our residences, in a hospital or even in the streets of a city. Through the composition of these sensors and actuators, diverse applications can be created to bring about improvements in our lives.

One of the great challenges in IoT is being able to extract more complex information from the sensor-generated data. For this, a paradigm called Stream Reasoning was created, which deals with this problem in a more general way, not necessarily focusing on IoT, since many of the systems developed based on Stream Reasoning do not support one of the great IoT prerequisites, which is real-time processing.

In view of the introduced challenges, this work presented a Real-Time Stream Reasoning Service for IoT, which addresses the points 1) generate higher level complex data from sensors' raw data using CEP and Stream Reasoning techniques and 2) do it in real-time. Besides that, this works makes use of a technology totally implemented in the Laboratory for Advanced Collaboration (LAC), from PUC-RIO, the ContextNet. One of the greatest features from the ContextNet is that it abstracts all the discovery and communication layer between the sensors and, with this, we could focus directly in the CEP (M-Hub) and reasoning (C-SPARQL) implementation and experiments.

The service implementation and later the tests and experiments, showed that it is possible to make real-time stream reasoning system for IoT in real-time, given some prerequisites. These prerequisites include, as we saw in Chapter 5, the choice of a not very complex ontology and, due to the C-SPARQL limitations, also discussed in Chapter 5 and 6, the KB size should also be carefully chosen.

The study of related works, which have some degree of similarity with the work developed in this dissertation, showed some of the difficulties and possible solutions within the scope of Stream Reasoning in IoT. Some of these papers outline the main points that such a system should have, but the approach applied here, which combines the use of ContextNet with CEP processing capability and also Stream Reasoning, is innovative within this context.

8.1 Future Work

One of the desirable aspects of a stream reasoning mechanism with IoT focus is its ability to respond quickly to stimuli from the environment that the sensor is in. As we have seen, this point is still open because for very complex ontologies and for a large KB, the response time can not be considered more as real-time.

Results obtained up to now are important. Some of the mature solutions were exploited in real scenarios, such as social media analytics. We should get inspired by such results, and see them as the foundations to build new research and to reach new ambitious achievements. An application of these techniques that we saw during work, especially Stream Reasoning, in real IoT scenarios, is still necessary.

Another point that we can analyze as a possible improvement concerns the M-Hub. As we saw in detail in Chapter 4, the SCA is responsible for converting the CEP data generated by M-Hub into RDF triples. The idea, as a future work, is to eliminate the dependency on SCA and genarate the RDF triple directly on the mobile nodes (M-Hub). This is in line with IoT's vision of the future, since we are decentralizing some of the processing and distributing it.

Bibliography

[1] BERNESS-LEE, T.;HENDLER, J.;ORA LASSILA. **The semantic web.** 2001.

[3] LUCKHAM, D. C.. **The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems.** Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001.

[4] UNEL, G.; ROMAN, D.. **Stream reasoning: A survey and further research directions.** In: PROCEEDINGS OF THE 8TH INTERNATIONAL CONFERENCE ON FLEXIBLE QUERY ANSWERING SYSTEMS, FQAS '09, p. 653–662, Berlin, Heidelberg, 2009. Springer-Verlag.

[5] SU, X.; GILMAN, E.; WETZ, P.; RIEKKI, J.; ZUO, Y. ; LEPPÄNEN, T.. **Stream reasoning for the internet of things: Challenges and gap analysis.** In: PROCEEDINGS OF THE 6TH INTERNATIONAL CONFERENCE ON WEB INTELLIGENCE, MINING AND SEMANTICS, WIMS '16, p. 1:1–1:10, New York, NY, USA, 2016. ACM.

[6] PRUD'HOMMEAUX, E.; SEABORNE, A.. **Sparql query language for rdf, w3c recommendation.** 01 2008.

[7] PÉREZ, J.; ARENAS, M. ; GUTIERREZ, C.. **Semantics and complexity of sparql.** ACM Trans. Database Syst., 34(3):16:1–16:45, Sept. 2009.

[8] DELL'AGLIO, D.; VALLE, E. D.; VAN HARMELEN, F. ; BERNSTEIN, A.. **Stream reasoning : a survey and outlook a summary of ten years of research and a vision for the next decade.** 2017.

[9] VIDACKOVIC, S.; RENNER, T.. **Market overview real-time monitoring software - review of event processing tools.** 2010.

[10] ONKAR WALAVALKAR, ANUPAM JOSHI, T. F.; YESHA, Y.. **Streaming knowledge bases.** In: PROCEEDINGS OF THE FOURTH INTERNATIONAL WORKSHOP ON SCALABLE SEMANTIC WEB KNOWLEDGE BASE SYSTEMS, October 2008.

[11] BARBIERI, D. F.; BRAGA, D.; CERI, S.; VALLE, E. D. ; GROSSNIKLAUS, M.. **Querying rdf streams with c-sparql**. SIGMOD Rec., 39(1):20–26, Sept. 2010.

[12] BARBIERI, D. F.; BRAGA, D.; CERI, S.; DELLA VALLE, E. ; GROSSNIKLAUS, M.. **C-sparql: Sparql for continuous querying**. In: PROCEEDINGS OF THE 18TH INTERNATIONAL CONFERENCE ON WORLD WIDE WEB, WWW '09, p. 1061–1062, New York, NY, USA, 2009. ACM.

[13] LE PHUOC, D.; DAO-TRAN, M.; LE TUAN, A.; DUC, M. N. ; HAUSWIRTH, M.. **Rdf stream processing with cqels framework for real-time analysis**. In: PROCEEDINGS OF THE 9TH ACM INTERNATIONAL CONFERENCE ON DISTRIBUTED EVENT-BASED SYSTEMS, DEBS '15, p. 285–292, New York, NY, USA, 2015. ACM.

[14] ANICIC, D.; FODOR, P.; RUDOLPH, S. ; STOJANOVIC, N.. **Ep-sparql: A unified language for event processing and stream reasoning**. In: PROCEEDINGS OF THE 20TH INTERNATIONAL CONFERENCE ON WORLD WIDE WEB, WWW '11, p. 635–644, New York, NY, USA, 2011. ACM.

[16] VALLE, E. D.; CERI, S.; HARMELEN, F. V. ; FENSEL, D.. **It's a streaming world! reasoning upon rapidly changing information**. IEEE Intelligent Systems, 24(6):83–89, Nov. 2009.

[17] HARRIS, S.; SEABORNE (EDS), A.. **SPARQL 1.1 query language**. Working draft, W3C, 2010.

[18] ENDLER, M.; BRIOT, J. P.; E SILVA, F. S.; DE ALMEIDA, V. P. ; HAEUSLER, E. H.. **Towards stream-based reasoning and machine learning for iot applications**. In: 2017 INTELLIGENT SYSTEMS CONFERENCE (INTELLISYS), p. 202–209, Sept 2017.

[19] LUCKHAM, D. C.; SCHULTE, R.. **Event processing glossary**. May 2007.

[20] ETZION, O.; NIBLETT, P.. **Event Processing in Action**. Manning Publications Co., Greenwich, CT, USA, 1st edition, 2010.

[21] BALOGH, L.; DÁVID, I.; RÁTH, I.; VARRO, D. ; VÖRÖS, A.. **Distributed and heterogeneous event-based monitoring in smart cyber-physical systems**. 01 2016.

[22] DÁVID, I.; RÁTH, I. ; VARRO, D.. **Streaming model transformations by complex event processing**. p. 68–83, 09 2014.

[23] HARMELEN, F. V.; MCGUINNESS, D. L.. **Web ontology language overview**. Accessed: 2018-08-19.

[24] ARASU, A.; BABU, S. ; WIDOM, J.. **Cql: A language for continuous queries over streams and relations**. In: Lausen, G.; Suciu, D., editors, DATABASE PROGRAMMING LANGUAGES, p. 1–19, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

[25] CUGOLA, G.; MARGARA, A.. **Processing flows of information: From data stream to complex event processing**. ACM Comput. Surv., 44(3):15:1–15:62, June 2012.

[26] EGGUM, M.. **Smartphone assisted, complex event processing**. PhD thesis, 2014.

[27] ENDLER, M.; BAPTISTA, G.; SILVA, L. D.; VASCONCELOS, R.; MALCHER, M.; PANTOJA, V.; PINHEIRO, V. ; VITERBO, J.. **Contextnet: Context reasoning and sharing middleware for large-scale pervasive collaboration and social networking**. In: PROCEEDINGS OF THE WORKSHOP ON POSTERS AND DEMOS TRACK, PDT '11, p. 2:1–2:2, New York, NY, USA, 2011. ACM.

[28] TALAVERA, L. E.; ENDLER, M.; VASCONCELOS, I.; VASCONCELOS, R.; CUNHA, M. ; D. S. E. SILVA, F. J.. **The mobile hub concept: Enabling applications for the internet of mobile things**. In: 2015 IEEE INTERNATIONAL CONFERENCE ON PERVASIVE COMPUTING AND COMMUNICATION WORKSHOPS (PERCOM WORKSHOPS), p. 123–128, March 2015.

[29] RIOS, L. E. T.. **An energy-aware iot gateway, with continuous processing of sensor data**, 2015.

[30] SILVA, B.; FISHER, R. M.; KUMAR, A. ; HANCKE, G. P.. **Experimental link quality characterization of wireless sensor networks for underground monitoring**. IEEE Transactions on Industrial Informatics, 11(5):1099–1110, Oct 2015.

[31] CHIWEWE, T. M.; MBUYA, C. F. ; HANCKE, G. P.. **Using cognitive radio for interference-resistant industrial wireless sensor networks: An overview**. IEEE Transactions on Industrial Informatics, 11(6):1466–1481, Dec 2015.

[32] ZHANG, J.; LI, W.; HAN, N. ; KAN, J.. **Forest fire detection system based on a zigbee wireless sensor network**. Frontiers of Forestry in China, 3(3):369–374, Sep 2008.

[33] ARRUE, B. C.; OLLERO, A. ; DE DIOS, J. R. M.. **An intelligent system for false alarm reduction in infrared forest-fire detection**. IEEE Intelligent Systems and their Applications, 15(3):64–73, May 2000.

[34] CANDAN, K. S.; LIU, H. ; SUVARNA, R.. **Resource description framework: Metadata and its applications**. SIGKDD Explor. Newsl., 3(1):6–19, July 2001.

[35] THUY, P. T. T.; THUAN, N. D.; HAN, Y.; PARK, K. ; LEE, Y.-K.. **Rdb2rdf: Completed transformation from relational database into rdf ontology**. In: PROCEEDINGS OF THE 8TH INTERNATIONAL CONFERENCE ON UBIQUITOUS INFORMATION MANAGEMENT AND COMMUNICATION, ICUIMC '14, p. 88:1–88:7, New York, NY, USA, 2014. ACM.

[36] SU, X.; RIEKKI, J.; NURMINEN, J. K.; NIEMINEN, J. ; KOSKIMIES, M.. **Adding semantics to internet of things**. Concurr. Comput. : Pract. Exper., 27(8):1844–1860, June 2015.

[37] OMG. **The real-time publish-subscribe wire protocol dds interoperability wire protocol (dds-rtps)**, 2009.

[38] CORCHO, O.; CALBIMONTE, J.-P.; JEUNG, H. ; ABERER, K.. **Enabling query technologies for the semantic sensor web**. Int. J. Semant. Web Inf. Syst., 8(1):43–63, Jan. 2012.

[39] ÖZÇEP, Ö. L.; MÖLLER, R. ; NEUENSTADT, C.. **A stream-temporal query language for ontology based data access**. In: Lutz, C.; Thielscher, M., editors, KI 2014: ADVANCES IN ARTIFICIAL INTELLIGENCE, p. 183–194, Cham, 2014. Springer International Publishing.

[40] GRAU, B. C.; HALASCHEK-WIENER, C. ; KAZAKOV, Y.. **History matters: Incremental ontology reasoning using modules**. In: PROCEEDINGS OF THE 6TH INTERNATIONAL THE SEMANTIC WEB AND 2ND ASIAN CONFERENCE ON ASIAN SEMANTIC WEB CONFERENCE, ISWC'07/ASWC'07, p. 183–196, Berlin, Heidelberg, 2007. Springer-Verlag.

[41] WALAVALKAR, O.; JOSHI, A.; FININ, T. ; YESHA, Y.. **Streaming knowledge bases**. 08 2018.

[42] DAVIDE, D. C.. **Sparkwave : Continuous schema-enhanced pattern matching over rdf data streams**. 2012.

[43] FORGY, C.. **Rete: A fast algorithm for the many pattern/many object pattern match problem**. Artificial Intelligence, 19(1):17–37, 1982.

[44] RINNE, M.; SOLANKI, M. ; NUUTILA, E.. **Rfid-based logistics monitoring with semantics-driven event processing**. In: PROCEEDINGS OF THE 10TH ACM INTERNATIONAL CONFERENCE ON DISTRIBUTED AND EVENT-BASED SYSTEMS, DEBS '16, p. 238–245, New York, NY, USA, 2016. ACM.

[45] BALDUINI, M.; DELLA VALLE, E.; DELL'AGLIO, D.; TSYTSARAU, M.; PALPANAS, T. ; CONFALONIERI, C.. **Social listening of city scale events using the streaming linked data framework**. In: Alani, H.; Kagal, L.; Fokoue, A.; Groth, P.; Biemann, C.; Parreira, J. X.; Aroyo, L.; Noy, N.; Welty, C. ; Janowicz, K., editors, THE SEMANTIC WEB – ISWC 2013, p. 1–16, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[46] BRESLIN, J. G.; DECKER, S.; HARTH, A. ; BOJARS, U.. **Sioc&#58; an approach to connect web&#45;based communities**. Int. J. Web Based Communities, 2(2):133–142, July 2006.

[47] COMPTON, M.; BARNAGHI, P.; BERMUDEZ, L.; GARCÍA-CASTRO, R.; CORCHO, O.; COX, S.; GRAYBEAL, J.; HAUSWIRTH, M.; HENSON, C.; HERZOG, A.; HUANG, V.; JANOWICZ, K.; KELSEY, W. D.; LE PHUOC, D.; LEFORT, L.; LEGGIERI, M.; NEUHAUS, H.; NIKOLOV, A.; PAGE, K.; PASSANT, A.; SHETH, A. ; TAYLOR, K.. **The ssn ontology of the w3c semantic sensor network incubator group**. Web Semant., 17(C):25–32, Dec. 2012.

[48] FRANCESCO BARBIERI, D.; DELLA VALLE, E.. **A proposal for publishing data streams as linked data - a position paper.**, 01 2010.

[49] BAADER, F.; HORROCKS, I.; LUTZ, C. ; SATTLER, U.. **An Introduction to Description Logic**. Cambridge University Press, United Kingdom, 4 2017.

[50] GRUBER, T. R.. **A translation approach to portable ontology specifications**. Knowl. Acquis., 5(2):199–220, June 1993.

[51] MERZ, S.; WIRSING, M. ; ZAPPE, J.. **A spatio-temporal logic for the specification and refinement of mobile systems**. In: Pezzè, M.,

editor, FUNDAMENTAL APPROACHES TO SOFTWARE ENGINEERING, p. 87–101, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg.

[52] CLARKE, B. L.. **A calculus of individuals based on "connection"**. Notre Dame Journal of Formal Logic, 22:204–218, 1981.

[53] EGENHOFER, M. J.; FRANZOSA, R. D.. **Point-set topological spatial relations**. 2001.

[54] RANDELL, D. A.; CUI, Z. ; COHN, A. G.. **A spatial logic based on regions and connection**. In: KR, 1992.

[55] ALLEN, J. F.. **Maintaining knowledge about temporal intervals**. Commun. ACM, 26:832–843, 1983.

[56] CLARKE, E. M.; EMERSON, E. A.. **Design and synthesis of synchronization skeletons using branching-time temporal logic**. In: LOGIC OF PROGRAMS, WORKSHOP, p. 52–71, Berlin, Heidelberg, 1982. Springer-Verlag.

[57] ICK MOON, S.; LEE, K. H. ; LEE, D.. **Fuzzy branching temporal logic**. IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics), 34(2):1045–1055, April 2004.

[58] SOTUDEH, G.; MOVAGHAR, A.. **Abstraction and approximation in fuzzy temporal logics and models**. Formal Aspects of Computing, 27(2):309–334, Mar 2015.

[59] MESEGUER, J.. **General logics**supported by office of naval research contracts n00014-82-c-0333 and n00014-86-c-0450, nsf grant ccr-8707155 and by a grant from the system development foundation**. In: Ebbinghaus, H.-D.; Fernandez-Prida, J.; Garrido, M.; Lascar, D. ; Artalejo, M. R., editors, LOGIC COLLOQUIUM'87, volumen 129 de **Studies in Logic and the Foundations of Mathematics**, p. 275 – 329. Elsevier, 1989.

[60] FINGER, M.; GABBAY, D. M.. **Adding a temporal dimension to a logic system**. Journal of Logic, Language and Information, 1:203–233, 1992.

[61] MERZ, S.; ZAPPE, J. ; WIRSING, M.. **A spatio-temporal logic for the specification and refinement of mobile systems**. In: Pezzè, M., editor, FUNDAMENTAL APPROACHES TO SOFTWARE ENGINEERING

(FASE 2003), volumen 2621 de **Lecture Notes in Computer Science**, p. 87–101, Warsaw, Poland, April 2003. Springer-Verlag.

[62] VILAIN, M.; KAUTZ, H. ; VAN BEEK, P.. **Readings in qualitative reasoning about physical systems**. chapter Constraint Propagation Algorithms for Temporal Reasoning: A Revised Report, p. 373–381. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1990.

[63] IBM. `https://www.ibm.com/support/knowledgecenter/en/SSGMCP_4.1.0/com.ibm.cics.ts.eventprocessing.doc/concepts/dfhep_model.html`. [Online; accessed 27-August-2018].

[64] ANTONIOU, G.; VAN HARMELEN, F.. **Web Ontology Language: OWL**, p. 67–92. Springer Berlin Heidelberg, Berlin, Heidelberg, 2004.

[65] HUSTADT, U.; MOTIK, B. ; SATTLER, U.. **Data complexity of reasoning in very expressive description logics**. In: PROCEEDINGS OF THE 19TH INTERNATIONAL JOINT CONFERENCE ON ARTIFICIAL INTELLIGENCE, IJCAI'05, p. 466–471, San Francisco, CA, USA, 2005. Morgan Kaufmann Publishers Inc.

[66] CHANDRASEKARAN, S.; COOPER, O.; DESHPANDE, A.; FRANKLIN, M. J.; HELLERSTEIN, J. M.; HONG, W.; KRISHNAMURTHY, S.; MADDEN, S. R.; REISS, F. ; SHAH, M. A.. **Telegraphcq: Continuous dataflow processing**. In: PROCEEDINGS OF THE 2003 ACM SIGMOD INTERNATIONAL CONFERENCE ON MANAGEMENT OF DATA, SIGMOD '03, p. 668–668, New York, NY, USA, 2003. ACM.