(FP7 614100)

D5.1.2 Updated Data Analysis & Knowledge Repository Technical Specifications & Guidelines

Date 3 March 2015 – Version 1.0

Published by the IMPReSS Consortium

Dissemination Level: Public

Project co-funded by the European Commission within the 7th Framework Programme and the Conselho Nacional de Desenvolvimento Científico e Tecnológico
Objective ICT-2013.10.2 EU-Brazil research and development Cooperation

Document control page

Document file:	IMPRESS Deliverable D5.1.2 v1.0.odt
Document version:	1.0
Document owner:	Djamel Sadok (UFPE)
Work package:	WP5 – Data Storage, Analysis & Decision Support
Task:	T5.1 Data and knowledge management support
Deliverable type:	R
Document status:	[*] approved by the document owner for internal review [*] approved for submission to the EC

Document history:

Version	Author(s)	Date	Summary of changes made
0.1	Lucas Lira Gomes (UFPE)	16/02/2015	First Draft.
0.2	Lucas Lira Gomes (UFPE)	18/02/2015	Updated the data model section.
0.3	Lucas Lira Gomes (UFPE)	21/02/2015	Introduced Gremlin Basics subsection.
0.4	Lucas Lira Gomes (UFPE)	25/02/2015	Updated steps and functions descriptions. Revised references.
0.5	Lucas Lira Gomes (UFPE)	29/02/2015	Provided examples for steps and functions.
1.0	Lucas Lira Gomes (UFPE)	03/03/2015	Sent for internal reviewing.

Internal review history:

Reviewed by	Date	Summary of comments
Marc Jentsch (FIT)	05/03/2015	Accepted with minor corrections and comments.
Enrico Ferrera (ISMB)	05/03/2015	Accepted with minor comments.

Index:

Table of Contents

Executive summary

This deliverable describes the updated technical specification of the data and knowledge repository used by the IMPReSS cloud. In the IMPReSS cloud, data semantics and analytics are fundamental to the the decision making process. Effectively managing the storage resources and data in the cloud is of paramount importance to maintaining satisfactory service levels.

In that sense, this deliverable aims to provide the foundations for a distributed and scalable storage solution for the IMPReSS cloud. Despite that, we organise this deliverable as follows. Section 2 contains the updated data model and the architecture for the Data, Policy and Knowledge Storage module. Section 3 focus on documenting the IMPReSS Storage domain specific language (DSL), which aims to facilitate queries related to the aforementioned data model, as well as providing examples of its usage. In section 4, the proposed architecture is evaluated in terms of performance. Finally, we provide our summary and conclusion.

Introduction

The Internet of Things (IoT) is a concept that encloses a plethora of technologies and their applications, providing the means to access and control all kinds of smart devices (also named as “things”). IoT covers a wide range of objects, such as sensors, actuators, mobile devices, industrial controllers, HVAC (heating, ventilation, air-conditioning) units, household appliances like smart TVs and refrigerators, and so on. Radio Frequency Identification (RFID) and sensor network technologies have often been used to measure, infer and understand environmental indicators around us. This often results in the generation of huge volumes of data that have to be stored, processed and presented in a seamless, efficient, and easily interpretable form [1]. To meet these requirements, the Cloud Computing technologies can be used as an infrastructure for the storage, processing and computing of such massive amounts of data generated by highly distributed smart devices (e.g. sensors and actuators).

It is in this context, that this deliverable presents the updated technical specification of the data and knowledge repository adopted for use within the IMPReSS project [2]. The main idea is to provide a cloud infrastructure able to manage very large sets of globally distributed non-structured or semi-structured data. The data generated by devices employed in the IMPReSS platform will be produced at very high rates and needs to be pre-processed in a timely manner, in order to be used as input by the data analysis and machine learning modules (as described in Tasks 5.2 and 5.3 of the project proposal).

The storage solution, in that sense, greatly affects Work Package 5 (Data Storage, Analysis & Decision Support). This Work Package is concerned with providing tools that allows developers to consistently manage massive amounts of data from smart devices. That is, analyze, extract information and finally transform data into knowledge that is useful for the application domain within the integrated IMPReSS platform. This deliverable, in particular, updates the initial Data, Policy and Knowledge Storage module specification.

1. IMPReSS Platform Overview

The IMPReSS platform can be seen as a software platform in the cloud designed to manage ubiquitous sensor data. It was engineered to provide a set of tools to help users (end-users and application developers) to build and manage connected smart devices and applications based on connected things.

The platform has been designed to facilitate the integration with sensing technologies, networking applications, data mining and processing tools. This enables users to collect and visualize environmental information while compiling and adding value to such information, in order to generate knowledge about the acquired data.

Figure 1 provides a complete overview of the IMPReSS platform components.

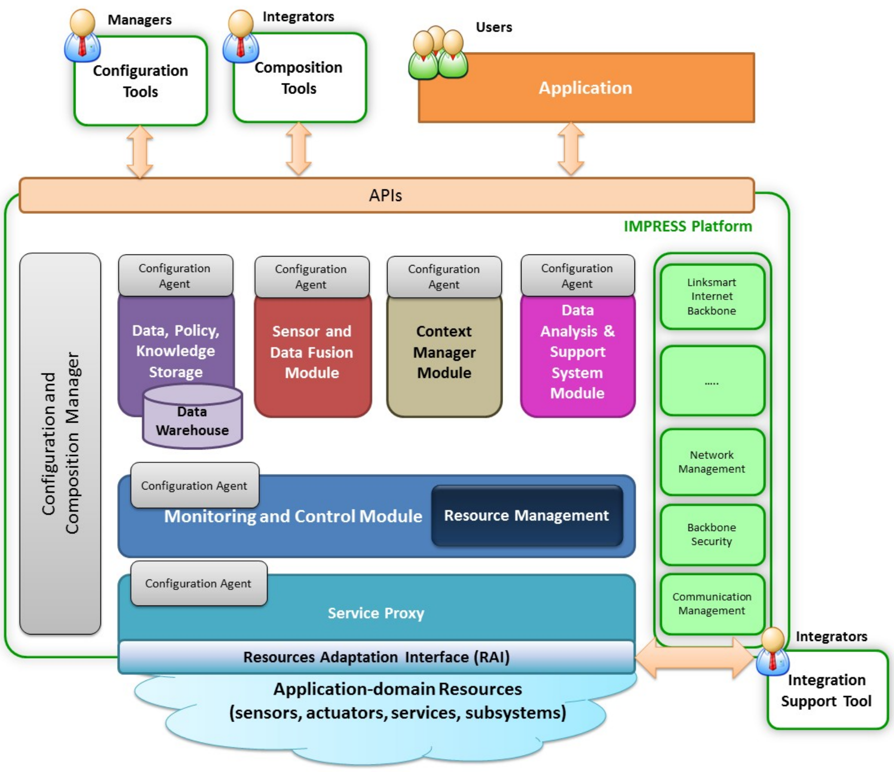
Managers
Configuration Tools
Integrators
Composition Tools
Users
Application
APIs
IMPRESS Platform
Configuration and Composition Manager
Configuration Agent
Data, Policy, Knowledge Storage
Data Warehouse
Sensor and Data Fusion Module
Context Manager Module
Data Analysis & Support System Module
Linksmart Internet Backbone
.....
Network Management
Backbone Security
Communication Management
Monitoring and Control Module
Resource Management
Service Proxy
Resources Adaptation Interface (RAI)
Application-domain Resources (sensors, actuators, services, subsystems)
Integrators
Integration Support Tool

Figure 1: IMPReSS platform Overview.

The Data, Policy and Knowledge Storage module is responsible for managing the persistence of various data and information sets. By leveraging on NoSQL databases, it maintains information such as historical sensor data, inferred knowledge, policies, configurations, etc. It makes services available for others components of the IMPReSS platform, such as the storage of both raw data and enhanced information.

Next sections provide details of data storage specification and implementation.

2. Data, Policy and Knowledge Storage Module

2.1 Data Model

The term data model has been used in the information management community with different meanings and in diverse contexts. In its most general sense, a data model is a concept that describes a collection of conceptual tools for representing real-world entities to be modelled and the relationships among these entities [3].

There are different models that may be used in the data modelling area such as hierarchical, relational, semantic, object-oriented, graph, and semi-structured. Among all of them, the relational model, which introduces the idea of separation between physical and logical levels, is the most popular and widely employed among the business applications [4]. However, this classical model has been criticized for its lack of semantics. Its flat structure imposes difficulties for the user to map the connectivity of the data, both conceptually and during the implementation.

Under the IMPReSS platform, the data semantics and analytics are fundamental features needed to support the decision making process. Multi sensor data fusion provides a means to fuse raw data into meaningful higher-level information for the users. Moreover, the recognition of the modelled situations requires understanding the technicalities of each sensor, signal processing and sensor fusion techniques to combine readings from different sensors. In such scenario, where the information about the interconnectivity or the topology of the data is more important than, or as important as, the data itself, the data modelling based on graph has several advantages.

First, graphs provide a natural and flexible way to represent information about real world (i.e. real world objects are vertexes and relations between different objects are edges).

Second, typical graph databases provide built-in structures (i.e. nodes and edges) to represent graphs. Whereas in other databases, relationships between entities in the data model would have to be handled by the modeller at the model level. Or in other words, new tables or columns, at least in the SQL case, would have to be maintained only for the sake of being used as query indirection stages that point to other entities, probably via foreign keys.

For these reasons, the data modelling adopted in the project is based on a property graph representation, implemented by most well known NoSQL graph databases (i.e. Titan, Neo4J and OrientDB.). In the realm of graphs' morphism, a property graph is a vertex/edge-labeled/attributed, directed, multi-graph. More details, on why a property graph representation was favoured over RDF's edge-labeled directed graphs, will be given in the Architecture section. The data modelling is based on sensor readings arranged in a certain physical environment. The setting may have an infinite number of areas, which in turn may or may not embody other areas within it. Each area may contain an indefinite number of devices that belong to a sensor network. These devices will perform several measurements of the various parameters throughout the day, while it is necessary to store a history of such readings possibly for an indefinite time, depending on application requirements. A generic description of the IMPReSS scenario is shown in Figure 2.

The data modelling has the following types of nodes:

- Area (yellow) - these nodes store a representation of a given monitored environment, such as its name and domain data. They are organised hierarchically in order to divide the environment into parts (e.g. rooms, hall, garden).
- Device (green) - these nodes represent the devices contained in the environment, such as sensors, actuators, controllers, and mobile devices. This node entails the type of device, its network address and domain data.
- Measurement Variable (red) – these nodes represent the variables being measured by a device, such as humidity, temperature, and energy.
- Measurement History (purple) – these nodes represent the measurements made by a device, at a given instant in time.
- Category (blue) – A node for classification of devices (e.g. illumination, HVAC). Each category is unique and can classify devices in a one-to-many fashion.

As for the edges, the types are:

- has – these edges link an area to the sub-areas it is composed of. Generating a hierarchy of spatial representations of the measured environment.
- interacts – this edge specifies which measurement variables can be measured by the device it is linked to.
- was measured – this edge link measurement histories, in a chronological order, to the device that measured it. Note that this edge, in particular, contains a timestamp property, representing a given instant in time.
- comprehends – is the relation between a category and the device it classifies. Each category can comprehend many devices with the same characteristics.

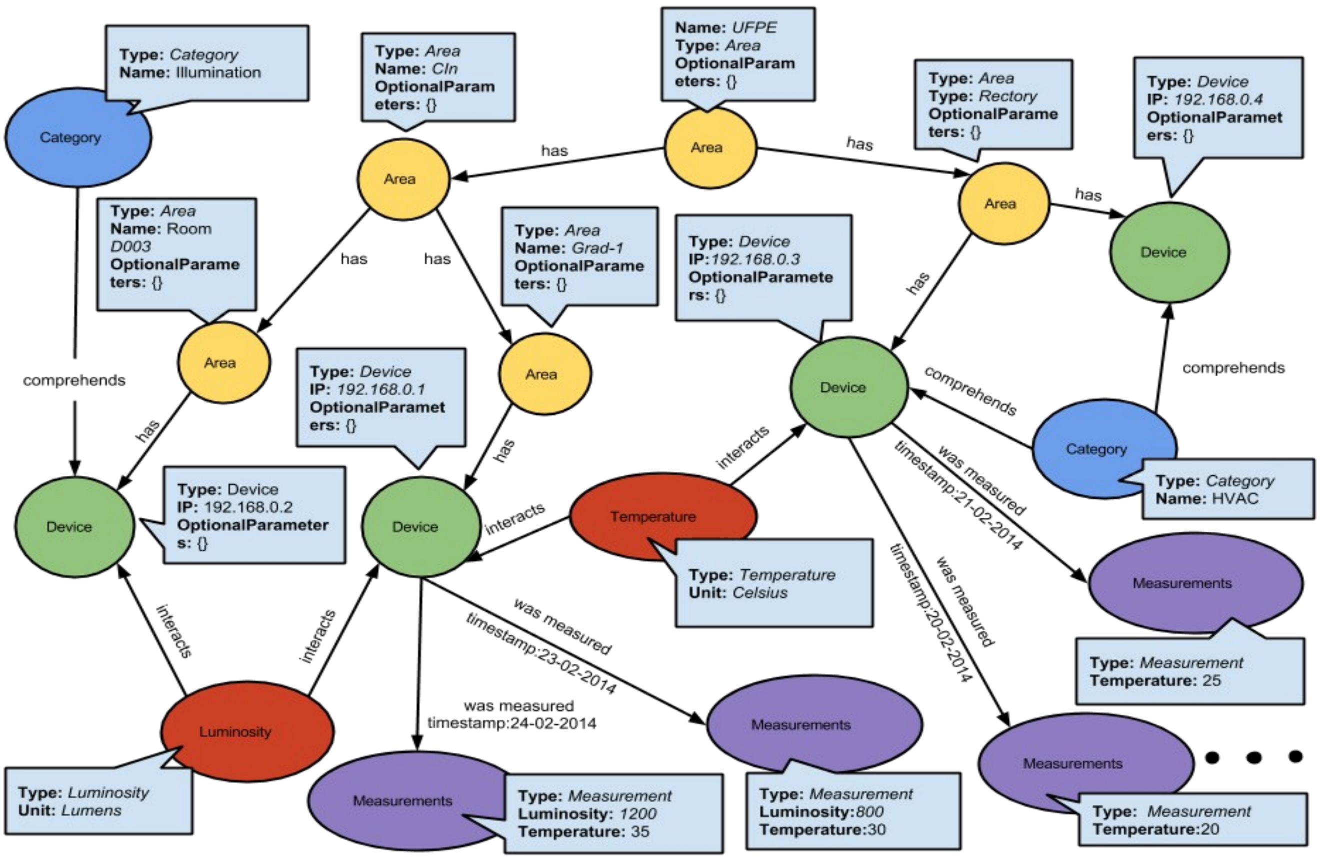
Type: Category Name: Illumination
Category
Type: Area Name: Cln OptionalParameters: {}
Name: UFPE Type: Area OptionalParameters: {}
Type: Area Type: Rectory OptionalParameters: {}
Type: Device IP: 192.168.0.4 OptionalParameters: {}
Area
has
Type: Area Name: Room D003 OptionalParameters: {}
Type: Area Name: Grad-1 OptionalParameters: {}
Type: Device IP:192.168.0.3 OptionalParameters: {}
Device
comprehends
Type: Device IP: 192.168.0.1 OptionalParameters: {}
Type: Device IP: 192.168.0.2 OptionalParameters: {}
interacts
Temperature
Type: Temperature Unit: Celsius
was measured timestamp:21-02-2014
was measured timestamp:20-02-2014
was measured timestamp:23-02-2014
was measured timestamp:24-02-2014
Type: Category Name: HVAC
Measurements
Type: Measurement Temperature: 25
Luminosity
Type: Luminosity Unit: Lumens
Type: Measurement Luminosity: 1200 Temperature: 35
Type: Measurement Luminosity:800 Temperature:30
Type: Measurement Temperature:20

Figure 2: Data Model for the Data, Policy and Knowledge Storage module.

As a result of the data model depicted at Figure 2, flexible and powerful queries could be performed, such as:

- Querying which devices can/cannot measure a given measurement variable. As well as list the areas that have devices measuring their temperature, for instance.
- Querying the area where a given device is located, via the device's IP. Or, alternatively, list all the devices in a given area.
- Querying all the sub-areas of a given area, via its area's name.
- Querying all the measurement histories, in a given time range, for a specific area. Despite the device that measured them.
- Etc.

2.2 Proposed Architecture

The Data and Knowledge Storage module consists of a set of technologies responsible for managing and storing data. These technologies are based on a NoSQL database, more specifically, a graph-based one.

Graph databases are perhaps the most popular graph computing technology. They provide transactional semantics such as ACID, which is typical of local databases, and eventual consistency, which is typical of distributed databases. Different from in-memory graph toolkits, graph databases use the disk to store the graph data. On sufficiently powerful machines, local graph databases can support a couple billion edges while distributed systems can handle hundreds of billions of edges. However most distributed graph-based NoSQL databases, like Neo4j [11], does not provide the means for global graph algorithms to be performed within a reasonable milliseconds time scale, in a hundreds of billions of edges scenario. And since WP5 tasks leverage heavily in the data processing for the machine learning and data fusion techniques, be able to have a continuous feedback loop that works almost in quasi real time and have a global view of the current and past state of the system, mainly due to global graph algorithms, is invaluable.

Considering this practical concern, we adopt the Titan [5] open implementation as distributed graph-based NoSQL database. Therefore, we can represent our data model, as depicted in Figure 2, without any modifications. This data model fits perfectly the knowledge inference case that further WP5 tasks require, since knowledge can be easily represented with graphs as a set of relations between concepts. RDF and ontologies, for instance, are just graphs connecting subjects and objects via a predicate, i.e., triples. Graph-based NoSQL databases, however, have a clear advantage over RDF and ontologies. They have built-in database support to triples, while ontologies and RDF require extra parsers, at the application level, to extract semantics from the employed syntax (e.g. XML, Turtle, Notation 3, etc). Obviously, RDF is a standard and is widely used by the linked data community, however it was not envisioned to be used in a distributed context that suits our proposed use case. It certainly fits well for the use case of the web, with a whole architecture based on documents being exchanged from a web server to clients using a request-reply pattern. But as the size of the a RDF document grows, it is up to the libraries' implementers to figure out how to deal with scalability problems. Like graph partitioning and distributed processing of the RDF documents. Not handling that can hinders the usage of RDF to store vast amounts of data.

As for Titan, graph partitioning, among Titan instances, and distributed batch processing, via Faunus, are already implemented. These two features per se permits Titan to scale horizontally, which was one of our major concerns from the very beginning.

Despite that, property graphs explicitly separate out node/edge specific key/value data from the underlying graph structure as a design-time decision. When using triple stores in practice, most of the edges turn out to be spurious. Since 'properties' of a node are not first class citizens of the graph structure itself. In RDF, for instance:

:a :hasAge "24".
:a foaf:knows :b.

These are both triples and hence considered graph edges, but only the second one represents connectivity in a graph sense. The first 'edge' is not really an edge, but a property of :a with no meaning outside of :a, since it is simply a literal and not a real entity per se.

As a side note, Titan supports several storage backends. Like Cassandra, which is a column-family NoSQL database developed and open sourced by Facebook in 2008, Hbase [19], which is an open source implementation of Google's BigTable and Oracle Berkeley DB [20]. For this proposed architecture, we favoured Cassandra, due to its maturity and large developer community.

At the scale of hundreds of billions of edges and with several concurrent users, where random access to disk and memory are at play, global graph algorithms are not feasible. What is feasible is local graph algorithms/traversals. Instead of traversing the entire graph, some set of vertices serve as the source (or root) of the traversal. To tackle the need for global graph algorithms/traversals, batch processing graph frameworks can be used. Most of the popular frameworks in this space leverage on Hadoop [16] for storage (HDFS) and processing (MapReduce). These systems are oriented towards global analytics. That is, transversals that pass through the entire graph dataset and, in the case of iterative algorithms, touch the entire graph many times. Such analyses do not run in real-time. However, because they perform global scans of the data, they can leverage sequential reads from disk (see [6]).

Along with Titan, we use Faunus [7] for distributed batch processing and support graph analytics in a timely manner. Faunus is able to distribute parts of a query among the available Titan clusters. Therefore, load balancing the workload drastically reduces latency for database operations in graphs with billions of edges and nodes. Faunus works on top of Hadoop, which is an open source project backed by the Apache Foundation and based on Google's Map-Reduce white paper. Also, it is noteworthy that both Titan and Faunus, following the trend of technologies around NoSQL databases, can scale horizontally by adding more clusters. That is, if in a given scenario the current servers can no longer handle the load, one may simply execute more instances of Faunus and Titan for employing a divide and conquer strategy to attend the usual batch of database queries performed in our proposed architecture.

For querying, we use a domain-specific language, the Gremlim language [8], which can perform complex operations in multi-relational graphs, called property graphs. Gremlim extends the Groovy language [10], which runs on top of the JVM. With Gremlin it is possible to perform operations such as the addition and/or removal of nodes/edges, manipulate graph indexes, complex graphs transversals, etc. Also, it is part of the Blueprints [5] stack. The equivalent of Gremlin in the RDF world is SPARQL [17]. Comparatively, SPARQL does not support iteration/looping, consequently being particularly hard to compute graph-structural metrics like centrality. In that sense, Gremlin is more powerful than SPARQL.

The Blueprints stack is an open source property graph model for a common interface that can facilitate the interaction with the underlying supported graph databases. Among the supported databases there are: Neo4j [11], Titan [5], OrientDB [12] and more. More importantly, these cited databases are currently seen as some of the major players in terms of graph database usage. Despite the considerable number of supported alternatives, leveraging on a common interface can help us to avoid having the IMPReSS architecture tied to specific proprietary solutions. Such problem could impose barriers in the event of changing the underlying graph database used by the IMPReSS cloud. The blueprints stack is maintained by a group called Tinkerpop [14], which entails as one of its members the lead developer of Titan. Also, notice that Blueprints is not a programming library per se. It is an API that has several implementations in many programming languages, like Java and Python.

Finally, we chose to use a Rexster server [9], which is also part of the Blueprints stack, to be the interface exposed for developers to execute database operations, via Gremlim queries. Or, in other words, the Rexster server allows developers to communicate with Blueprints-enabled graphs in a language agnostic fashion. That is, we could change the underlying NoSQL graph database at any time, without requiring any source code change in the clients of the Data, Policy and Knowledge Storage module. So, by using Rexster, developers can access the Blueprints API over HTTP/REST directly or by using libraries that support the Blueprints API, like PyBulbs [18] for Python. Both Titan and Faunus clusters are just part of the required infrastructure, but they are not directly exposed to other modules. The Rexster server supports both a JSON-based REST interface and a binary protocol called RexPro. In our architecture, we favoured the RexPro case due to its smaller footprint. When a Gremlim query is received, the Rexster server passes it to one of the Faunus clusters and waits for the response, which is then replied back to the requester.

Finally, Figure 3 shows the overview of the Data, Policy and Knowledge Storage module architecture.

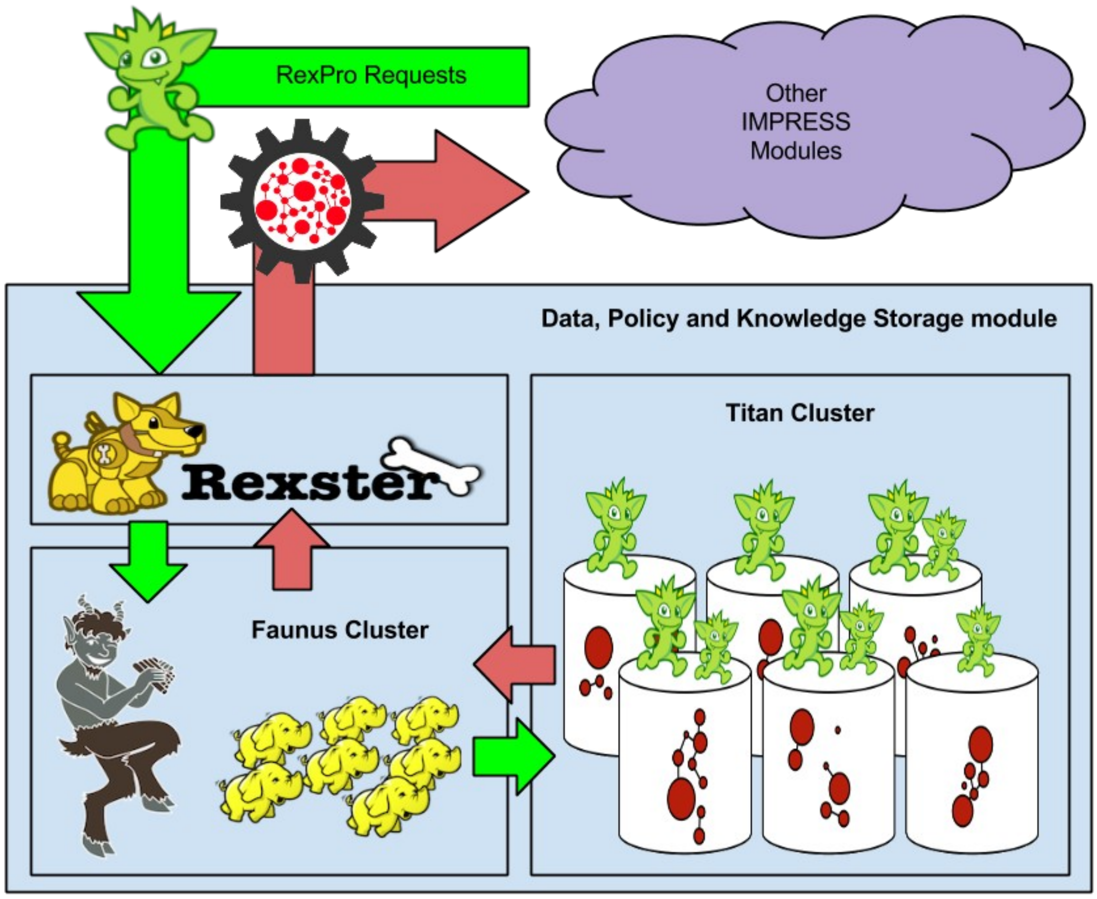

Figure 3: Data, Policy and Knowledge Storage module architecture.

3. Domain Specific Language – DSL

Gremlim is a flexible and powerful query language, but it certainly requires previous knowledge about graphs structures (i.e. nodes and edges) and their algorithms. Albeit graphs are certainly not a new concept, especially due to their mathematical origin, their usage for modelling data and consequently the awareness of their implications still not mainstream. Therefore, we have built a domain specific language (DSL) on the top of Gremlim, so that we can abstract the most common operations (e.g. adding/removing nodes, dealing with relationship between nodes, etc) regarding the data model depicted in Figure 2. For instance, instead of directly creating a node for a device, set its IP property, link it to the intended area and to a set of measurement variable nodes, a developer could just use the createDevice (g, ip, parentAreaName, measurementVariableNames) construct for doing all that at once. Therefore, by leveraging on the DSL, most parts of the graph manipulation process, regarding the data model, is transparent for clients. Alternatively, the DSL can be seen as a higher level interface (API) to interact with the proposed data model depicted in Figure 2. From this point on, we will refer to this new DSL as IMPReSS' Storage DSL. The IMPReSS' Storage DSL, however, is not as flexible as Gremlim's core constructs. Nevertheless, since IMPReSS' Storage DSL is basically just another layer on top of Gremlim, the underlying query language remains Gremlim. As a result of that, the DSL can please both newcomers, eager to perform simple queries, and more advanced users, that can mix Gremlim core constructs with IMPReSS DSL's calls.

3.1 Gremlin Basics

It is important to realize that Gremlin provides two abstractions for performing queries/operations: Functions and Steps. The former is exactly the same as functions in traditional programming languages. That is, there a set of arguments that will be received as input, it will be processed and an output, if any, will be generated. As for steps, they are better described as a chain of operations that are read from left to right. Each step, which is separated by the dot operator, can be seen as a function that operates on the output from the previous step.

3.2 IMPReSS' Storage DSL Steps

Steps can be used in two different notations:

Postfix notation:
g.V.data.step

InFix notation:
g.V.step(data)

Both notations produce the same outcome, so it is up to developers to use the syntax that suits them best. In order to clarify steps usage, see the following example and keep in mind that each one of the four parts of the aforementioned query are steps per se:

```
gremlin> g.V.area().name
```

The query above can be read in four steps, from left to right:

1. g: get the current graph.

2. V: get all vertices from the provided graph.
3. area(): get all the area's vertices.
4. name: get the name property from area's vertices.

Finally, following subsections will present the documentation related to each one of the nodes in the data model section. That is, areas, devices, categories, measurement variables and measurements' nodes. Each subsection will contain a table, in which the “Description” column explains the expected behaviour for a step and its arguments, while the “DSL Code” column presents the query signature.

Area Steps

The table bellow encopasses the documentation for all steps related to area nodes, as presented in the data model section.

Description	DSL Code
Returns a given area, if an area with the provided areaName exist. Also, if areaName is null, all areas will be returned. areaName: String, null	g.V.area(areaName)
Returns an area, if it contains a device with the provided deviceIp inside. Also, if deviceIp is null, all areas that contains devices will be returned. deviceIp: String, null	g.V.areaPerDevice(deviceIp)
Returns a given area, if an area with the provided areaName is inside it. Also, if areaName is null, all areas with parents will be returned. areaName: String, null	g.V.areaPerArea(areaName)
Returns the list of all areas, which contains devices that had already measured a given measurementVariableName (e.g. Temperature, Humidity). Also, if measurementVariableName is null, all areas with devices that have measurement variables will be returned. measurementVariableName: String, null	g.V.areaPerMeasurementVariable(measurementVariableName)
Returns the list of all areas that has the specified key-value pair, as part of their optional parameters. In case no value is specified, areas containing the provided key, despite their value, will be returned. key: String value: String, Int, Float, null	g.V.areaPerOptionalParameters(key, value)

Device Steps

The table bellow encopasses the documentation for all steps related to device nodes, as presented in the data model section.

Description	DSL Code
Returns a given device, if a device with the provided deviceIp exist. Also, if deviceIp is null, all devices will be returned. deviceIp: String, null	g.V.device(deviceIp)
Returns the list of all devices inside a given area, if an area with the provided areaName exist. Also, if areaName is null, all devices inside areas will be returned areaName: String, null	g.V.devicePerArea(areaName)
Returns the list of all devices, which already measured a given measurementVariableName (e.g. Temperature, Humidity). Also, if measurementVariableName is null, all devices that have measurement variables will be returned. measurementVariableName: String, null	g.V.devicePerMeasurementVariable(measurementVariableName)
Returns the list of all devices, which are part of a given category called categoryName. Also, if categoryName is null, all devices inside areas will be returned. categoryName: String, null	g.V.devicePerCategory(categoryName)
Returns the list of all devices that has the specified key-value pair, as part of their optional parameters. In case no value is specified, devices containing the provided key, despite their value, will be returned. key: String value: String, Int, Float, null	g.V.devicePerOptionalParameters(key, value)

Category Steps

The table bellow encopasses the documentation for all steps related to category nodes, as presented in the data model section.

Description	DSL Code
Returns a given category, if a category with the provided categoryName exist. Also, if categoryName is null, all categories will be returned. categoryName: String, null	g.V.category(categoryName)

Measurement Variable Steps

The table bellow encopasses the documentation for all steps related to measurement variable nodes, as presented in the data model section.

Description	DSL Code
Returns the list of all measurement variables, which are measured by devices inside a given area called areaName. Also, if areaName is null, all last measurements, despite the area it was measured, will be returned. areaName: String, null	g.V.measurementVariablePerArea(areaName)
Returns the list of all measurement variables, which are measured by a device with the provided deviceIp. deviceIp: String	g.V.measurementVariablePerDevice(deviceIp)

Measurement Steps

The table bellow encopasses the documentation for all steps related to measurement nodes, as presented in the data model section.

Description	DSL Code
Returns the list of all last measurements, up to one per device, in case it was measured inside a given area called areaName. Also, if areaName is null, all last measurements from devices inside areas will be returned. `areaName: String, null`	g.V.measurementFromArea(areaName)
Returns the last measurement from the device with the provided deviceIp. Also, if deviceIp is null, all last measurements, despite the device, will be returned. `deviceIp: String, null`	g.V.measurementFromDevice(deviceIp)
Returns the list of all last measurements, up to one per device, in case it measured a given measurementVariableName (e.g. Temperature, Humidity). Also, if measurementVariableName is null, all last measurements, despite the measurement variable, will be returned. measurementVariableName: String, null	g.V.measurementFromMeasurementVariable(measurementVariableName)

3.2.1 Examples

This subsection contains examples of the aforementioned steps. They were made using Blueprints API over HTTP/REST directly, as provided by Rexster. Alternatively, one could use libraries that support the Blueprints API, like PyBulbs [18] for Python. For higher throughput, one should favour a library that implements RexPro binary protocol, due to its smaller footprint when in comparison with Blueprints API over HTTP/REST.

Area Steps

This subsection encopasses examples of HTTP/REST calls for all steps related to area nodes. They can be reproduced in any web browser.

```
GET /graphs/graph/tp/gremlin?script=g.V.area() HTTP/1.1
Host: impress-storage.cloudapp.net:8182
```

```
{
   "results": [
      {
         "Name": "UFPE",
         "Type": "Area",
         "_id": 25601792,
         "_type": "vertex"
      },
      {
         "Name": "Theater UFPE",
         "Type": "Area",
         "_id": 25602048,
         "_type": "vertex"
      },
      {
         "Name": "CIN",
         "Type": "Area",
         "_id": 25602304,
         "_type": "vertex"
      },
      {
         "Name": "Room D001",
         "Type": "Area",
         "_id": 25602560,
         "_type": "vertex"
      },
      {
         "Name": "Room D002",
         "Type": "Area",
         "_id": 25602816,
         "_type": "vertex"
      },
      {
        "Name": "Room D003",
        "Type": "Area",
        "_id": 25603072,
        "_type": "vertex"
    },
    {
        "Name": "Lab Grad-1",
        "Type": "Area",
        "_id": 25603328,
        "_type": "vertex"
    },
    {
        "Name": "DINE",
        "Type": "Area",
        "_id": 25603584,
        "_type": "vertex"
    },
    {
        "Name": "GPRT",
        "Type": "Area",
        "_id": 25603840,
        "_type": "vertex"
    },
    {
        "Name": "IMPReSS Room",
        "Type": "Area",
        "_id": 25604096,
        "_type": "vertex"
    },
    {
        "Name": "Witec Room",
        "Type": "Area",
        "_id": 25604352,
        "_type": "vertex"
    },
    {
        "Name": "Secfunet Room",
        "Type": "Area",
        "_id": 25604608,
            "_type": "vertex"
        },
        {
            "Name": "Electronics Lab",
            "Type": "Area",
            "_id": 25604864,
            "_type": "vertex"
        }
    ],
    "success": true,
    "version": "2.5.0",
    "queryTime": 26.4055
}
```

GET /graphs/graph/tp/gremlin?script=g.V.area('GPRT') HTTP/1.1

```
Host: impress-storage.cloudapp.net:8182
```

```
{
    "results": [
        {
            "Name": "GPRT",
            "Type": "Area",
            "_id": 25603840,
            "_type": "vertex"
        }
    ],
    "success": true,
    "version": "2.5.0",
    "queryTime": 11.8112
}
```

GET /graphs/graph/tp/gremlin?script=g.V.areaPerMeasurementVariable('Power') HTTP/1.1

```
Host: impress-storage.cloudapp.net:8182
```

```
{
    "results": [
        {
            "Name": "UFPE",
            "Type": "Area",
            "_id": 25601792,
    "_type": "vertex"
  },
  {
    "Name": "Theater UFPE",
    "Type": "Area",
    "_id": 25602048,
    "_type": "vertex"
  },
  {
    "Name": "CIN",
    "Type": "Area",
    "_id": 25602304,
    "_type": "vertex"
  },
  {
    "Name": "Room D001",
    "Type": "Area",
    "_id": 25602560,
    "_type": "vertex"
  },
  {
    "Name": "Room D002",
    "Type": "Area",
    "_id": 25602816,
    "_type": "vertex"
  },
  {
    "Name": "Room D003",
    "Type": "Area",
    "_id": 25603072,
    "_type": "vertex"
  },
  {
    "Name": "Lab Grad-1",
    "Type": "Area",
    "_id": 25603328,
    "_type": "vertex"
  },
  {
        "Name": "DINE",
        "Type": "Area",
        "_id": 25603584,
        "_type": "vertex"
      },
      {
        "Name": "GPRT",
        "Type": "Area",
        "_id": 25603840,
        "_type": "vertex"
      },
      {
        "Name": "IMPReSS Room",
        "Type": "Area",
        "_id": 25604096,
        "_type": "vertex"
      },
      {
        "Name": "Witec Room",
        "Type": "Area",
        "_id": 25604352,
        "_type": "vertex"
      },
      {
        "Name": "Secfunet Room",
        "Type": "Area",
        "_id": 25604608,
        "_type": "vertex"
      },
      {
        "Name": "Electronics Lab",
        "Type": "Area",
        "_id": 25604864,
        "_type": "vertex"
      }
    ],
    "success": true,
    "version": "2.5.0",
    "queryTime": 25.8147
```

```
}
```

GET /graphs/graph/tp/gremlin?script=g.V.areaPerArea('GPRT') HTTP/1.1

```
Host: impress-storage.cloudapp.net:8182
```

```
{
   "results": [
      {
         "Name": "DINE",
         "Type": "Area",
         "_id": 25603584,
         "_type": "vertex"
      }
   ],
   "success": true,
   "version": "2.5.0",
   "queryTime": 20.6975
}
```

Device Steps

This subsection encopasses examples of HTTP/REST calls for all steps related to device nodes. They can be reproduced in any web browser.

GET /graphs/graph/tp/gremlin?script=g.V.device() HTTP/1.1

```
Host: impress-storage.cloudapp.net:8182
```

```
{
   "results": [
      {
         "OptionalParameters": {
            "Name": "Light11",
            "Consumption": "20 W",
            "Reference": "GE Light Bulb",
            "Type": "IT"
         },
         "Type": "Device",
         "IP": "192.168.1.1",
         "_id": 25607168,
         "_type": "vertex"
      },
      {
            "OptionalParameters": {
                "Name": "Light2x",
                "Consumption": "30 W",
                "Reference": "GE Light Bulb",
                "Type": "IT"
            },
            "Type": "Device",
            "IP": "192.168.1.2",
            "_id": 25614848,
            "_type": "vertex"
        },
        ...
    ],
    "success": true,
    "version": "2.5.0",
    "queryTime": 104.996799
}
```

```
GET /graphs/graph/tp/gremlin?script=g.V.device('192.168.3.7') HTTP/1.1
Host: impress-storage.cloudapp.net:8182
```

```
{
    "results": [
        {
            "OptionalParameters": {
                "Name": "Projector GG",
                "Consumption": "200 W",
                "Reference": "Sony Projector",
                "Type": "IT"
            },
            "Type": "Device",
            "IP": "192.168.3.7",
            "_id": 25714688,
            "_type": "vertex"
        }
    ],
    "success": true,
    "version": "2.5.0",
    "queryTime": 8.9046
}
```

GET /graphs/graph/tp/gremlin?script=g.V.devicePerArea('GPRT') HTTP/1.1

```
Host: impress-storage.cloudapp.net:8182
```

```
{
   "results": [
      {
         "OptionalParameters": {
            "Name": "Light1xxx",
            "Consumption": "20 W",
            "Reference": "GE Light Bulb",
            "Type": "IT"
         },
         "Type": "Device",
         "IP": "192.169.2.1",
         "_id": 25799168,
         "_type": "vertex"
      },
      {
         "OptionalParameters": {
            "Name": "Light2 XSA",
            "Consumption": "30 W",
            "Reference": "GE Light Bulb",
            "Type": "IT"
         },
         "Type": "Device",
         "IP": "192.169.2.2",
         "_id": 25806848,
         "_type": "vertex"
      },
      ...
   ],
   "success": true,
   "version": "2.5.0",
   "queryTime": 60.2657
}
```

GET /graphs/graph/tp/gremlin?script=g.V.devicePerMeasurementVariable('Power') HTTP/1.1

```
Host: impress-storage.cloudapp.net:8182
```

```
{
   "results": [
      {
         "OptionalParameters": {
            "Name": "Light11",
            "Consumption": "20 W",
            "Reference": "GE Light Bulb",
            "Type": "IT"
         },
         "Type": "Device",
         "IP": "192.168.1.1",
         "_id": 25607168,
         "_type": "vertex"
      },
      ...
   ],
   "success": true,
   "version": "2.5.0",
   "queryTime": 114.5046
}
```

Category Steps

This subsection encopasses examples of HTTP/REST calls for all steps related to category nodes. They can be reproduced in any web browser.

```
GET /graphs/graph/tp/gremlin?script=g.V.category() HTTP/1.1
Host: impress-storage.cloudapp.net:8182
```

```
{
   "results": [
      {
         "Name": "IT",
         "Type": "Category",
         "_id": 25606144,
         "_type": "vertex"
      },
      {
         "Name": "HVAC",
            "Type": "Category",
            "_id": 25606400,
            "_type": "vertex"
        },
        {
            "Name": "Illumination",
            "Type": "Category",
            "_id": 25606656,
            "_type": "vertex"
        },
        {
            "Name": "Audiovisual",
            "Type": "Category",
            "_id": 25606912,
            "_type": "vertex"
        }
    ],
    "success": true,
    "version": "2.5.0",
    "queryTime": 13.6438
}
```

```
GET /graphs/graph/tp/gremlin?script=g.V.category('IT') HTTP/1.1
Host: impress-storage.cloudapp.net:8182
```

```
{
    "results": [
        {
            "Name": "IT",
            "Type": "Category",
            "_id": 25606144,
            "_type": "vertex"
        }
    ],
    "success": true,
    "version": "2.5.0",
    "queryTime": 14.3249
}
```

Measurement Variable Steps

This subsection encopasses examples of HTTP/REST calls for all steps related to measurement variable nodes. They can be reproduced in any web browser.

```
GET /graphs/graph/tp/gremlin?script=g.V.measurementVariablePerDevice('192.168.3.7') HTTP/1.1
Host: impress-storage.cloudapp.net:8182
```

```
{
   "results": [
      {
         "Type": "Power",
         "Unit": "Watts",
         "_id": 25605632,
         "_type": "vertex"
      }
   ],
   "success": true,
   "version": "2.5.0",
   "queryTime": 18.0474
}
```

```
GET /graphs/graph/tp/gremlin?script=g.V.measurementVariablePerArea('GPRT') HTTP/1.1
Host: impress-storage.cloudapp.net:8182
```

```
{
   "results": [
      {
         "Type": "Power",
         "Unit": "Watts",
         "_id": 25605632,
         "_type": "vertex"
      }
   ],
   "success": true,
   "version": "2.5.0",
   "queryTime": 19.4948
}
```

Measurement Steps

This subsection encopasses examples of HTTP/REST calls for all steps related to measurement nodes. They can be reproduced in any web browser.

```
GET /graphs/graph/tp/gremlin?script=g.V.measurementFromArea('GPRT') HTTP/1.1
Host: impress-storage.cloudapp.net:8182
```

```
{
   "results": [
      {
         "Type": "Measurement",
         "Power": 125,
         "_id": 25716224,
         "_type": "vertex"
      },
      {
         "Type": "Measurement",
         "Power": 124,
         "_id": 25716480,
         "_type": "vertex"
      },
      ...
   ],
   "success": true,
   "version": "2.5.0",
   "queryTime": 71.2574
}
```

```
GET /graphs/graph/tp/gremlin?script=g.V.measurementFromDevice('192.168.3.7') HTTP/1.1
Host: impress-storage.cloudapp.net:8182
```

```
{
   "results": [
      {
         "Type": "Measurement",
         "Power": 155,
         "_id": 25714944,
         "_type": "vertex"
      }
   ],
   "success": true,
   "version": "2.5.0",
   "queryTime": 47.1994
```

}

GET /graphs/graph/tp/gremlin?script=g.V.measurementFromMeasurementVariable('Temperature') HTTP/1.1

```
Host: impress-storage.cloudapp.net:8182
```

```
{
   "results": [],
   "success": true,
   "version": "2.5.0",
   "queryTime": 47.1994
}
```

3.2.2 Definitions

This subsection contains the Groovy code that implements the aforementioned steps. They must be executed every time Rexster is started, for the DSL's steps to be available.

Area Steps

```
/**
* Returns a given area, if an area with the provided areaName
* exist. Also, if areaName is null, all areas will be returned.
*
*    areaName: String, null
*/
GremlinGroovyPipeline.metaClass.area = {
    areaName -> _().ifThenElse{areaName == null}
    {delegate.has('Type','Area')}
    {delegate.has('Name',areaName).has("Type","Area")}
}

/**
* Returns an area, if it contains a device with the provided
* deviceIp inside. Also, if deviceIp is null, all areas that
* contains devices will be returned.
*
*    deviceIp: String, null
*/
GremlinGroovyPipeline.metaClass.areaPerDevice = {
    deviceIp -> _().ifThenElse{deviceIp == null}
    {delegate.has('Type','Device').in('has')}
    {delegate.has('IP',deviceIp).in('has')}
}

/**
* Returns a given area, if an area with the provided areaName
* is inside it. Also, if areaName is null, all areas with parents
* will be returned.
*
*    areaName: String, null
*/
GremlinGroovyPipeline.metaClass.areaPerArea = {
    areaName -> _().ifThenElse{areaName == null}
    {delegate.has('Type','Area').in('has')}
    {delegate.has('Type','Area').has('Name',areaName).in('has')}
}

/**
* Returns the list of all areas, which contains devices that had
* already measured a given measurementVariableName (e.g. Temperature,
* Humidity). Also, if measurementVariableName is null, all areas with
* devices that have measurement variables will be returned.
*
*    measurementVariableName: String, null
*/
GremlinGroovyPipeline.metaClass.areaPerMeasurementVariable = {
    measurementVariableName -> _().ifThenElse{measurementVariableName == null}
    {delegate.has('Type',it.Type).out.areaPerDevice}
    {delegate.has('Type',measurementVariableName).out.areaPerDevice.unique()}
}

/**
* Returns the list of all areas that has the specified key-value pair,
* as part of their optional parameters. In case no value is specified,
* areas containing the provided key, despite their value, will be
* returned.
*
*    key: String
*    value: String, Int, Float, null
*/
GremlinGroovyPipeline.metaClass.areaPerOptionalParameters = {
    key,value=null -> _().ifThenElse{value == null}
    {
        delegate.has("Type","Area").has("OptionalParameters")
        .ifThenElse{it.OptionalParameters[key]!=null}
        {it}{null}.except([null])
    }
    {
        delegate.has("Type","Area").has("OptionalParameters")
        .ifThenElse{it.OptionalParameters[key]==value}
        {it}{null}.except([null])
    }
}
```

Device Steps

```
/**
* Returns a given device, if a device with the provided deviceIp exist.
* Also, if deviceIp is null, all devices will be returned.
*
*    deviceIp: String, null
*/
GremlinGroovyPipeline.metaClass.device = {
    deviceIp -> _().ifThenElse{deviceIp == null}
    {delegate.has('Type','Device')}
    {delegate.has('IP',deviceIp)}
}

/**
* Returns the list of all devices inside a given area, if an area with
* the provided areaName exist. Also, if areaName is null, all devices
* inside areas will be returned.
*
*    areaName: String, null
*/
GremlinGroovyPipeline.metaClass.devicePerArea = {
    areaName ->_().ifThenElse{areaName == null}
    {delegate.has('Type','Area').has('Name',it.Name).out.has('Type','Device')}
    {delegate.has('Type','Area').has('Name',areaName).as('x').out().loop('x')
    {it.object.Type != "Device"}}
}

/**
* Returns the list of all devices, which already measured a given
* measurementVariableName (e.g. Temperature, Humidity). Also, if
* measurementVariableName is null, all devices that have measurement
* variables will be returned.
*
*    measurementVariableName: String, null
*/
GremlinGroovyPipeline.metaClass.devicePerMeasurementVariable = {
    measurementVariableName -> _().ifThenElse{measurementVariableName == null}
    {delegate.has('Type',it.Type).out}
    {delegate.has('Type',measurementVariableName).out}
}

/**
* Returns the list of all devices, which are part of a given category
* called categoryName. Also, if categoryName is null, all devices
* inside areas will be returned.
*
*    categoryName: String, null
*/
GremlinGroovyPipeline.metaClass.devicePerCategory = {
    categoryName ->_().ifThenElse{categoryName == null}
    {delegate.has('Type','Category').out.has('Type','Device')}

{delegate.has('Name',categoryName).has('Type','Category').out.has('Type','Device
')}
}

/**
* Returns the list of all devices that has the specified key-value pair,
* as part of their optional parameters. In case no value is specified,
* devices containing the provided key, despite their value, will be
* returned.
*
*    key: String
*    value: String, Int, Float, null
*/
GremlinGroovyPipeline.metaClass.devicePerOptionalParameters = {
    key,value=null -> _().ifThenElse{value == null}
    {
        delegate.has("Type","Device").has("OptionalParameters")
        .ifThenElse{it.OptionalParameters[key]!=null}
        {it}{null}.except([null])
    }
    {
        delegate.has("Type","Device").has("OptionalParameters")
        .ifThenElse{it.OptionalParameters[key]==value}
        {it}{null}.except([null])
    }
}
```

Category Steps

```
/**
* Returns a given category, if a category with the provided categoryName
* exist. Also, if categoryName is null, all categories will be returned.
*
*    categoryName: String, null
*/
GremlinGroovyPipeline.metaClass.category = {
    categoryName -> _().ifThenElse{categoryName == null}
    {delegate.has('Type','Category')}
    {delegate.has('Name',categoryName).has('Type','Category')}
}
```

Measurement Variable Steps

```
/**
* Returns the list of all measurement variables, which are measured by
* devices inside a given area called areaName. Also, if areaName is null,
* all last measurements, despite the area it was measured, will be returned.
*
*    areaName: String, null
*/
GremlinGroovyPipeline.metaClass.measurementVariablePerArea = {
    areaName -> _().ifThenElse{areaName == null}

{delegate.has('Type','Area').has('Name',it.Name).devicePerArea.measurementVariab
lePerDevice.unique()}

{delegate.has('Type','Area').has('Name',areaName).devicePerArea.measurementVaria
blePerDevice.unique()}
}

/**
* Returns the list of all measurement variables, which are measured by
* a device with the provided deviceIp.
*
*    deviceIp: String
*/
GremlinGroovyPipeline.metaClass.measurementVariablePerDevice = {
    deviceIp -> _().ifThenElse{deviceIp == null}
    {delegate.has('Type','Device').has('IP',it.IP).in('interacts')}
    {delegate.has('Type','Device').has('IP',deviceIp).in('interacts')}
}
```

Measurement Steps

```
/**
* Returns the list of all last measurements, up to one per device, in
* case it was measured inside a given area called areaName. Also, if
* areaName is null, all last measurements from devices inside areas
* will be returned.
*
*    areaName: String, null
```

```
*/
GremlinGroovyPipeline.metaClass.measurementsFromArea = {
    areaName ->_().ifThenElse{areaName == null}
    {delegate.has('Type','Area').has('Name',it.Name).out.has('Type','Device').
out('was measured')}

{delegate.has('Type','Area').has('Name',areaName).out.has('Type','Device').out('
was measured')}
}

/**
* Returns the last measurement from the device with the provided
* deviceIp. Also, if deviceIp is null, all last measurements, despite
* the device, will be returned.
*
*    deviceIp: String, null
*/
GremlinGroovyPipeline.metaClass.measurementFromDevice = {
    deviceIp ->_().ifThenElse{deviceIp == null}
    {delegate.has('Type','Device').has('IP',it.IP).out('was measured')}
    {delegate.has('Type','Device').has('IP',deviceIp).out('was measured')}
}

/**
* Returns the list of all last measurements, up to one per device,
* in case it measured a given measurementVariableName (e.g. Temperature,
* Humidity). Also, if measurementVariableName is null, all last
* measurements, despite the measurement variable, will be returned.
*
*    measurementVariableName: String, null
*/
GremlinGroovyPipeline.metaClass.measurementFromMeasurementVariable = {
    measurementVariableName ->_().ifThenElse{measurementVariableName == null}

{delegate.has('Type',it.Type).devicePerMeasurementVariable.measurementFromDevice
}

{delegate.has('Type',measurementVariableName).devicePerMeasurementVariable.measu
rementFromDevice}
}
```

3.3 IMPReSS' Storage DSL Functions

Different from steps, functions cannot be chained in a single call. In other words, prefix and postfix notations cannot be used.

Function notation:
function(parameter1, parameter2, ...)

Create Functions

The table bellow encopasses the documentation for all functions related to the creation of the different nodes present in the data model section.

Description	DSL Code
Creates a new area, called areaName, with no parent area nor optional parameters. g: the instance of the graph areaName: String	createArea(g, areaName)
Creates a new area, called areaName, which will be contained inside an area called parentAreaName, with no optional parameters. g: the instance of the graph name: String parentAreaName: String deviceIp: String, null	createArea(g, areaName, parentAreaName)
Creates a new area, called areaName, containing the key-value pairs of optionalParameters, with no parent area. g: the instance of the graph areaName: String optionalParameters: Map <String,Int>, <String,String>, <String,Float>	createArea(g, areaName, optionalParameters)
Creates a new area, called areaName, containing the key-value pairs of optionalParameters, which will be contained inside an area called parentAreaName. g: the instance of the graph areaName: String parentAreaName: String optionalParameters: Map <String,Int>, <String,String>, <String,Float>	createArea(g, areaName, parentAreaName, optionalParameters)
Creates a new device, identified by the network address provided in deviceIp. The device capable of measuring all measurement variables specified in measurementVariableNames. Despite that, this device, will have no optional parameters, parent area or categories. g: the instance of the graph deviceIp: String measurementVariableNames: List of String, Int, Float	createDevice(g, deviceIp, measurementVariableNames)
Creates a new device, containing the key-value pairs of optionalParameters. This device will be	createDevice(g, deviceIp,

Description	DSL Code
identified by the network address provided in deviceIp. Also, it will be capable of measuring all measurement variables specified in measurementVariableNames. Despite that, this device, will have no parent area or categories. g: the instance of the graph deviceIp: String measurementVariableNames: List of String, Int, Float optionalParameters: Map <String,Int>, <String,String>, <String,Float>	measurementVariableNames, optionalParameters)
Creates a new device, containing the key-value pairs of optionalParameters. This device will be identified by the network address provided in deviceIp. Also, it will be capable of measuring all measurement variables specified in measurementVariableNames, have a parent area and categories. g: the instance of the graph deviceIp: String measurementVariablesNames: List of String, Int, Float parentAreaName: String categoryName: String optionalParameters: Map <String,Int>, <String,String>, <String,Float>	createDevice(g, deviceIp, measurementVariableNames, parentAreaName, categoryName, optionalParameters)
Creates a new category called categoryName. g: the instance of the graph categoryName: String	createCategory(g, categoryName)
Creates a new measurement variable, called measurementVariableName (e.g. Temperature, Humidity, Light Intensity), measured in a given unit (e.g. Celsius, Watt) called unitName. g: the instance of the graph measurementVariableName: String unitName: String	createMeasurementVariable(g, measurementVariableName, unitName)
Creates a new measurement for a given device with the provided deviceIp. Values contains a set of key-value pairs, where keys represent the name of the measured variable and values are	createMeasurement(g, deviceIp, values)

Description	DSL Code
the respective measurement from a sensor, for instance. g: the instance of the graph deviceIp: String values: Map <String,Int>, <String,String>, <String,Float>	

Chart Helpers Functions

The table bellow encopasses the documentation for functions that focus on returning a set of historical measurements, in order to facilitate application developers to plot timeseries.

Description	DSL Code
Returns measurements, from the device with the provided deviceIp, measured between beginTimestamp and endTimestamp. g: the instance of the graph deviceIp: String beginTimestamp: String endTimestamp: String	measurementsBetweenTimestamps(g, deviceIp, beginTimestamp, endTimestamp)
Returns a fixed amount of last measurements from the device with the provided deviceIp. The number of measurements is dictated by numTicks. g: the instance of the graph deviceIp: String numTicks: Unsigned Int	measurementsPerTicks(g, deviceIp, numTicks)
Returns measurements, from the device with the provided deviceIp, measured between beginTimestamp and endTimestamp. However, if numTicks is less than the amount of available measurements for the specified device, slices will be made based on the provided time range, so that the mean of the measurements, for each slice, will be returned instead of individual samples. g: the instance of the graph deviceIp: String numTicks: Unsigned Int beginTimestamp: String endTimestamp: String	MeasurementsPerTicksAndTimestamps(g, deviceIp, numTicks, beginTimestamp, endTimestamp)

3.3.1 Examples

This subsection contains examples of the aforementioned functions. They were made using Blueprints API over HTTP/REST directly, as provided by Rexster. Alternatively, one could use libraries that support the Blueprints API, like PyBulbs [18] for Python. For higher throughput, one should favour a library that implements RexPro binary protocol, due to its smaller footprint when in comparison with Blueprints API over HTTP/REST.

Chart Helpers Functions

This subsection encopasses examples of HTTP/REST calls for all chart helpers' functions.

```
GET /graphs/graph/tp/gremlin?
script=measurementsBetweenTimestamps(g,'192.168.3.7','02/21/2015', '02/22/2015') HTTP/1.1
```

```
Host: impress-storage.cloudapp.net:8182
```

```
{
   "results": [
      {
         "Node": {
            "Type": "Measurement",
            "Power": 155,
            "_id": 25714944,
            "_type": "vertex"
         },
         "Timestamp": "Sun Feb 22 22:07:15 UTC 2015"
      },
      {
         "Node": {
            "Type": "Measurement",
            "Power": 48,
            "_id": 25715200,
            "_type": "vertex"
         },
         "Timestamp": "Sun Feb 22 16:07:15 UTC 2015"
      },
      {
         "Node": {
            "Type": "Measurement",
            "Power": 71,
            "_id": 25715456,
            "_type": "vertex"
         },
         "Timestamp": "Sun Feb 22 07:07:15 UTC 2015"
```

```
        },
        {
            "Node": {
                "Type": "Measurement",
                "Power": 33,
                "_id": 25715712,
                "_type": "vertex"
            },
            "Timestamp": "Sat Feb 21 22:07:15 UTC 2015"
        },
        {
            "Node": {
                "Type": "Measurement",
                "Power": 142,
                "_id": 25715968,
                "_type": "vertex"
            },
            "Timestamp": "Sat Feb 21 18:07:15 UTC 2015"
        },
        {
            "Node": {
                "Type": "Measurement",
                "Power": 125,
                "_id": 25716224,
                "_type": "vertex"
            },
            "Timestamp": "Sat Feb 21 06:07:15 UTC 2015"
        }
    ],
    "success": true,
    "version": "2.5.0",
    "queryTime": 50.3831
}
```

```
GET /graphs/graph/tp/gremlin?script=measurementsPerTicks(g,'192.168.3.7', 2) HTTP/1.1
Host: impress-storage.cloudapp.net:8182
```

```
{
    "results": [
        {
            "Type": "Measurement",
            "Power": 117,
            "_id": 25722112,
            "_type": "vertex"
        },
        {
            "Type": "Measurement",
            "Power": 116,
            "_id": 25721856,
            "_type": "vertex"
        }
    ],
    "success": true,
    "version": "2.5.0",
    "queryTime": 26.3154
}
```

GET /graphs/graph/tp/gremlin?script=measurementsPerTicksAndTimestamps(g,'192.168.3.7',5,'02/21/2015', '02/22/2015') HTTP/1.1

```
Host: impress-storage.cloudapp.net:8182
```

```
{
    "results": [
        {
            "Node": {
                "Power": 155,
                "Type": "Measurement"
            },
            "Timestamp": "Sun Feb 22 22:07:15 UTC 2015"
        },
        {
            "Node": {
                "Power": 48,
                "Type": "Measurement"
            },
            "Timestamp": "Sun Feb 22 16:07:15 UTC 2015"
        },
        {
            "Node": {
                "Power": 71,
                "Type": "Measurement"
            },
            "Timestamp": "Sun Feb 22 07:07:15 UTC 2015"
        },
        {
            "Node": {
                "Power": 33,
                "Type": "Measurement"
            },
            "Timestamp": "Sat Feb 21 22:07:15 UTC 2015"
        },
        {
            "Node": {
                "Power": 142,
                "Type": "Measurement"
            },
            "Timestamp": "Sat Feb 21 18:07:15 UTC 2015"
        }
    ],
    "success": true,
    "version": "2.5.0",
    "queryTime": 33.9015
}
```

3.3.1 Definitions

This subsection contains the Groovy code that implements the aforementioned functions. They must be executed every time Rexster is started, for the DSL's functions to be available.

Create Functions

```
/**
* Creates a new measurement variable, called measurementVariableName
* (e.g. Temperature, Humidity, Light Intensity), measured in a
* given unit (e.g. Celsius, Watt) called unitName.
*
*    g: the instance of the graph
*    measurementVariableName: String
*    unitName: String
*/
def createMeasurementVariable(g,measurementVariableName,unitName){
    mVariable = g.addVertex([Type:measurementVariableName,Unit:unitName])
    g.commit()
    return mVariable
}

/**
* Creates a new category called categoryName.
*
*    g: the instance of the graph
*    categoryName: String
*/
def createCategory(g,categoryName){
    category = g.addVertex([Type:'Category',Name:categoryName])
    g.commit()
    return category
}

/**
* Creates a new measurement for a given device with the provided
* deviceIp. Values contains a set of key-value pairs, where keys
* represent the name of the measured variable and values are the
* respective measurement from a sensor, for instance.
*
*    g: the instance of the graph
*    deviceIp: String
*    values: Map <String,Int>, <String,String>, <String,Float>
*/
def createMeasurement(g, deviceIp, values){
    measure = [Type:"Measurement"]

    for(value in values){
        measure[value.key] = value.value.toFloat()
    }

    measureVertex = g.addVertex(measure)
    try{
        device = g.V.has("IP",deviceIp).next()
    }
    catch(FastNoSuchElementException e){
        measureVertex.remove()
        throw new DeviceNotFoundException(deviceIp)
    }

    try{
        devLastMeasurement = g.V.measurementFromDevice(deviceIp).next()
        newEdge = g.addEdge(device, measureVertex,
                            "was measured", [timestamp:(new Date()).toString()])

        oldEdge = devLastMeasurement.inE().next()
        oldTimestamp = oldEdge.timestamp

        g.addEdge(measureVertex, devLastMeasurement,
                    "was measured", [timestamp:oldTimestamp])
        oldEdge.remove()

    }catch(FastNoSuchElementException e){
            edge = g.addEdge(device, measureVertex,
                            "was measured", [timestamp:(new Date()).toString()])
    }
    g.commit()
    return measureVertex
```

```
}

/**
* Creates a new device, containing the key-value pairs of
* optionalParameters. This device will be identified by the
* network address provided in deviceIp. Also, it will be capable
* of measuring all measurement variables specified in
* measurementVariableNames. Despite that, this device, will have
* no parent area or categories.
*
*    g: the instance of the graph
*    deviceIp: String
*    measurementVariableNames: List of String, Int, Float
*    optionalParameters: Map <String,Int>, <String,String>, <String,Float>
*/
def createDevice(g, String deviceIp, List measurementVariableNames,
                optionalParameters=[]){
    devProps = [Type:'Device']
    devProps['IP'] = deviceIp

    device = g.addVertex(devProps)

    for(type in measurementVariableNames){
        try{
           vertex = g.V.has("Type",type).next()
        }
        catch(FastNoSuchElementException e){
            device.remove()
            throw new MeasurementVariableNotFoundException(type)
        }
        g.addEdge(vertex,device,'interacts')
    }
    if(optionalParameters){
        m=[:]
        for(i in optionalParameters){
            try{
                optionalParameters[i.key].toFloat()
                m[i.key]=optionalParameters[i.key].toFloat()
            }
            catch(NumberFormatException e){
                m[i.key]=optionalParameters[i.key]
            }
        }
        device["OptionalParameters"] = m
    }
    g.commit()
    return device
}

/**
* Creates a new device, containing the key-value pairs of optionalParameters.
* This device will be identified by the network address provided in deviceIp.
* Also, it will be capable of measuring all measurement variables specified in
* measurementVariableNames, have a parent area and categories.
*
*    g: the instance of the graph
*    deviceIp: String
*    measurementVariables: List of String, Int, Float
*    parentAreaName: String
*    categoryName: String
*    optionalParameters: Map <String,Int>, <String,String>, <String,Float>
*/
def createDevice(g, String deviceIp, List measurementVariableNames,
                String parentAreaName, String categoryName,
                optionalParameters=[]){

    devProps = [Type:'Device']
    devProps['IP'] = deviceIp

    try{
        category = g.V.category(categoryName).next()
    }
    catch(FastNoSuchElementException e){
        throw new CategoryNotFoundException(categoryName)
    }
    try{
        parentArea = g.V.area(parentAreaName).next()
    }
    catch(FastNoSuchElementException e){
        throw new AreaNotFoundException(parentAreaName)
    }

    device = createDevice(g, devProps["IP"],
                          measurementVariableNames, optionalParameters)

    g.addEdge(category, device, "comprehends")
    g.addEdge(parentArea, device, 'has')

    g.commit()
    return device
}

/**
* Creates a new area, called areaName, which will be contained
* inside an area called parentAreaName, with no optional parameters.
*
*    g: the instance of the graph
*    name: String
*    parentAreaName: String
*/
def createArea(g, areaName, String parentAreaName=""){
    area = g.addVertex([Type:'Area',Name:areaName])

    if(parentAreaName.length() > 0){
        try{
            g.addEdge(g.V.area(parentAreaName).next(), area, "has")
        }
        catch(FastNoSuchElementException e){
            area.remove()
            throw new AreaNotFoundException(parentAreaName)
        }
    }
    g.commit()

    return area
}

/**
```

```
* Creates a new area, called areaName, containing the key-value pairs
* of optionalParameters, which will be contained inside an area called
* parentAreaName.
*
*    g: the instance of the graph
*    areaName: String
*    parentAreaName: String
*    optionalParameters: Map <String,Int>, <String,String>, <String,Float>
*/
def createArea(g, areaName, parentAreaName, optionalParameters=[]){
    area = g.addVertex([Type:'Area',Name:areaName])

    if(parentAreaName.length() > 0){
        try{
            g.addEdge(g.V.area(parentAreaName).next(), area, "has")
        }
        catch(FastNoSuchElementException e){
            area.remove()
            throw new AreaNotFoundException(parentAreaName)
        }
    }
    if(optionalParameters){
        m=[:]
        for(i in optionalParameters){
            try{
                optionalParameters[i.key].toFloat()
                m[i.key]=optionalParameters[i.key].toFloat()
            }
            catch(NumberFormatException e){
                m[i.key]=optionalParameters[i.key]
            }
        }
        area["OptionalParameters"] = m
    }
    g.commit()

    return area
}

/**
* Creates a new area, called areaName, containing the key-value
* pairs of optionalParameters, with no parent area.
*
*    g: the instance of the graph
*    areaName: String
*    optionalParameters: Map <String,Int>, <String,String>, <String,Float>
*/
def createArea(g, areaName, Map optionalParameters){
    area = g.addVertex([Type:'Area',Name:areaName])

    if(optionalParameters){
        m=[:]
        for(i in optionalParameters){
            try{
                optionalParameters[i.key].toFloat()
                m[i.key]=optionalParameters[i.key].toFloat()
            }
            catch(NumberFormatException e){
                m[i.key]=optionalParameters[i.key]
            }
        }
        area["OptionalParameters"] = m
    }
    g.commit()

    return area
}
```

Chart Helpers Functions

```
/**
* Returns a fixed amount of last measurements from the device
* with the provided deviceIp. The number of measurements is
* dictated by numTicks.
*
*    g: the instance of the graph
*    deviceIp: String
*    numTicks: Unsigned Int
*/
def measurementsPerTicks(g, deviceIp, numTicks){
    g.V.has("IP",deviceIp)
    .as('x').out("was measured")
    .loop('x'){it.loops<=numTicks}{true}.transform{
        ["Node":it,"Timestamp":it.inE.next().timestamp]
    }
}

/**
* Returns measurements, from the device with the provided deviceIp,
* measured between beginTimestamp and endTimestamp. However, if numTicks
* is less than the amount of available measurements for the specified
* device, slices will be made based on the provided time range, so that
* the mean of the measurements, for each slice, will be returned instead
* of individual samples.
*
*    g: the instance of the graph
*    deviceIp: String
*    numTicks: Unsigned Int
*    beginTimestamp: String
*    endTimestamp: String
*/
def measurementsPerTicksAndTimestamps(g, deviceIp, numTicks,
                                        beginTimestamp, endTimestamp){
    path = measurementsBetweenTimestamps(
                g,deviceIp,beginTimestamp,endTimestamp)

    if(path.getClass() == String || numTicks >= path.size()){
        return path
    }

    pathList=[]
    pathJumps=numTicks

    numVertices = path.size()

    numSamples = (numVertices/
                  numTicks).toInteger()

    newMeasurement = path[0].Node.map.next()
    newMeasurement = resetMeasurement(newMeasurement)

    node=0
    timestamp=""
    for(i=0; i<numTicks ;i++){
        for(j=0; j<numSamples ;j++){
                if(j==0){
                    timestamp = path[node].Timestamp
                }
                newMeasurement = incMeasurement(newMeasurement,path[node].Node)
                node++
        }
        newMeasurement = meanMeasurement(newMeasurement,numSamples)
        pathList+=["Node":newMeasurement.clone(),"Timestamp":timestamp]
        newMeasurement = resetMeasurement(newMeasurement)
    }
    return pathList
}

/**
* Returns measurements, from the device with the provided deviceIp,
* measured between beginTimestamp and endTimestamp.
*
*    g: the instance of the graph
*    deviceIp: String
*    beginTimestamp: String
*    endTimestamp: String
*/
def measurementsBetweenTimestamps(g,deviceIp,beginTimestamp,
                                endTimestamp,timestamps=true){
    beginDate = new Date(beginTimestamp)
    endDate = new Date(endTimestamp)
    device = null
    try{
        device = g.V.has("IP",deviceIp).next()
    }
    catch(FastNoSuchElementException e){
        throw new DeviceNotFoundException(deviceIp)
    }

    if(endDate < beginDate){
        throw new IllegalArgumentException(
                "beginTimestamp is older than endTimestamp"
            )
    }

    if( endDate.hours   == 0 &&
        endDate.minutes == 0 &&
        endDate.seconds == 0){
        endDate.hours   = 23
        endDate.minutes = 59
        endDate.seconds = 59
    }

    try{
        firstVertex =   device.as('x').out("was measured")
```

```
                        .loop('x'){
                            new Date(it.object.inE.map.next()
                            .timestamp.toString()) >= endDate
                        }.next()
    }catch(IllegalArgumentException e){
        throw new IllegalArgumentException("Invalid endTimestamp")
    }catch(FastNoSuchElementException e){
        throw new IllegalArgumentException("There is no measurements from "
                                        +endTimestamp+" or before.")
    }

    try{
        path = []
        for(i in firstVertex.in.as('x').out("was measured").loop('x'){
                new Date(it.object.inE.map.next()
                .timestamp.toString()) >= beginDate
            }
            {true}){

            if(timestamps == true){
                path.add(["Node":i,"Timestamp":i.inE.timestamp.next()])
            }else{
                path.add(i)
            }
        }
        path.pop()
    }catch(IllegalArgumentException e){
        throw new IllegalArgumentException("Invalid beginTimestamp")
    }

    return path
}

private def resetMeasurement(vertexMap){
    for(key in vertexMap.keySet()){
        if(key != "Type"){ vertexMap[key] = 0 }
    }
    return vertexMap
}

private def incMeasurement(vertexMap,newVertex){
    for(key in vertexMap.keySet()){
        if(key != "Type"){ vertexMap[key]+=newVertex[key]  }
    }
    return vertexMap
}

private def meanMeasurement(vertexMap,numSamples){
    for(key in vertexMap.keySet()){
        if(key != "Type"){ vertexMap[key] = vertexMap[key]/numSamples  }
    }
    return vertexMap
}
```

4. Initial Performance Evaluation

To evaluate the performance of our proposed architecture, we populated an instance of Titan with the data model proposed in Figure 2, via a Rexster Server. This is important since this evaluation can be envisaged as a real performance evaluation of the proposed architecture, not just an indiscriminate stress test. Especially, because query times depends on the data model used. We conducted the initial experiments using the following number of nodes:

- Devices: 30
- Measurement Histories per device: 500
- Areas: 7
- Measurement Variables: 4 (i.e. energy consumption, temperature, humidity and light intensity)

As a result, this experiment generated a total of 120.252 vertices and 120.966 edges in our Titan instance. Given the populated database, we performed 10,000 random automated queries, in order to evaluate query performance for our model. Both the population and query performance steps, were executed by two Python scripts, developed for this matter. These scripts were executed on an Intel Core i3 - 2100 CPU @ 3.10 GHz with two GB of Ram. The query performance benchmark took 200 seconds to finish, with an average of 20 milliseconds per query. It is also noteworthy that 75% of the queries run in less than this average time, as depicted in Figure 4.

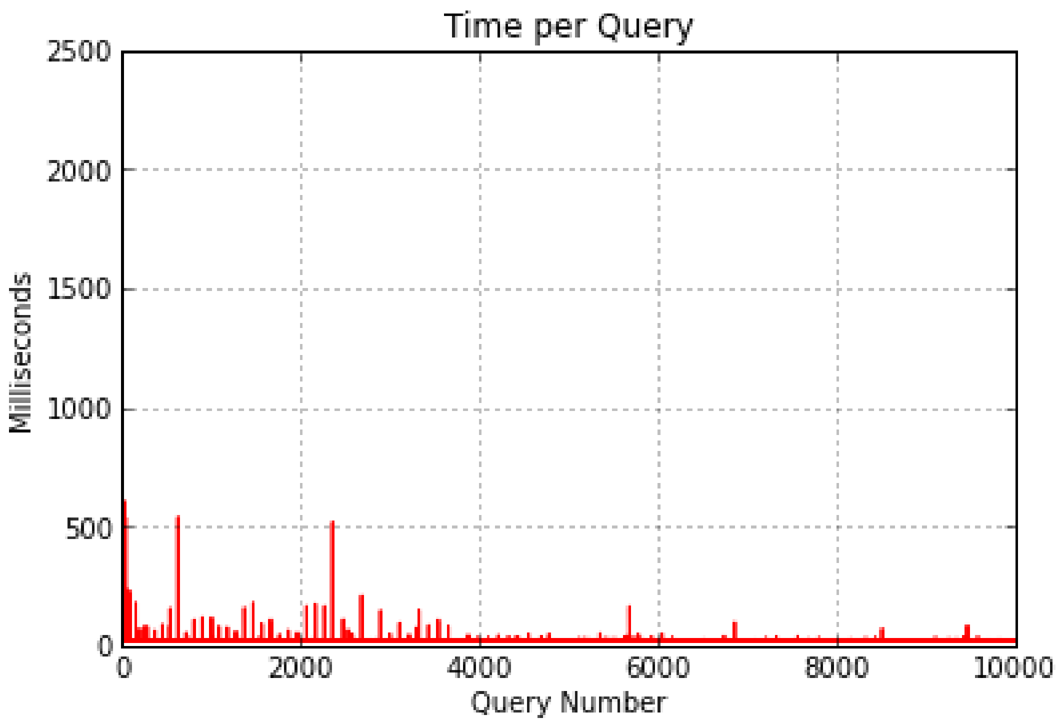

Figure 4: Histogram of query time per query number.

In the end, as previously said, the performance benchmark had only one instance of Titan as part of it. Obviously, in real case scenarios, that would hardly be the case. The major advantage of NoSQL technologies, in general, is exactly the capability of distributing the workload with all the servers running a database instance.

Summary & Conclusion

In this deliverable, the current Data, Policy and Knowledge Storage module architecture as well as the updated data model have been outlined. The architecture, based on NoSQL graph database technologies, was maintained as integration efforts among modules are demonstrating that the architecture is able to comply with the workload.

Furthermore, this deliverable details how application developers and other IMPReSS modules can interact with the Data, Policy and Knowledge Storage module, for both getting and/or providing data. In D5.1.1, a domain specific language (DSL) was provided on top of the available query language. That DSL had improvements also, in order to be more consistent, easier to learn and to achieve greater performance.

Finally, the domain specific language (DSL) section was expanded. Effort that included providing better documentation and more comprehensive examples, for developers to better grasp how to interface the Data, Policy and Knowledge Storage module.

References

[1] Jayavardhana G., Rajkumar B., Slaven M., Marimuthu P. Internet of Things (IoT): A vision, architectural elements, and future directions. Future Generation Comp. Syst. 29(7): 1645-1660 (2013).

[2] IMPReSS (Intelligent System Development Platform for Intelligent and Sustainable Society) project, Description of Work. EU-Brazil research and development Cooperation

[3] Silberschartz, A., Korth, H.F., and Sudarshan, S. Data models. ACM Computing Surveys, 28, 1, 105-108.

[4] Levene, M., Poulovassilis, A. The hypernode model and its associated query language, Proceedings of the fifth Jerusalem conference on Information technology, p.520-530, 1990.

[5] Titan – Distributed Graph Database. http://thinkaurelius.github.io/titan/

[6] Adam Jacobs, The Pathologies of Big Data, 1010data Inc., 2009. https://queue.acm.org/detail.cfm?id=1563874

[7] Faunus – Graph Analytics Engine. http://thinkaurelius.github.io/faunus/

[8] Gremlin – Graph Database Language. https://github.com/tinkerpop/gremlin

[9] Rexster Graph Server. https://github.com/thinkaurelius/titan/wiki/Rexster-Graph-Server

[10] Groovy. http://groovy.codehaus.org/

[11] Neo4J. http://www.neo4j.org/

[12] OrientDB. http://www.orientechnologies.com/orientdb/

[14] Tinkerpop. http://www.tinkerpop.com/

[15] Blueprints. https://github.com/tinkerpop/blueprints/wiki

[16] Hadoop. https://hadoop.apache.org/

[17] SPARQL. http://www.w3.org/TR/rdf-sparql-query/

[18] PyBulbs. http://bulbflow.com/overview/

[19] Hbase. https://hbase.apache.org/

[20] Oracle Berkeley DB. http://www.oracle.com/technetwork/database/database-technologies/berkeleydb/overview/index.html