SEVENTH FRAMEWORK PROGRAMME

Specific Targeted Research Project

Call Identifier:	FP7–ICT–2011–7
Project Number:	287305
Project Acronym:	OpenIoT
Project title:	Open source blueprint for large scale self-organising cloud environments for IoT applications

D5.1.2 Self-management and Optimization Framework

Document Id:	OpenIoT-D512-271213-Draft
File Name:	OpenIoT-D512-271213-Draft.pdf
Document reference:	Deliverable 5.1.2
Version:	Draft
Editor:	Jean-Paul Calbimonte
Organisation:	EPFL / AIT / DERI
Date:	2013 / 12 / 30
Document type:	Deliverable
Dissemination level:	PU (Public)

DOCUMENT HISTORY

Rev.	Author(s)	Organisation(s)	Date	Comments
V20	Robert Gwadera	EPFL	2013/10/18	Initial ToC
V21	Ivana Podnar Žarko	UNIZG-FER	2013/11/02	Added sections 3.5.2 and 3.7.1
V22	Chris Georgoulis	AIT	2013/11/22	Added sections 3.5.3.5, 3.8 and 4, Updated Issues with
V23	Mehdi Riahi	EPFL	2013/12/11	Added sections 3.4 and 4.1
	Jean-Paul Calbimonte	EPFL	2013/12/12	Adapted contents in Section 3.4 and Section 4, Updated Sec 1
	Jean-Paul Calbimonte	EPFL	2013/12/16	Rearranged 3.8 in 3.7.2, Updated 1.4, Modified 3.5.1
	Tian Guo	EPFL	2013/12/17	Added 3.3 and 4.5
	Mehdi Riahi	EPFL	2013/12/19	Modified 3.4 and 4.1, connected both sections.
	Jean-Paul Calbimonte	EPFL	2013/12/20	Edition Sections 3.4, 3.3, 4, 4.3, 4.4, Addressed early comments from project
V24	Ivana Podnar Žarko	UNIZG-FER	2013/12/23	Modified 3.5.2 Technical Review
V25	Jean-Paul Calbimonte	EPFL	2013/12/24	Addressed TR comments
V26	Chris Georgoulis	AIT	2013/12/27	Updated Section 3.1.5.3, 4.3, addressed TR Comments
V27	Martin Serrano	DERI	2013/12/30	Quality Review
V28	Martin Serrano	DERI	2013/12/30	Circulated for Approval
V29	Martin Serrano	DERI	2013/12/30	Approved
Draft	Martin Serrano	DERI	2013/12/30	EC Submitted

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

TERMS AND ACRONYMS

ACF	Autonomic Computing Forum
CQELS	Continuous Query Evaluation over Linked Stream
DCOM	Distributed Component Object Model
DMTF	Distributed Management Task Force
GSN	Global Sensor Network
ICO	Internet-Connected Objects
IoT	Internet of Things
LSM	Linked Stream Middleware
NGN	Next Generation Networks
PaaS	Platform as a service
RDF	Resource Description Framework
RMI	Remote Method Invocation
SaaS	Software as a service
SD&UM	Service Delivery and Utility Manager
SLA	Service Level Agreement
SLO	Service Logic
SPARQL	SPARQL Protocol and RDF Query Language
SQL	Structured Query Language
SSN	Semantic Sensor Networks
TMF	TeleManagement Forum
W3C	World Wide Web Consortium
WSN	Wireless Sensor Networks
UBO	Utility-based Optimizer

1 INTRODUCTION

1.1 Scope

One of the key characteristics of the OpenIoT cloud platform is its ability to make optimal usage of the resources that it comprises, with a goal to maximizing efficiency, sustainability and costs of both the sensing process and resource usage within the cloud. Furthermore, it is envisaged that these resource optimization functionalities are provided by the OpenIoT sensor-cloud infrastructure itself, in an autonomous fashion and without any human intervention. To this end, OpenIoT specifies and implements a framework for «self-management and optimization» associated with sensors, services and applications that are executed over the OpenIoT cloud. The framework optimizations have been designed in order to be executed at various levels, from cloud storage to bandwidth efficiency or query results caching.

This deliverable presents the specifications of the self-management and optimization framework of the OpenIoT platform, providing insights on its implementation in-line with the OpenIoT architecture. The deliverable introduces first the algorithms techniques and experimental evaluation that validate them; and then describes their implementation in the OpenIoT platform.

Towards the implementation of the OpenIoT self-management and optimization framework, this deliverable has a bi-directional interaction with other work packages dealing with the OpenIoT platform architecture (WP2) and implementation (WP4): On the one hand it provides inputs on the information that should be stored and managed within the OpenIoT system in order to enable the implementation of the algorithms, while on the other it takes into account the results of these work packages in order to properly design the practical implementation of the algorithms within the OpenIoT self-management and optimization framework.

1.2 Audience

The target audience of this deliverable includes:

- The consortium partners and more specifically consortium members dealing with the design and implementation of the OpenIoT open source platforms. These members take into account the results of this deliverable in order to design the OpenIoT platform elements (such as data structures) in a way that facilitates the implementation of the presented algorithms.
- Cloud computing and/or IoT researchers, which could be offered with a range of resource optimization schemes, that could be valuable in the scope of current and future implementations of systems attempting the IoT/cloud convergence.

1.3 Summary

This deliverable describes the OpenIoT self-management and optimization framework, in terms of algorithms and mechanisms that it comprises as well as in terms of their implementation over the OpenIoT platform and associated cloud infrastructure. As a first step the main operations and functionalities of the OpenIoT self-management and optimization infrastructure are described and related to the structure of management operations defined in state-of-the-art frameworks for autonomic computing and self-management. Along with a brief description of the optimization techniques that are employed in OpenIoT, an initial mapping of the various techniques on the OpenIoT architecture is performed.

Following the overview of the OpenIoT self-management and optimization infrastructure, the deliverable delves into more details about each one of the mechanisms. In particular:

- Efficient scheduling mechanisms are presented, aiming at optimizing the rates according to which the various sensors streams are streamed to the cloud and/or accessed by consumers.
- A variety of caching mechanisms are presented, aiming at accelerating access to frequently used/requested data.
- Cloud optimization for sensor data storage, using approximation of raw sensor data to view-models represented as functions.
- Utility-driven mechanisms are illustrated, aiming at maximizing the utility of the services, while minimizing the cost for setting them up and maintaining them.
- Efficient sensor data collection using the utility metrics, and also context-aware filtering from mobile devices.
- Optimization techniques employing the temporal and/or spatial aspects of the OpenIoT queries and services, along with semantic techniques for correlating queries and associated with reasoning operations over multiple data streams.

These techniques target two of the main goals of WP5, which are to **investigate techniques for energy-efficient service delivery**, and **resource sharing algorithms for accessing OpenIoT resources**.

Finally, we provide details and insight about the implementation of the abovementioned techniques for self-management and optimization within the OpenIoT platform. Therefore, and as it was stated in the goals of WP5, we **established an overall management and optimization framework for the OpenIoT infrastructure**. The framework incorporates the optimization algorithms listed above, researched in this work package. Specifically, we specify the following components, and indicate how they are being integrated into the OpenIoT platform:

- Utility-based optimization: related to Task **T5.2**, which addresses **optimization of resource sharing across service requests**, e.g. sensor data collection. This utility function in this module also incorporate privacy for the computation, addressing also **T5.3**, is related to the **definition of utility metrics in order to ensure the trustworthiness of the services**.
- Dynamic Sensor Control, which is also related to **T5.2**, but focusing on **resource and sharing of resources at the scheduling level**, with a view to making optimal use of the resources (sensors). The sensor control techniques are also related to **T5.1** as they deal with **energy efficiency at the (virtual) sensor level**.
- Caching Simulation: The cache simulator of query responses is directly related to **T5.1** as it deals with **bandwidth optimization**, reducing the data transmission volumes between the data layer and the rest of the OpenIoT architecture.
- Cloud Optimization is related to both **T5.2** and **T5.1** as it deals with **resource management for cloud environments** (e.g. for efficient cloud storage and processing), and also on **bandwidth optimization** (reduced data volumes over the wire).

1.4 Structure

The remainder of the deliverable is structured as follows:

Section 2 concentrates on the OpenIoT management and optimization functionalities, related to the integration of large-scale sensor data into the cloud. This requires establishing a common understanding of the challenges and features for enhancing complex systems functionality to support a large number of sensors, devices and services, and their dynamic deployment and implementation within the OpenIoT platform. In particular, Section 2.2 describes the OpenIoT self-management features assessed in terms of performance, reliability, scalability, resource optimization and cost efficiency. Section 2.3 introduces the OpenIoT vision to define an ICO service lifecycle control. Self-management operations represent the building blocks of the core IoT service lifecycle in OpenIoT, and provide significant contributions to the OpenIoT platform in general. As per definition, service creation, service customization, service management, service operation, service billing and customer support, complete the ICO service lifecycle. In this chapter, these operations are explained and related with OpenIoT functionalities and technologies supporting ICO service lifecycle in cloud environments.

Section 3 is devoted to presenting a detailed analysis of OpenIoT approaches that enable self-management and optimization. It presents the techniques OpenIoT is proposing to use, as part of the self-management and optimization functionalities introduced in Chapter 2. Section 3.1 introduces a summary where self-management and optimization functionalities are related with OpenIoT techniques. Section 3.2 provides details about the scheduling functionality in OpenIoT and describes the

related work in sensor networks and multi-query optimization. Later in this section we discuss different multi-query data management solutions and caching techniques in OpenIoT as well as pull approaches and caching of sensor ICO data (random or based on frequency request). In Section 3.3 cloud optimization is discussed for efficient sensor data storage of segment approximations of sensor data instead of raw measurements. This novel querying mechanism combines in-memory and key-value stored data management in the cloud. Section 3.4 discusses utility-based optimization specifications associated with the OpenIoT platform. We adapt a utility-driven approach to the system optimization, which tries to maximize the net benefit measured as a difference between the benefit of the provided information and the cost of maintaining the system in terms of energy consumption/bandwidth and the cost of ensuring privacy. Section 3.5 specifies efficient sensor data collection techniques, which OpenIoT is proposing to be part of its final architecture, including algorithms for utility-based sensor data acquisition and filtering on mobile devices. Section 3.6 introduces efficient query processing techniques implemented by the LSM module of the OpenIoT architecture. Section 3.7 introduces energy efficiency and bandwidth optimization and provides an analysis for specifying this functionality in OpenIoT final architecture. We include also an analysis on energy and bandwidth consumption on mobile devices.

Section 4 provides details about the implementation of the techniques in Section 3 for the OpenIoT platform architecture. Specifically, we provide the specification and details of the utility-based optimization (Section 4.1) to be used by the OpenIoT integrated prototype, based on the techniques described in Section 3.4. In Section 4.2 we provide details on the implementation of the Dynamic Sensor Control Module, based on indirect sensor control, as introduced in Section 3.7.2. The Caching Scenarios prototype is detailed in Section 4.3. Finally, the cloud optimization approach implementation is specified in Section 4.4, based on the algorithms and techniques described previously in Section 3.3.

Section 5 concludes the deliverable. We also include references where more detailed specifications and descriptions to the proposed technologies can be found.

2 OPENIOT MANAGEMENT AND OPTIMIZATION FUNCTIONALITIES

2.1 Overview

The process of integrating sensor data and cloud infrastructures as part of a blueprint open source solution in OpenIoT, creates new challenges in terms of enhancing complex system functionality, enables large support of sensors, devices and service systems, and enables dynamic deployment and implementation of various innovative IoT services.

IBM[1] has introduced automaticity as part of the vision of autonomic computing *"systems manage themselves according to an administrator's goals. New components integrate as effortlessly as a new cell establishes itself in the human body. These ideas are not science fiction, but elements of the grand challenge to create self-managing computing systems"*. This principle has emerged and transcended beyond computing frontiers and also in the area of the communications management, the term autonomic communications has been researched for several years, reflecting a real challenge to materialize the vision of transparent interaction between administrator's goals and systems self-management operations. In the late 90's supported by the Autonomic Computing Forum (ACF) autonomics brought the concept of seamless mobility associated to scenarios for people configuring new personalized services using displays, smart posters and other end-user interaction facilities, as well as their own personal devices. Named lately as pervasive computing, autonomics bring the inherent necessity to increase the functionality of those systems dealing with additional information and funded on communication system infrastructures. Pervasive service requirements are headed by the interoperability of data, voice, and multimedia using the same (converged) network.

This requirement defines a new challenge: the necessity to integrate smartness to the systems and make the infrastructure more reactive by means of data and services control. Nowadays the Future Internet design with the inclusion of ICOs is motivated by both, the necessity to support the requirements of pervasive services and the necessity to satisfy the challenges of self-operations dictated by the largely named IoT paradigm.

Autonomic systems must dynamically adapt the services and resources that they provide to meet the changing needs of users and/or to respond to changing environmental conditions alike that of system control; this requires the integration of management information into the OpenIoT platform. Figure 1 depicts the OpenIoT autonomic control loop proposed in OpenIoT. This model for OpenIoT is crucial, as each day more complex ICO consumers require novel services, which in turn require

[1] IBM The Vision of Autonomic Computing, IBM Research, Vision and Manifesto.

more complex support systems that must harmonize multiple technologies and linked information from sub-systems interacting with the offered embedded services.

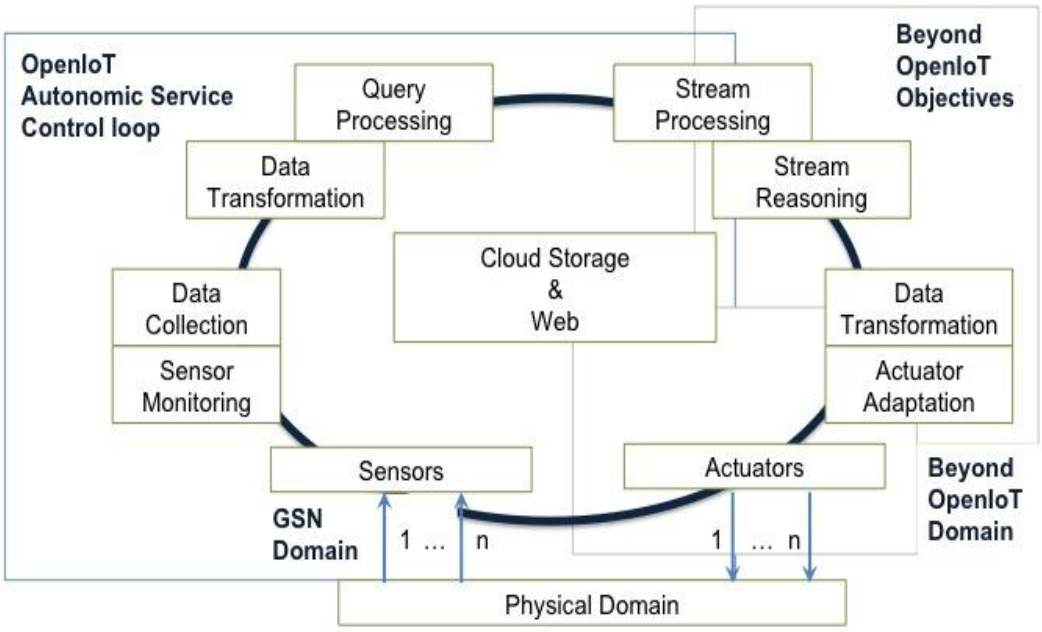


Figure 1. OpenIoT Autonomic Service Control Loop for ICO's.

ICOs promise new smart scenarios, and at the same time create challenging environments for deploying user-centric applications and services. ICO systems require information and systems able to support services and especially interoperable applications. In autonomic systems linked data plays the important role of enabling the management plane to adapt the services and resources that it is offering to the changing demands of the user, as well as adapt to changing environmental conditions, by meaning of the linked nature, thus enabling the management of new functionalities in ICO complex systems [Serrano 2008].

2.2 OpenIoT Self-management Features

2.2.1 Assessing Self-Management Functionalities within OpenIoT Architecture

The vision of self-management creates an environment that hides the underlying complexity of the management operations, and instead provides a façade that is appealing to both administrators and end-users alike. It is based on consensual agreements between different systems (e.g., management systems and information support systems), and it requires a certain degree of cooperation between the systems to enable interoperable data exchange.

One of the most important benefits of this agreement is the resulting improvement of the management tasks and operations using such information to control ICO's and their applications. However, the descriptions and rules that coordinate the control operations of an ICO system are not the same as those that govern the sensor data in each application system. For example, information present in a particular sensor network with primarily proprietary technology is often restricted to control the operation of a service, and usually has nothing to do with service management.

In the scope of OpenIoT, cooperation and interactions between various components of the OpenIoT architecture are required in order to support the self-management functionalities. Figure 2 depicts the OpenIoT Autonomic Self-management Framework for IoT Systems (ICO's), and presents the OpenIoT architecture components that implement and support the various functionalities.

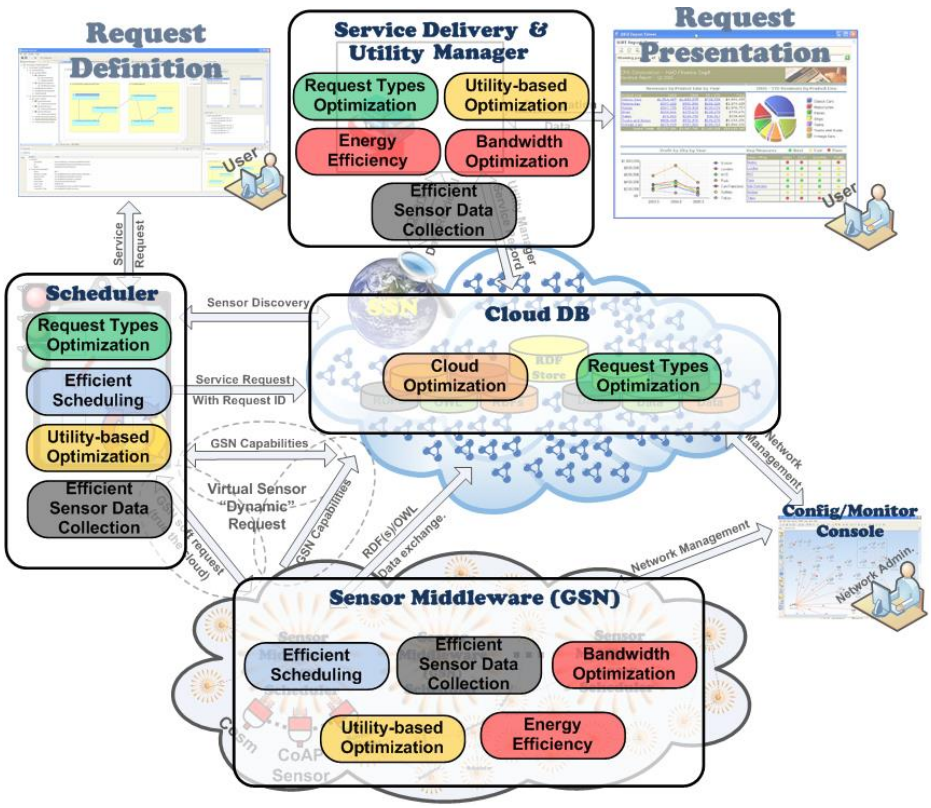


Figure 2. OpenIoT Architecture with Self-management Features.

Figure 2 serves as a reference for the positioning of these functionalities in the scope of the OpenIoT architecture. It summarizes the main functionalities provided by the OpenIoT architecture, namely efficient scheduling, cloud optimization, utility-based optimization, request types optimization, efficient sensor data collection, and energy/bandwidth optimization. The following paragraphs present in detail the various functionalities depicted in the figure, and how they are assessed in terms of performance, reliability, scalability and resource optimization.

2.2.2 Performance

Scalability and interoperability between heterogeneous, complex and distributed ICO systems is always a challenge and it requires new management and optimization functionalities. However, this added complexity might hurt the overall performance of the platform. Performance may be understood in different ways, depending on the perspective to be taken. From the point of view of an end user, performance can be perceived, for instance, in terms of query response times. In the case of a sensor it can be related to data transmission rates, or for a processing application it can be related to the throughput of data analysis of ICO data. In this regard, most of the optimization schemes proposed for the OpenIoT should have a positive impact on performance. Efficient scheduling and utility-based optimization result in reduced amounts of sensor interactions with the query middleware, thus reducing query times. Cloud optimization is precisely devised to also reduce the time spent in processing queries, using highly efficient cloud storage mechanisms. Efficient sensor data collection and bandwidth optimization improve performance in terms of throughput.

As an inherent functional limitation, ICO management systems do not support a large spectrum of devices, such as wearable computers and specialized sensors. Furthermore, ICO systems are every day being provided with embedded technology/connectivity, which is used to make new types of networks that provide their own services (e.g., simple services supporting other, more complex, services), which implies that management task become more difficult and complex in terms of scalability.

In OpenIoT, we deal with linked data or information sharing. In project scenarios use a broad mixture of technologies and devices (sensors) that generate an extensive amount of different types of information, many of which need to be shared and reused among the different service management components with different data representation mechanisms. This requires the use of different data models, due both to the nature of the information being managed as well as the physical and logical requirements of applications. However, information/data models (linked data and particularly RDF) do not have everything necessary to build up this single common interoperable sharing support system. In particular, there is a need to delegate the ability to describe behaviour of the services and application with the infrastructure.

2.2.3 Reliability

Traditionally management systems approaches define a strict layering of functionality and cross-layered interactions are left aside. In OpenIoT we explore the broad diversity of resources, devices, services, and systems, which are interconnected and exchange information across layers.

This complex structure also plays a role in terms of the overall reliability of an IoT platform. Each device that contributes to the OpenIoT ecosystem is subject to varying

degrees of dependability. Some devices may provide high quality data (e.g. well calibrated sensors) while others may not even be available at some time of the day (e.g. community sensing where citizens voluntarily provide data through home-made devices). Moreover, reliability of mobile sensing devices can, in some cases, be undermined by external factors such as interference, signal problems or unreachability. In OpenIoT, we address some of these issues with the utility-based optimization, where the reliability of ICOs can influence the inclusion or exclusion of its data from the acquisition process.

In OpenIoT we pursue the objective of annotating information, described in services and data models, so to provide an extensible, reusable common management platform that provides new functionality to better manage resources, devices, networks, systems and services [Serrano 2008]. Given the fact that different data representations are a necessity in the next generation Internet solutions [Clark 2003], the typical solutions have attempted to define a single common information model that can harmonize the information present in each of these different management data models. Using a single information model prevents different data models from defining the same concept in conflicting ways. In addition, the use of a single common information model enables the reuse and exchange of service management information. Examples of using a single common information model include the initiative CIM/WBEM (Common Information Model/Web Based Enterprise Management), [DMTF-CIM] from the DMTF (Distributed Management Task Force, Inc.) and broadly supported by the Shared Information Model [TMF-SID] of the TMF [TeleManagement Forum]. However neither of them has been completely successful, as evidenced by the lack of support for either of these approaches in network devices currently manufactured. This indicates that SID model lacks the extensibility to promote the interoperability and enhance its acceptance and expand its standardization.

In OpenIoT we are proposing an alternative to facilitate the interoperability, by semantically enriching the information models to contain the references in the form of relationships between sensor data required to provide the service. By using one or more ontologies and the referenced sensor data ontology [W3C SSN][2] then service systems and applications using information contained in the service model can access and do operations and functions for which they were designed. This functionality in particular impacts the performance of the ICO systems by its unique and novel feature of enabling management operations using the information contained in the information models (sensor data) for ICO service provisioning.

2.2.4 Scalability

The vision of ICOs which enable societies to use a wide range of sensors, devices and computing systems to “transparently” create smart applications and on-demand

[2] http://www.w3.org/2005/Incubator/ssn/

services automatically, requires beyond sub-systems offering reliable control and connectivity, associated management systems that are able to support such exponentially growing and dynamic service creation. In this vision, not only the numbers of potential users may rapidly grow (as in traditional web platforms), but also the number of ICOs and participating sensors may grow. The scale of queries, data streams and sensor metadata that needs to be processed requires carefully designed algorithms that go beyond centralized traditional data analysis techniques. Moreover, the inherently distributed nature of sensors, require a wide scale network of decentralised processing, to which the OpenIoT users and application should be able to issue continuous queries. This adds even another level of scale, as the data is highly dynamic, hence not efficiently tractable with standard query processing techniques. Data streams and dynamic queries at potentially very fast rates require highly scalable processing, which is addressed in OpenIoT through Request type optimization and Cloud Optimization.

2.2.5 Resource Optimization and Cost Efficiency

Self-management features depend on both the requirements as well as the capabilities of the middleware frameworks or platforms for managing information describing the services as well as information supporting the delivery and maintenance of the services. The representation of information impacts the design of novel syntax and semantic tools for achieving the interoperability necessary when ICO resources and services are being managed. Middleware capabilities influence the performance of the information systems, their impact on the design of new services, and the adaptation of existing applications to represent and disseminate the information.

In OpenIoT, the use of rule-based engines for controlling ICO's service management is augmented with the use of standard ontologies. This enables the management systems to support the same management data to accommodate the needs of different management applications through the use of rich semantics [Serrano 2012]. Service management applications for IoT systems highlight the importance of formal information.

The rules are used for managing various aspects of the service lifecycle. It is important to identify in OpenIoT what is meant by the term "service lifecycle". Currently, the TMF is specifying many of the management operations in networks for supporting services [TMN-M3050][TMN-M3060], in a manner similar to how the W3C specifies web services [W3C-WebServices]. However, a growing trend is to manage the convergence between infrastructure and services (i.e., the ability to manage different service requirements for data, voice, and multimedia serviced by the same network), as well as the resulting converged services themselves. The management of NGN pervasive services involves self-management capabilities for improving performance and achieving the interoperability necessary to support current and next generation services.

2.3 Self-management Framework – ICO Services Lifecycle

This section describes an organizational view that enables the ICO service lifecycle to be explicitly modelled and semantically managed. This in turn ensures information interoperability necessary to manage different services in IoT applications. This section describes the organizational view for the Autonomic Self-management Framework, which can be divided into six distinct phases with specific tasks [Serrano 2008].

Management operations enabling the autonomic nature of ICO systems are the core part of the IoT service lifecycle, and where the contributions in OpenIoT are focused. The management phase for IoT services is highlighted in Figure 3. Creation and customization of services, accounting, billing and customer support are outside the scope of OpenIoT. However, they are considered for a design description of ICO systems. The different service phases exposed in this section describe the service lifecycle foundations. The objective is to focus the research efforts on understanding the underlying complexity of service management, as well as a better understanding about the roles for the components that make up the service lifecycle, using interoperable information that is independent of any specific type of infrastructure that is used in the deployment of IoT services.

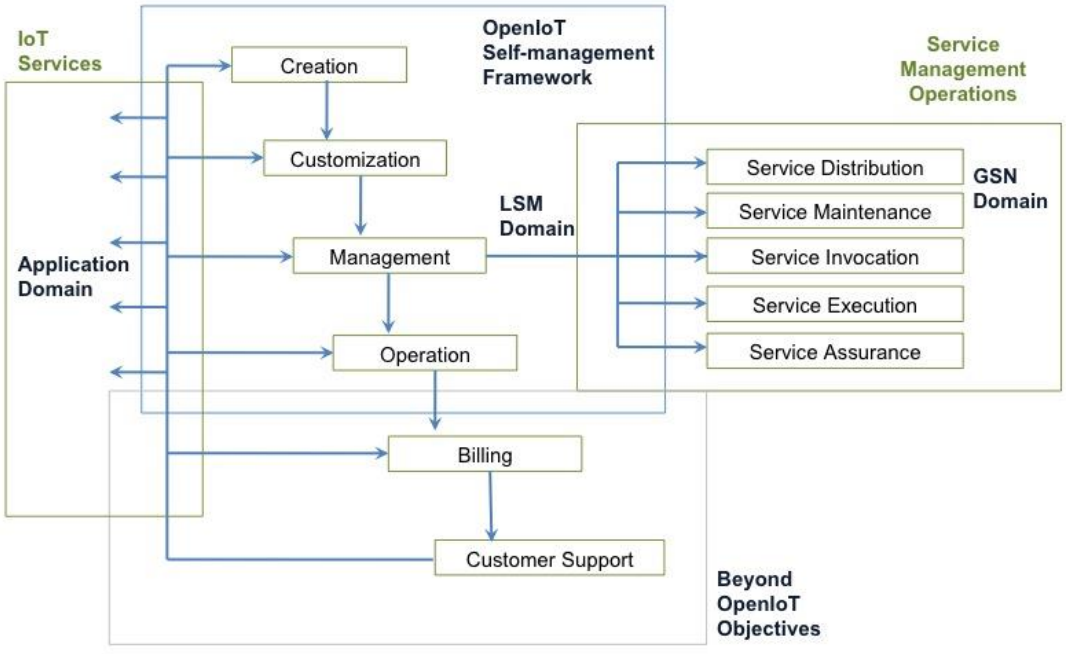


Figure 3. OpenIoT Autonomic Self-management Framework for IoT Services (ICO's).

2.3.1 Service Creation

The creation of each new IoT service starts with a set of requirements; the service at that time exists only as an idea. This idea of the service originates from the

requirements produced by market analysis and other business information. At this time, technology-specific resources are not considered in the creation of a service. However, the infrastructure for provisioning this service must be abstracted in order to implement the business-facing aspects of the service as specified in a service definition process [Serrano 2008].

The idea of IoT service must be translated into a technical description of a new service, encompassing all the necessary functionality for fulfilling the requirements of that service (e.g., physical devices interconnection, sensor data collection, virtual sensor aggregation, etc.). A service is conceptualized as the instructions or set of instructions to provide the necessary mechanism to provide the service itself and called service logic (SLO).

2.3.1.1 Efficient Scheduling

The OpenIoT system comprises the notion of scheduling of requests, which undertakes the task of technically describing a new service. The OpenIoT global scheduler component, which OpenIoT architecture specifies, receives all the User requests for IoT services and fulfils the requirements of that service. A wide range of different optimization algorithms can be implemented at the scheduler component of the OpenIoT architecture. So the main OpenIoT efficient multi-level (global, local) scheduling optimization scheme involves multi-query data management and caching techniques that include:

- pull approach at local scheduling,
- caching of sensor/ICO data,
- caching of sensor/ICO data based on frequency of requests, and
- caching of (SPARQL) queries

2.3.2 Service Customization

Service customization, which is also called authoring, is necessary for enabling the IoT service provider to offer for its consumers the ability to customize aspects of their IoT services (i.e., ICO selection and/or configuration) according to their personal needs and/or desires (e.g. defined by a query language). Today, this is a growing trend in web-services and business orientation. An inherent portion of the customization phase is an extensible infrastructure, which must be able to handle service subscription and customization requests from administrators as well as ICO consumers.

2.3.2.1 Efficient Sensor Data Collection

In OpenIoT we focus on stream data processing components enabling the deployment over multiple infrastructures. By combining query languages (i.e. SPARQL) and stream data processing components to enable the User to customize

a service on its own needs. And based on an efficient stream processing, at the collection and distribution, making the sensor data system more efficient towards previous identification of “intelligent” data providers’ by sensing and actuating over streaming data infrastructures, rather than having to deploy data processing infrastructures by themselves.

2.3.2.2 Request Types Optimization

Another type of service customization in OpenIoT exists in the request types optimization where we make use of LSM (Linked Sensor Middleware) [Le-Phuoc 2011]. We use LSM as an extended middleware with functionalities to transparently cater for dynamic stream information. LSM uses efficient query algorithms that may provide a global view of the whole dataset to the data processing operators.

2.3.3 Service Management

In this section, the management operations of an ICO service and its interactions are identified as distinct management operations from the rest of the service lifecycle phases. Figure 4 depicts management operations as part of the management phase in a pervasive service lifecycle.

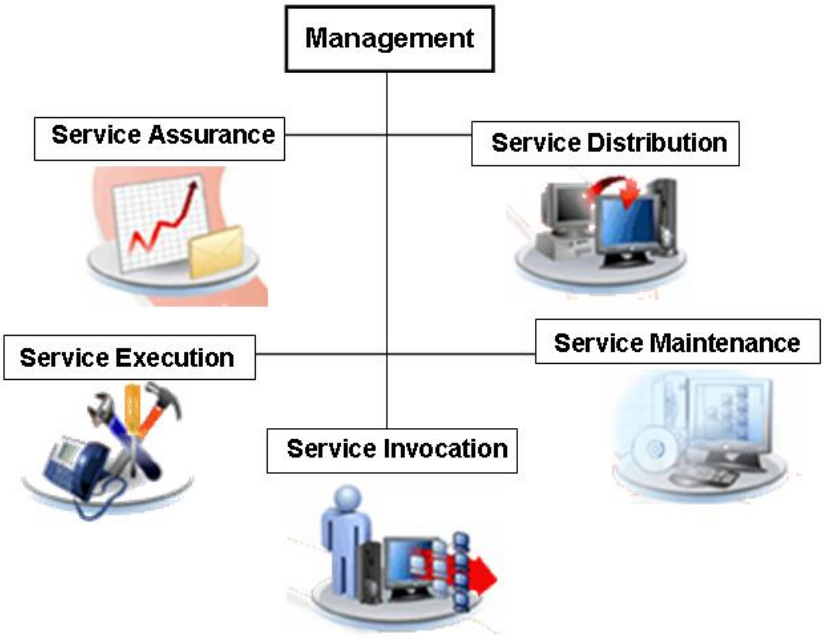


Figure 4. Service Management & Operations.

The main service management tasks are service distribution, service maintenance, service invocation, service execution and service assurance. An important functional aspect of the OpenIoT service management framework implementation is the dynamic (on the fly) deployment of IoT services using specific logic rules. For instance, when an IoT service is going to be deployed, decisions have to be taken in

order to determine which sensor or devices (things) are going to be used to support the service. This activity is most effectively done through the use of particular logic rules that map the user with the desired data sources and with the capabilities of the set of ICO that are going to support the service. Moreover, service invocation and execution can also be controlled by same logic rules, which enable a flexible approach for customizing one or more service templates to multiple users.

On the other hand, management is an effective mechanism for maintaining code to realize the IoT services, changes and assurance of the IoT service included. For example, when variations in the delivery of the service are sensed by the system, one or more policies can define the set of actions that need to be taken to solve the problem. In this way, the use of policies enables different behaviour to be orchestrated as a first step to implement self-management functionality.

2.3.3.1 Service Distribution

This step takes place immediately after the service creation and customization in the service lifecycle. It consists of storing the service code in specific storage points. Policies controlling this phase are termed code distribution policies (*Distribution*). The mechanism controlling the code distribution determines the specific set of storage points that the code should be stored in. The enforcement is carried out by the components that are typically called Code Distribution Action Consumers.

2.3.3.2 Service Maintenance

Once the code is distributed, it must be maintained in order to support updates and new versions. For this task, we use special policies, termed code maintenance Policies (*CMaintenance*). These policies control the maintenance activities carried out by the system on the code of specific services. A typical trigger for these policies could be the creation of a new code version or the usage of a service by the consumer. The actions include code removal, update and redistribution. These policies are enforced by the component that is typically named the Code Distribution Action Consumer.

2.3.3.3 Service Invocation

The service invocation is controlled by special policies that are called *SInvocation* Policies. The service invocation tasks are realized by components named Condition Evaluators, which detect specific triggers produced by the service consumers. These triggers also contain the necessary information that policies require in order to determine the associated actions. These actions consist of addressing a specific code repository and sending the code to specific execution environments in the network. The policy enforcement takes place in the Code Execution Controller Action Consumer.

2.3.3.4 Service Execution

Code execution policies, named *CExecution* policies govern how the service code is executed. This means that the decision about where to execute the service code is based on one or more factors (e.g., using performance data monitored from different network nodes, or based on one or more context parameters, such as location or user identity). The typical components with the capability to execute these activities are commonly named Service Assurance Action Consumers, which evaluate network conditions. Enforcement of these policies is the responsibility of the components that are typically called Code Execution Controller Action Consumers.

2.3.3.5 Service Assurance

This phase is under the control of special policies termed service assurance policies, termed *SAssurance*, which are intended to specify the system behaviour under service quality violations. The Service Assurance Condition Evaluator evaluates rule conditions. These policies include preventive or proactive actions, which are enforced by the component typically called the Service Assurance Action Consumer. Information consistency and completeness is guaranteed by a policy-driven system, which is assumed to reside in the service creation and customization framework.

2.3.3.6 Utility-based Optimization

In OpenIoT for the dynamic deployment of IoT services we adapt a utilitarian approach optimization for the system's logic rules. The utilitarian approach tries to maximize the net benefit measured as difference between the benefit of the provided information and the cost of maintaining the system in terms of energy consumption/bandwidth and the cost of ensuring privacy.

2.3.4 Service Operation

The operation of a deployed IoT service is based on monitoring aspects of the cloud infrastructure that support that service, and variables that can modify the features and/or perceived status of the communications. Usually, monitoring tasks are done using agents, as they are extensible and can only accommodate a wide variety of information, and are easy to deploy. The information is processed by the agent and/or by middleware that can translate raw data into and from having explicit semantics that suit the needs of different applications.

2.3.4.1 Cloud Optimization

In OpenIoT we enforce adaptive cloud optimization algorithms based on the needs of each deployed scenario. The cloud infrastructure can be managed based on its functional schemes (i.e. access/storage charges). In particular we target model-based sensor data approximation to reduce the amount of data for query processing,

using a MapReduce evaluation paradigm. In this way the OpenIoT platform adapts the service runtime having in mind its cost-effectiveness and data integrity.

2.3.5 Service Billing

Service billing is just as important as service management, since without the ability to bill for delivered IoT services provided, the organization providing those services cannot make money. Service billing is often based on using one or more accounting mechanisms that charge the customer based on the resources used in the network. In OpenIoT, we particularly align our approach with the cloud paradigm enabling pay-as-you-go services. In the billing phase, the information required varies during the business lifecycle, and may require additional resources to support the billing. Service metering is implemented in the utility manager, which keeps track of the utility metrics specified in D4.2.1. This metering can then serve as a foundation for service billing.

2.3.6 Customer Support

Customer support provides assistance with purchased IoT services, while IoT main feature is the non-dependence or dependency of service provider, computational[3] resources or software[4] , or other support goods are required for the provisioning of complex IoT services. Therefore, a range of services[5] and resources (mainly cloud) related are required to facilitate the maintenance and operation of the IoT services, and additional context (and sometimes the uncovering of implicit semantics) is necessary in order for user or operators to understand problems with purchased services and resources. OpenIoT foresees to enable the User with the ability to configure, monitor and maintain IoT operative services (as described in deliverable D2.3, Chapter 9.2). This is done through specialized monitoring and configuration interfaces, which are able, for example, to modify object-objet connections or activate/de-active sensors, instead of relaying this capacity to the implemented service maintenance functionality in the subsystem. This is mainly an OpenIoT platform system administrator/ service provider tool, which would enable him to deploy, configure, manage and offer service customer support more dynamically when necessary.

3 http://en.wikipedia.org/wiki/Computer

4 http://en.wikipedia.org/wiki/Software

5 http://en.wikipedia.org/wiki/Customer_service

3 DETAILED ANALYSIS OF OPENIOT APPROACHES TO SELF-MANAGEMENT AND OPTIMIZATION

3.1 Overview and Summary of Contributions

The previous section has introduced the main techniques that are employed in the scope of the OpenIoT self-management framework. The following paragraphs provide more details on each of the presented schemes (i.e. on Efficient Scheduling including Caching, Cloud Optimization, Utility-based Optimization, Efficient Sensor Data Collection, Request Types Optimization, Energy Efficiency and Bandwidth Optimization). Furthermore, Table 1 illustrates how different schemes contribute to the various non-functional goals of the OpenIoT system. In particular:

- Efficient scheduling and caching mechanisms reduce the time and resources needed to deliver an OpenIoT service (e.g., to dynamically formulate a SPARQL query) and/or its results (e.g. accelerates access to frequently used sensor data). In this way it contributes to the performance and resource optimization goals.
- Cloud optimization techniques reduce the overall storage costs, while also boosting the ability of OpenIoT to interface and use multiple elastic cloud computing infrastructures. In this way, it contributes to performance, resource optimization, as well as the scalability of the overall OpenIoT infrastructure.
- Utility-based optimization maximizes the net benefit stemming from the use of the cloud, while accounting for the cost for setting up and maintaining the cloud infrastructure and services. It therefore addresses optimization of aspects such as performance, reliability and resource optimization (depending also on the utility metrics employed).
- Efficient sensor data collection exploits spatial correlation of the data and/or the queries in order to boost performance and scalability.
- Finally, both request types optimization and efficient bandwidth allocation enable faster access to data of specific services requests thereby boosting performance and resource optimization.

These optimization characteristics and properties are more explicitly presented and justified in the following subsections, which elaborate on the various optimization schemes.

Table 1. Self-Management and Optimization Functionalities vs. OpenIoT Techniques.

Self-Management and Optimization Functionalities vs. OpenIoT Techniques	Performance	Reliability	Scalability	Resource/ Cost Optimization
Efficient Scheduling	X		X	X
Cloud Optimization	X		X	X
Utility-based Optimization	X	X		X
Efficient Sensor Data Collection	X		X	
Request Types Optimization	X			X
Energy Efficiency and Bandwidth Optimization	X			X

3.2 Efficient Scheduling

3.2.1 Context of Scheduling Functionality in OpenIoT

The OpenIoT system comprises the notion of scheduling of requests for OpenIoT service formulation. Indeed, the OpenIoT architecture specifies a global scheduler component, which receives all requests for IoT services, which are submitted, to the OpenIoT system. At the level of this component, the platform has the ability to access information about the data requested by each service as well as about the sensors and ICOs that are used in order to deliver the requested data. As a result, a wide range of different optimization algorithms can be implemented within the scheduler component of the OpenIoT architecture. Overall, OpenIoT makes provisions for scheduling at multiple levels (global, local), which enable a wide range of optimization schemes.

3.2.2 Related Work in Sensor Networks and Multi-Query Optimization

3.2.2.1 Pre-Processing, Data Aggregation and In-Network Processing in WSN

In terms of specific optimizations OpenIoT is inspired by a number of optimization algorithms that exist in the Wireless Sensor Networks (WSN) literature, where data management is commonly applied as a means to optimize the energy efficiency of the network [Abadi 2005]. In WSN a set of in-network processing algorithms are applied in order to optimize the use of the network on the basis of aggregate operations [Yao 2002]. Query aggregation follows typically a model that includes: (a) The establishment of a query in the sensor network, (b) The assembly of partial results from multiple nodes in the network, on the basis of proper query processing and (c) The accommodation of multiple applications requests which send a number of queries to the network. This is accomplished based on the query processing mechanisms outlined above [Meng 2008]. Different research works have focused on

a variety of in-network processing and data management techniques in order to optimize processing times and/or reduce the required access to the sensor network [Madden 2005], [Trigoni 2005]. For example [Lee 2006] proposes an in-network materialized view that could be shared by multiple queries to reduce the number of messages to the WSN. Another characteristic of the systems is that the aggregated sensed results concern specific common spatial regions (i.e., provide aggregate data from sensors residing in a specific geographical regions of high interest).

In general, the in-network processing approaches outlined above can be classified into three broad categories, namely:

- **Push Approaches**, which proactively disseminate sensor readings to upstream entities (nodes), since they anticipate that queries for their data/readings are asked. Push approaches are useful when multiple queries are executed in the network and their locations are not known in advance. Examples can be found in [Ye 2002] and [Heinzelman 1999].
- **Pull Approaches,** which keep sensor silent until a request for their data arrives. Upon this arrival, relevant sensors are traversed and their readings are collected and aggregated in an access point (sink). Optimizations focus therefore in the most appropriate ways to collect the readings. Example systems can be found in [Yao 2002], [Intanagonwiwat 2000] and [Maddan 2002].
- **Hybrid approaches,** which comprise a two-step process aiming at leveraging the advantages of both push and pull approaches [Li 2004]. The first step involves pushing of sensor readings to collection points on the basis of a given algorithm. Accordingly, the second step involves pulling readings from sinks on the basis of application requirements. Hybrid approaches provide the means for spatial efficiency in query retrieval (see for example [Lee 2006] and [Ratnasamy 2002]).

Note that optimizations in WSN have to deal with resource constraints, such as memory limitations in the sensor nodes. Furthermore, they deal with the problem of energy efficiency in sensor networks. In OpenIoT those problems are not the primary ones to be solved, especially for the part of the platform that deal with virtual sensors and which resides in the cloud. Moreover, the pull and push approaches outlined above, give rise to ideas and techniques for optimizing the efficiency of OpenIoT in serving multiple IoT services.

3.2.2.2 Caching in WSN

Caching is another technique that can reduce network traffic, while also enhancing the availability of data to the users (sink). The caching concept involves maintaining sensor (data streams) data to a cache memory in order to facilitate fast and easy access to them. Likewise caching mechanisms could also maintain the sensor queries themselves along with their data, which in the case of OpenIoT could obviate the need to execute the results of previously executed SPARQL queries. In the area

of OpenIoT this has one extra benefit, which is associated with the cost of accessing the cloud infrastructure.

A large number of caching algorithms for WSN have been proposed in literature and an exhaustive presentation is out of the scope of this deliverable. However, among the prominent examples are caching schemes based on the formulation of network trees per sink and the subsequent identification of a common sub tree whose root is used as the caching backbone [Li 2009]. There are also techniques that consider the mobility patterns of the nodes in order to form groups [Chow 2007]. Moreover, [Chand 2006] has introduced the formulation of non-overlapping clusters (for caching) based on geographical proximity. In this case sensor networks (e.g., MANETs) are partitioned in equal size cluster, where each client looks for the data in the case of a miss in the local cache of the node.

3.2.2.3 Optimizing Queries to Distributed Data Streams

Relevant to OpenIoT are also data streams system, which handle data from multiple geographically distributed sources. Typical examples of such systems are e-science systems leveraging multiple distributed sensor-driven measurements. In such systems, the in-network processing techniques ([Madden 2005], [Yao 2002], [Ahmad 2004]), and source filtering [Olston 2003] facilitates load distribution and overall boosts performance. In such systems it is also common to execute continuous queries, i.e., recurrent queries running periodically and asking for the same data. A popular optimization approach used in this case involves the construction of a query plan (e.g., a plan involving specific join ordering) before the execution of the queries as a pre-planning step. At run time, this plan is deployed in order to improve performance [Pietzuch 2006].

These systems give rise to ideas about anticipating and pre-planning the number of queries that are submitted to the OpenIoT system. As part of pre-planning a number of (frequently used) multi-sensor queries could be cached in the scope of the OpenIoT platform, thereby enabling the system to provide a fast response when these queries are asked again.

3.2.3 Multi-Query Data Management and Caching Techniques in OpenIoT

On the basis of the schemes outlined above, OpenIoT employs a number of relevant strategies as part of its management and optimization framework. These strategies take into account the differences of the OpenIoT sensor-cloud from conventional wireless sensors networks, in terms of both structures and costs.

3.2.3.1 Implementation of a Pull Approach at Local Scheduling

OpenIoT can be thought as a highly distributed network of multi-sensor nodes, notably nodes of the X-GSN sensor middleware that stream data to the OpenIoT

cloud. A pull approach is adopted in terms of accessing the sensor networks of the various nodes. As part of this approach:

- Each X-GSN node can maintain a list of services that need data from each of its sensors/ICOs. For each sensor/ICOs and each service using the sensor, the minimum frequency of data retrieval is also recorded (i.e. denoting the time window that of the streaming for the j^{th} sensor of the i^{th} service).
- In cases when a X-GSN node is not streaming any data from a particular sensor to the cloud, the system is saving in terms of bandwidth and costs associated with cloud access. This rule can be changed only in the case when a cloud provider specifies it otherwise. This can be the case when historical data are required from a given sensors e.g., as part of the SLA (Service Level Agreement) with an end-user.
- In all other cases the X-GSN node can stream data on the basis of the minimum frequency among those specified for the sensors that participate in the active services associated with the X-GSN node.

The above-mentioned pull approach can minimize the accesses to the cloud infrastructure, which could be costly in terms of both latency and monetary cost. It is be however possible for the OpenIoT platform to explicitly activate and deactivate sensors in X-GSN. The activation or de-activation of X-GSN virtual sensors, in order to optimize resources is performed by the OpenIoT Scheduler module, as described in Section 3.7.2, through indirect sensor control.

3.2.3.2 Caching of sensor/ICO data

As already outlined, the access to the cloud infrastructure could be a precious resource, especially in cases when it is associated with monetary cost. To this end, OpenIoT attempts to cache frequently requested and used data to a store outside the cloud infrastructure (e.g., to a local memory or even local database). The aim is to allow queries to be answered through accessing the cache memory (or local storage) rather than accessing the cloud infrastructure. This access capability is naturally implemented at the Service Delivery and Utility Manager (SD&UM) component of the OpenIoT infrastructure.

In order to understand and quantify the benefits of the caching mechanism, a cost model is needed. This model/function quantifies the cost associated with access to and maintenance of the cache for a given object O, and compares it to the respective cost associated with access to the cloud infrastructure. In the above formulas K and K' denote the cost functions, while c/c' and d/d' denote parameters associated with the monetary cost and the delay/latency associated with each of the access modalities. An OpenIoT service is typically provided using a number of ICOs, where q is the number of objects supporting the delivery of service S_i and n is the number of services running in the system at a given time instant. The caching mechanism

should therefore (at a given time instant) attempt to optimize (i.e. minimize) the following cost quantity where:

- K_0: The initial cost for purchasing and setup of the cache or local storage
- p_{ij} : The probability that service i uses sensor/ICO j.
- O_{ij} : The j^{th} ICO is in use by the i^{th} service.
- l : The number of sensors/ICOs whose data is in the cache or local storage. The capacity of the cache (or local storage placing a limitation on this parameter.
- m : The total number of sensors/ICOs available in the system at a given time instant.

The above cost calculation is time dependent and hence we need to calculate an integrated cost across a given time scale (e.g., days, weeks, months). In order to minimize the cost we need to ensure that the requested services demand/select objects whose data is in the cache with much higher probabilities than those whose data is only accessible via the cloud.

To this end, caching could be based on the following policies:

- **Location based data caching**: Cache data related to popular locations, which involves maintaining statistics about the frequency of accesses to sensors in each location. For example in several applications (e.g., meteorological services) data in popular locations (e.g., capital cities, densely populated cities, monuments, and travel locations) are likely to be accessed more frequently compared to data in other locations.
- **Utility Driven data caching**: This involves caching the data with the highest utility. The latter could be either a user-assigned parameter, or calculated on the basis of the utility metrics specified in D4.2 of the project.
- **Caching based on the frequency of access**: Such a policy is based on tracking of the frequency of access to ICO data. The most frequently accessed data streams are the ones cached that should be cached with high priority.
- **Hybrid approaches using more than one of the above criteria**: Combinations of the above criteria are possible.

The previously listed policies can ensure that services access the cache with higher probability that the cloud infrastructure, thereby economizing on latency and cost. The potential improvement can be benchmarked and quantified based on ground truth from the use of the OpenIoT prototype implementation (based on real or simulated data). To this end, empirical probabilities (relative frequencies) could be estimated on the basis of data sets associated with the operational use of the system [Mood 1974].

3.2.3.3 Caching of sensor/ICO data based on frequency of requests

A specific approach implemented for caching of sensor/ICO data can be based on the frequency by which data of specific sensors are requested by OpenIoT services. The following algorithm can be periodically invoked, where a user/administrator defined the frequency of algorithm invocation:

- All OpenIoT services using the data produced by the same sensors are clustered into respective groups (i.e. equal to the number of sensors). This is possible and supported by the data structures specified in the scope of deliverables D4.1 and D2.3.
- For all groups larger than that a specific (user/administrator configured) number i.e. a number denoting a critical mass of services using one ICO the OpenIoT management system can do the following:
    - Calculate the largest time-window of the data to be requested from all the services.
    - Cache all the data of the sensors for the specific time-window to the SD&UM component.
    - Update the cache (on a specific time interval based on the services) only with the new data available.
    - Check for new services available that comply with the rule, and update the time-window and cache update interval respectively.
    - Check for suspended/disabled services and update the time-window and cache update interval respectively.

3.2.3.4 Caching of entire (SPARQL) Queries

Based on the current OpenIoT architecture, OpenIoT services are associated with SPARQL queries denoting queries over the sensors/ICOs of the underlying IoT infrastructures. Hence, in addition to caching sensor data, caching of SPARQL scripts is also performed, on the basis of «frequency of request» criteria.

- Multi-query optimization of different query types (aggregate, location monitoring, trajectory, point, region)
- Real-time/batch processing
- Data sharing
- Eligible resources per task
- Assigning sensors to queries to maximize a social welfare in the long-run
- Efficiently announcing sensor capabilities

The objective is to provide rapid access to frequently requested queries/services, thereby economizing on the time needed to construct, validate and deploy the services. Caching is performed at two levels:

- At the level of a complete SPARQL query, note that the probability of requesting exactly the same query (in a short timescale) is relatively low.
- At the level of the execution plan of a SPARQL query and the individual sub-queries that it comprises.

3.2.3.5 SD & UM Requests Caching Scenarios

Probably the most significant drawback of using triple stores for the deployment of semantic web technologies is their performance. In comparison to relational databases, there is an obvious trade-off between flexibility in information structuring and raw performance. In order to approach the performance of relational databases, implementing a caching solution would significantly increase the performance of triple stores [Martin 2010].

Besides performance however, accessing remote data-stores like Amazon S3 or Google Cloud Data-store, usually incurs **an extra cost** depending on the provider's pricing scheme**.** Indicatively, Amazon S3 charges $0.005 per 1,000 (Amazon Inc.) per request, while Google Cloud charges $0.01 – $0.09 per 100k operations (an operation may include a variable number of requests depending on the type) (Google Inc.). It is therefore understandable, that besides increasing performance, caching may potentially decrease the overall cost of operating the LSM Cloud Datastore.

In the context of **T5.2 Resource Sharing & Management** and in order to assess the attainable level of cost reduction, a simulation is performed that is based on the Erfurt Semantic Web Application Development Middleware [Martin 2010] as explained in the following sections.

3.2.3.5.1 Erfurt Caching Architecture

The architecture of the Erfurt Middleware Architechture is illustrated in Figure 5:

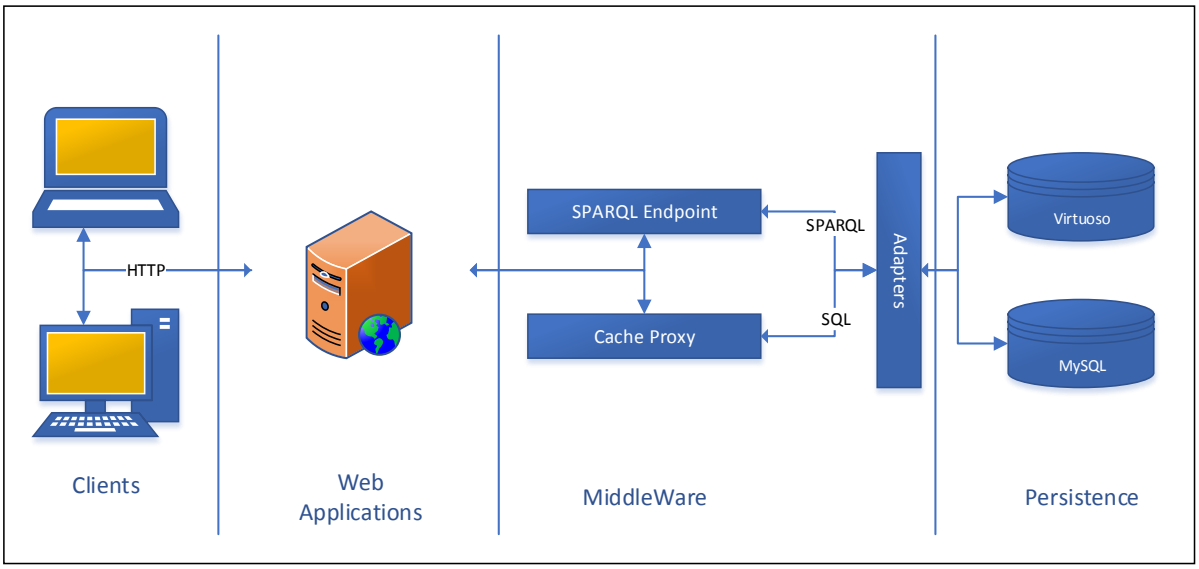


Figure 5. SPARQL Cache Architecture.

The specific implementation implements a small proxy layer, between the Web Application and the SPARQL data-store endpoint (Figure 5. SPARQL Cache Architecture). All SPARQL queries are routed through this proxy. When a query is entered into the system the proxy layer checks if the result has already been cached. In such a case, the result is returned to the client directly through the cache without accessing the SPARQL data-store. In any other case the query is redirected to the SPARQL data-store and the result is stored in the local cache before it is returned to the client.

3.2.3.5.2 Cache Population and Maintenance

In general the architecture of caching solutions is quite simple and analogous to the approach implemented by the Erfurt Middleware. Each object cached at the proxy layer must be uniquely identifiable. At certain time intervals certain objects may be invalidated. It is important however, that in contrast to caching implementations from conventional web applications, cache objects are also invalidated based on updates on the triple store. Additionally, it is important to cache objects of increased complexity that are aggregators of multiple query results [Martin 2010]. An indicative cache object schema is visualized below in Figure 6.

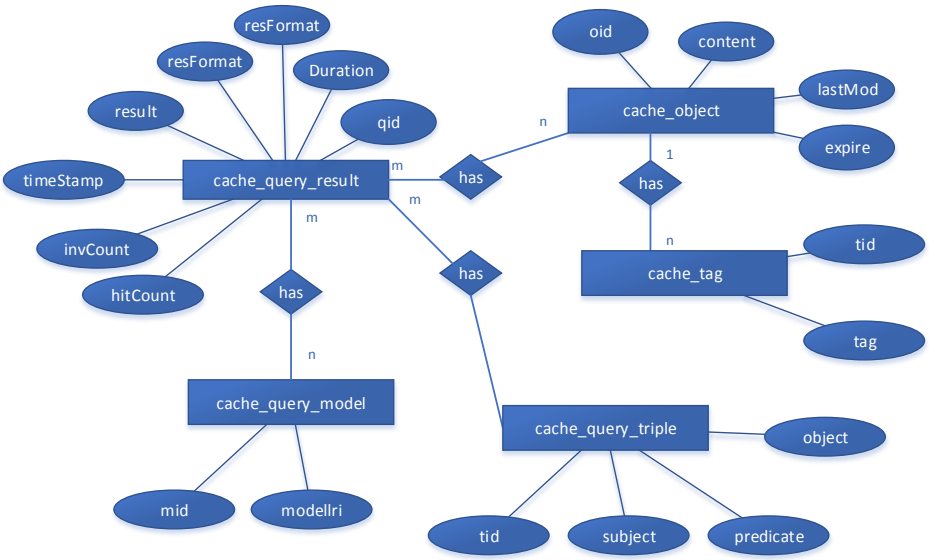


Figure 6. Cache Population ER Diagram.

3.2.3.5.3 Berlin SPARQL Benchmark Results

In order to evaluate the Erfurt caching solution its querying performance was measured against the **Berlin SPARQL Benchmark** [Bizer 2009]. The Berlin SPARQL Benchmark is based on an e-commerce use case, simulating an end-user search for products, vendors and reviews. The resulting SPARQL queries are grouped into mixes, each one consisting of 25 queries. The queries are derived from twelve different types and are instantiated by replacing parameters with concrete, randomized values. The QueryMixes per Hour (QMpH) assessment then states, how many of these query mixes a certain triples store is able to execute per hour. [Bizer 2009]

While in the original benchmark the probability for selecting a specific parameter is equal for each parameter, in the cache benchmark the parameters are selected according to the Pareto distribution, since this reflects practical use cases better and enables the measurement of performance gain in such scenarios. The probability density function that models the level of the cache hit rate increase according to the increase of QMpH can be described by the following formula [Martin 2010]:

$$P(x) = \frac{ab^a}{x^{a+1}}$$

The parameter *a* defines the distribution, whereas *b* defines the minimum value. Applied to the benchmark scenario, this implies that we have a number of products or offers that are queried more often than others. In the current benchmark implementation, parameter *a* was varied in order to see how well the caching

implementation adopts to a *wider* or *narrower* spectrum of repeated queries. For the Pareto principle (commonly known also as the 80/20 rule of thumb),

Table 2 shows how the choice of *a* broadens the distribution of the parameter (based on a benchmark with 10 million triples and 12,500 queries).

Table 2. Cache hit/miss rate in relation to the choice of parameter *a*

Distr. Parameter	**linear**	**a = 0.1**	**a = 0.3**	**a = 0.5**	**a = 1.0**	**a = 2.0**	**a = 4.0**
Unique Queries	11718	6205	4147	2953	1694	624	142
Res Distribution	50.5/49.5	64/36	72/28	78/22	84/16	88/12	90/10

Therefore we can see that the wider the variety of queries, we have fewer unique queries that are serviced directly from the LSM repository rather than the cache.

3.2.3.5.4 Application of Scenario on Cloud Datastores

As mentioned previously, a very important issue of using cloud data-stores is the price per request payment scheme. Therefore, minimizing the number of requests that occur directly on the LSM cloud repository which drastically reduce the overall operational costs.

In the context of **T5.2 Resource Sharing & Management**, in order to demonstrate and simulate scenarios associated with cloud data-store access and caching solutions, a spreadsheet calculator prototype has been created that models the various costs. This section examines such a particular scenario, aiming to determine to what level a caching solution may provide benefits to the overall cost efficiency of the system. In particular, this scenario is based on usage of the Amazon S3 Cloud Data-store, which can be modelled easily, since it is based on a linear pricing scheme ($0.005/1000 requests). Additionally, the particular scenario also assumes the cache miss rates previously displayed in

Table 2.

According to

Table 2, there are 7 scenarios examined that concern the cache hit/miss rate, according to the *a distribution parameter*. These scenarios are:

- linear distribution
- a = 0.1
- a = 0.3
- a = 0.5
- a = 1
- a = 2
- a = 4

In the case of linear distribution, cache miss rate is at almost 50% while in the other scenarios, the higher the *a distribution parameter,* the lower the miss rate. These scenarios are compared primarily to the first column group on the chart where no caching is used.

Additionally, for this scenario, yearly server operational costs that support a 20TB cache have been taken into account. It is assumed that the server storage capacity is sufficient to store all the query results obtained from the cloud data-store. For this server setup the yearly cost of ownership is visualized in

Table 3:

Table 3. Caching Scenario - Server Cost of Ownership.

Caching Server Cost / Unit	
Server Disk Capacity (TB)	5
Unit Cost(€)	3500
Lifespan (years)	3
PV Discount Rate (%)	5
Server Maintenance/year (€)	1500
Energy cost / year (€)	1000

Server Cost / Year (Present Value)	
Cache Size Required (TB)	20
Servers Required	4
Cost / Year (PV)	**22.635,00 €**

Applying the above scenario, it is shown in Table 3 how the cost changes depending on the variety of the query spectrum. All the categories that are displayed on the horizontal axis in Figure 7 **except the "no cache"** category include cumulative costs. This means that all the other categories have yearly server costs included along with datastore usage costs.

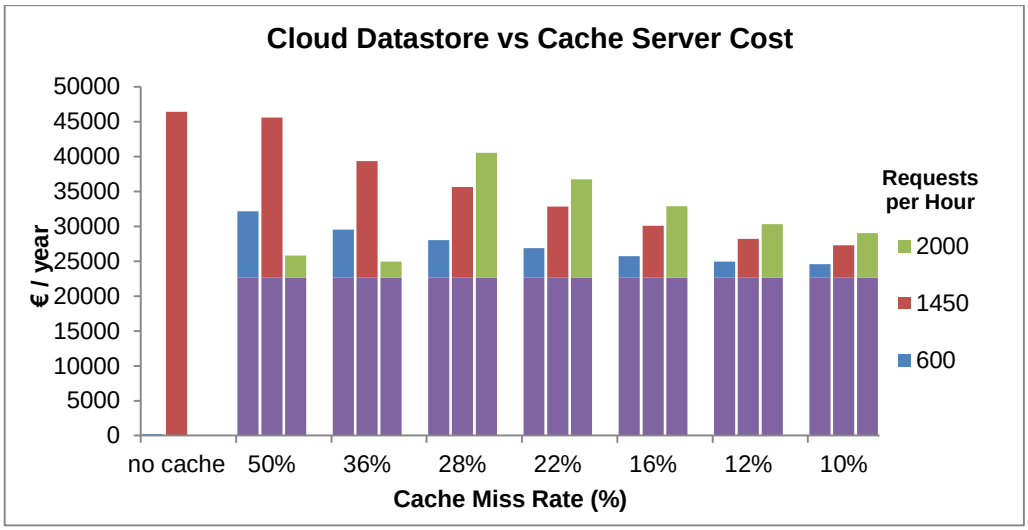


Figure 7. Amazon S3 Data-store Prices Based on Caching and Spectrum Width.

As expected, for a low number of requests per hour, there is no benefit in using a cache. In the first category at **600 Krph**, even with a cache miss rate at 10% (a=4.0) it is still preferable not to use any caching at all. In contrast, it is actually quite inefficient to use a cache server at that level, since the costs is even greater. Even at a medium-high number of hourly requests such as the second category at **1450 Krph** the scenario just hits the threshold where it becomes more efficient to use caching. In the final category at **2000 Krph** it is finally evident that at a high number of requests, it is far more efficient to use the cache. This is visible at the extreme situation with a 10% cache miss rate, where the data-store usage costs for the high request category at **2000 Krph,** are marginally higher than even the low category with 1/3 of its rph (**600 Krph**).

Consequently, in order to achieve an efficient caching solution there must be a clear estimate first of all, of the average requests per hour on the cloud data-store, as well as to what extent the caching storage capacity is sufficient.

Finally, it is also evident by this simulation that the determining factor for cache performance is not the absolute number of queries. Rather, it is the variety of different queries that are performed on the cloud data-store in order to quickly build up the cache.

3.3 Cloud Optimization

Large amounts of data coming from ICOs and (virtual) sensors are expected in the context of OpenIoT. This is a result of both the number of potential *entities* or *things* that provide data to the cloud, and the high rates at which they can push measurements. Such scenario that combines large volumes of data and high velocity of data calls for scalable data management and querying solutions, spanning multiple storage backends and processing units in the cloud. However, this is not a straightforward task, as most cloud-storage systems are designed for batch processing (e.g. Hadoop) or stored static data (e.g. HBase). In the use cases of

OpenIoT, these solutions are not adequate, as they would need to store all the raw data measurements in the cloud, which is not efficient in terms of storage and can potentially saturate the data backend.

In this section we propose a cloud-based framework for sensor data management, which optimizes storage and querying of sensor time series measurements. In particular, we exploit key-value stores and the MapReduce parallel computing paradigm, two significant aspects of cloud computing, to realize indexing and querying model-view sensor data in the cloud. In order to process range or point queries on model-view sensor data, our KVI-index in the cloud store has shown good performance in processing interval data, while current key-value built-in indices do not support interval related operations. The interval index for sensor data management not only works on static data sets, but it is dynamically updated based on the new arriving segments of sensor data.

Various sensor data segmentation and modelling algorithms have been extensively researched, such as PCA, PLA, DFT, etc. [Guo 2012], [Papaioannou 2011], [Ding 2008]. The core idea is to fragment the time series from one sensor into modelled data segments, and then approximates each data segment by a mathematical function with certain parameters [Guo 2012], [Papaioannou 2011], [Ding 2008], such that a specific error norm is satisfied. The chosen mathematical model for each segment takes as dependent variable the sensor value and as independent variable the time-stamp. For simplicity, we refer to the modelled segment as a segment in the rest of this deliverable. For example, in Figure 8 (a), the time series from a mobile accelerometer sensor is divided into eight disjoint segments each of which is modelled by a linear regression function and has associated time domain and value range shown in Figure 8 (b). For model-view sensor data management, only the segment models are materialized and therefore the query processing is performed on the segments instead of the raw sensor data, as in [Thiagarajan 2008].

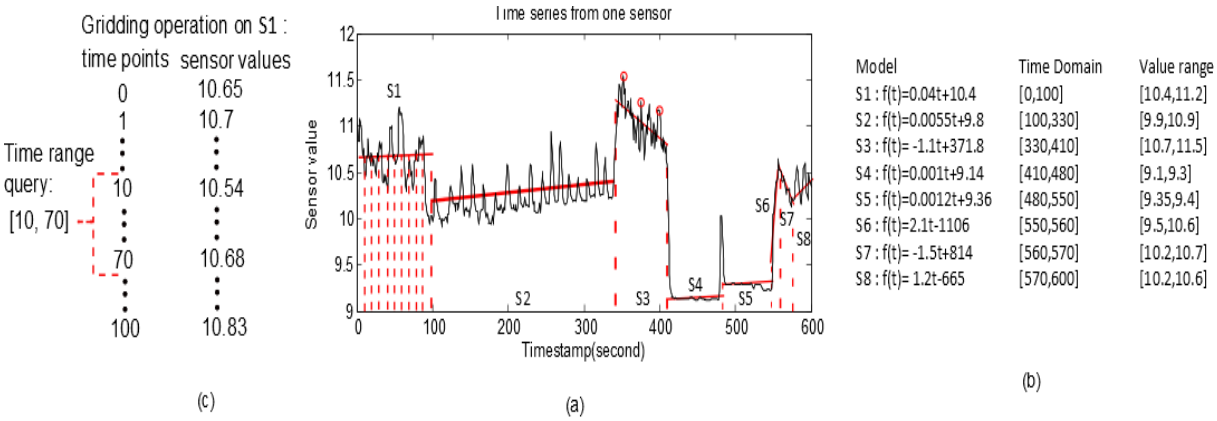


Figure 8. (a) Model view sensor data. (b) Polynomial models of segments. (c) Query processing on the gridded segment.

We exploit key-value stores and the MapReduce parallel computing paradigm, two significant aspects of cloud computing, to realize indexing and querying model-view sensor data in the cloud. One modelled segment is characterized by its time and value intervals [Deshpande 2006], [Thiagarajan 2008], [Papaioannou 2011] which enables us to design a distributed interval index for querying sensor data segments. The supported categories of queries on model-view sensor data to are as follows:

Time point or range query: return the values of one sensor at a specific time point or during a time range.

Value point or range query: return the timestamps or time intervals when the values of one sensor are equal to the query value or fall within the query value range. There may be multiple time points or intervals of which sensor values satisfy the query predicate.

The contributions of our work can be summarized as follows:

Innovative interval index: We propose an innovative interval index for model-view based sensor data management in key-value stores, referred to as the KVI-index. The KVI-index is a two-tier structure consisting of one lightweight and memory-resident binary search tree and one index-model table materialized in the key-value store. This composite index structure can dynamically accommodate new sensor data segments very efficiently.

Intersection search: We introduce an enhanced intersection search algorithm (iSearch+) that produces consecutive results suitable for MapReduce processing.

Hybrid modelled-segment query processing: After exploring the search operations in the in-memory structure of the KVI-index for range and point queries that locate modelled segments that may satisfy the query, we introduce a hybrid query processing approach that integrates both range scan and MapReduce to process these segments in parallel and identify the qualified ones.

3.3.1 Key-Value Interval Index

Our KVI-index is a novel in-memory and key-value composite index structure. The virtual searching tree (vs-tree) resides in memory, while an index-model table in the key-value store is devised to materialize the secondary structure (SS) of each node in vs-tree.

3.3.1.1 In-memory structure

The in-memory vs-tree is a standard binary search tree shown in Figure 9 (a). Each time (or value) interval is registered on only one node of vs-tree, which is the one with the interval first overlaps along the searching path from root. This node is

defined as a registration node for the interval to index. Each node of vs-tree has an associated secondary structure (SS), materialized in the key-value store, which stores the substantial information of the modelled segments registered at this node.

All the operations on vs-tree are performed in memory and are thus very efficient. As the domains of time and value of the sensor data are different, two vs-trees, one for time stamps and another for values, are kept in memory simultaneously for answering time and value queries respectively.

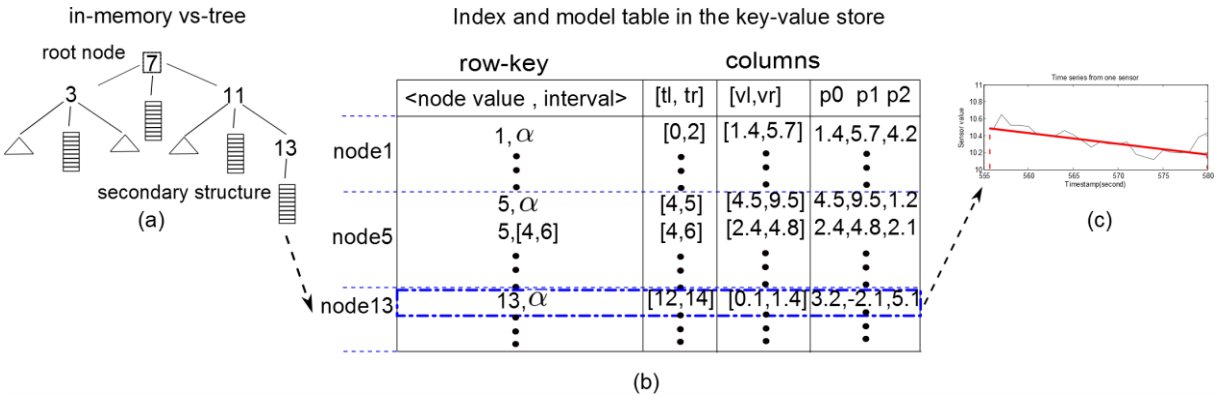


Figure 9. (a) In-memory vs-tree. (b) Index-model table in the key-value store. (c) One segment of sensor data.

3.3.1.2 Index-model table

We designed a novel index-model composite storage schema, which enables one key-value table not only to store the modelled segments, but also to materialize the structural information of the vs-tree, i.e., the SSs for each tree node.

The index-model table is shown in Figure 9 (b). Each row corresponds to only one modelled segment of sensor data, e.g., the data segment shown in Figure 9 (c). A row key consists of the node value and the interval of an indexed segment at that node. One modelled segment's time, value interval and coefficients are all stored in different columns of the same row.

3.3.1.3 KVI-index updates

The complete segment-updating algorithm of KVI-index includes two processes: registration node searching and materialization of modelled segments.

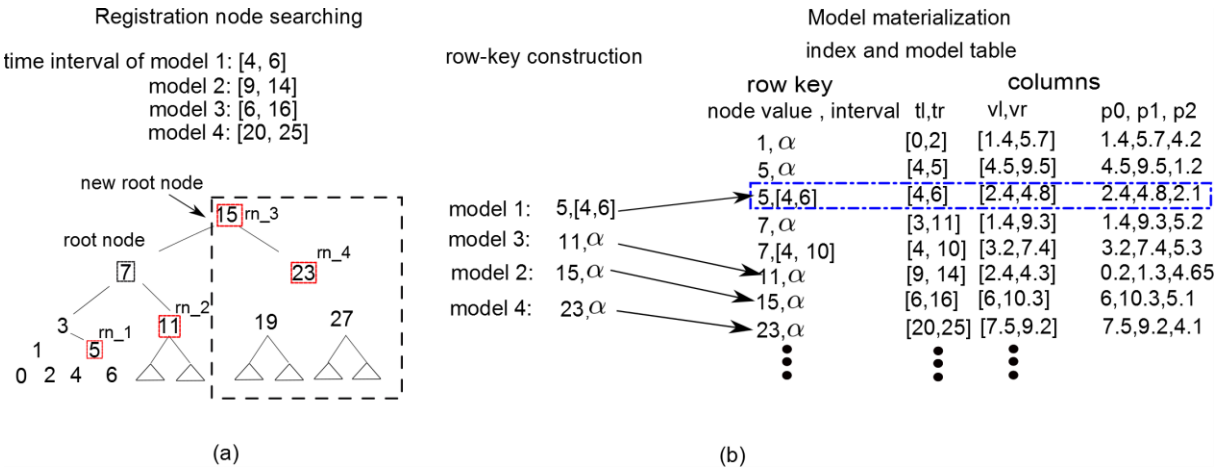


Figure 10. (a) rSearch. (b) Segment Materialization.

Registration node searching (rSearch)

As the segment model of sensor data is generated in real-time, the time (value) domain of vs-tree should be able to catch up with the variation of that of sensor data. Therefore, the update algorithm first involves a domain expansion process to dynamically adjust the domain of the vs-tree according to the domain variation of the sensor data. Then, the registration node can be found on the validated vs-tree. The complete rSearch algorithm can be illustrated by Figure 10 (a).

Materialization of modelled segment

When materializing one segment into the SS of a node, the row-key may be chosen in two ways. When no modelled segment has been stored at that SS, the row key is the concatenation of the binary representations of the registration node and postfix for the segment. When the SS has already been initialized, the time or value interval of one segment to index is incorporated into the row key. In this way, different segments stored in the same SS of a node do not overwrite each other.

3.3.2 Query Processing via KVI-index and MapReduce

In order to query model-view sensor data, the searching process of qualified segments in KVI-index includes intersection and point searches which are responsible for collecting the nodes that accommodate qualified segments in their secondary structures SSs. Afterwards we design a novel hybrid parallel computing and sequential scan approach for model filtering and gridding.

3.3.2.1 Enhanced interval intersection search

Given a time (resp. value) range query, iSearch+ first calls the rSearch to find the registration node of the query range. The nodes on the searching path from the root node to the one preceding the registration node form a node set. The iSearch+ stops

at the node, which is closest to the left-end point. All the nodes along the left-descending path form a node set. Analogously, the nodes from the right-descending path form another node set. Any node outside the search path does not have any qualified segments.

3.3.2.2 Point search

We denote the point search by sSearch as it functions as the stabbing search in interval data management. The sSearch is a binary search that records the nodes along the descending path. Since there is no split searching, as in iSearch+, only one node set is produced here.

3.3.2.3 Hybrid KVI-Scan-MapReduce query processing

Our idea is to design a hybrid KVI-Scan-MapReduce paradigm that combines both range scan and MapReduce for processing SSs.

The height of vs-tree is bounded, and thus the amount of computation is limited. As the SSs are sparsely distributed in the index-model table and each SS can be considered as a small range of clustered index, the random-access and range-scan based model filtering and gridding is suitable. The successive range from left and right path sub-search delimits the tight boundaries of the sub-index-model table over the relevant SSs that are suitable for distributed processing with MapReduce. This hybrid paradigm eliminates the Map-phase processing of SSs of irrelevant nodes. Moreover, it is non-intrusive for both the key-value store and MapReduce.

The functionalities of mappers and reducers are depicted in detail below.

Mapper: Each mapper gets the time (resp. value) interval of one segment to check whether it intersects with the query time (resp. value) range. The qualified segments are sent to the next reduce phase.

Reducer: Each reducer receives a list of qualified modelled segments. For each segment, the reducer invokes a model gridding function to compute discrete values for constructing query results.

Regarding the scan-based model filtering and gridding, as SSs are located in different regions of the index-model table, the query processor makes use of thread pool to process each SS in parallel**.**

3.4 Utility Based Optimization

In the OpenIoT context heterogeneous mobile and stationary sensing devices co-exist. This heterogeneity makes it complex to efficiently acquire the data from the different virtual and real sensors that feed the OpenIoT cloud. It is desirable that based on the query requirements, the system could be able to optimize the

acquisition of data from the available sensors. However, this is not a trivial problem, given the different user expectations, costs, type of queries, etc. In this section we formulate the optimal data acquisition problem as a multi-query optimization with the objective of maximizing the total utility and propose efficient heuristic solutions for various query types and query mixes. This section is based on the theoretical and experimental results described in detail in [Riahi 2013].

According to the OpenIoT architecture, the sensing devices communicate with GSN, which in turn pushes the sensed data and their metadata to the cloud storage. The OpenIoT Scheduler consults the cloud storage and finds out about the available sensing data and the metadata. For the sake simplicity, in this section *we ignore the intermediaries between sensors and the scheduler and assume that sensors communicate directly with the scheduler*. In order to enable utility-based optimization, sensing devices are expected **to take measurements only when they are selected by the scheduler** to do so. We also make the following assumptions: (i) sensing device owners ask for a payment for each provided measurement. (ii) Each sensor has a specific sensing range. (iii) Each measurement includes a sensor-specific inherent inaccuracy. In this section, we use the term *sensor* to refer to the actual sensor on the sensing device, the sensing device, or even the combination of the sensing device owner and the device she carries.

According to the OpenIoT architecture, end users submit queries to the scheduler by defining services. The scheduler periodically collects the queries and tries to answer them in an optimal way. The challenge is how to answer queries based on the data availability and the capabilities of various sensors that may belong to different sensor deployments. Therefore, we take a utility-driven approach, which aims at maximizing the total utility for the queries posed by the end users. Utility maximization can be achieved by selecting appropriate sensors for providing measurements, considering the value of the measurements to the queries, the cost of obtaining such measurements, and exploiting possible common data requirements among queries.

In the context of OpenIoT with diverse sets of end users who have different criteria for evaluating the quality of the query results, ideally the scheduler relies on the end users to provide a valuation function, $v_q(.)$, with each query q. This function returns the value of a set of measurements, which can be used as the answer to the query. Users have a limited budget to spend for obtaining query answers. It is assumed that the amount that can be paid in return to a specific response quality is embodied in the valuation function of the query. This means that the return type of $v_q(.)$ is of the unit that is used for issuing payments. However, if end users are not experts in defining the valuation functions, they can select one from a predefined set of valuation functions when defining the services. Valuation functions can also be assigned by the scheduler or the request definition module if the user wishes so.

Queries defined by end users can generally fall into two major categories, namely *one-shot queries* and *continuous queries*:

- **One-shot queries** are executed only once. Major one-shot queries in our context are *point queries*, *spatial aggregate queries over a region*, and *queries over trajectories*.
- **Continuous queries** are continuously evaluated, and can be split into the two sub-categories of *monitoring queries* and *event detection queries*.

Single-sensor queries only need one sensor reading while *multi-sensor* queries need multiple sensor readings.

3.4.1 Problem Formulation

Without loss of generality, we assume that the system runs for a period of T, e.g., from 6 a.m. to 9 p.m. in a day. This period is discretized into several time slots of fixed length, e.g., 5 minutes. We assume that all the sensors connect to a unique scheduler and, if necessary, at the beginning of each time slot announce their location and price of providing a measurement at that location.

The objective is to acquire data for the queries from the available sensors in order to maximize the utility over T. Formally, we let Q denote the set of all queries issued from time 1 to T, $\mathcal{S}^t$ denote the set of available sensors at time slot t, and $K: Q \rightarrow \times_{t=1}^{T} 2^{\mathcal{S}^t}$ define an allocation scheme that assigns sensors to each query. $Y(K,t)$ is a function that returns the set of sensors that are assigned to all queries at time t. We denote by $c_s(K,t)$ the cost of sensor s at time t given the allocation K. Let $\mathcal{K}$ denote the set of all possible allocation schemes. The goal is to find allocation $K^* \in \mathcal{K}$ that maximizes the *social welfare*:

$$K^* = \underset{K \in \mathcal{K}}{\operatorname{argmax}} \left(\sum_{q \in Q} v_q\big(K(q)\big) - \sum_{t=1}^{T} \sum_{s \in Y(K,t)} c_s(K,t) \right)$$

For solving the above optimization problem we need to know in advance all the queries that are issued over T, and the location and cost of all the sensors at each time slot. However, in the context of OpenIoT, users must be able to submit new queries whenever they desire and it is not realistic to ask the users to pose all their queries in the beginning of the period T. Due to the uncontrolled mobility of the (mobile) sensors, their exact locations at a specific time slot cannot be determined a priori. Moreover, the cost of a sensor might vary from one time slot to another based on the preferences of the sensor owner. Due to the lack of access to all the required information to solve the above long-term optimization problem, we resort to a *myopic* approach, in which we try to maximize the utility at the current time slot without considering the future state of the system. In this approach, when finding the optimal allocation scheme, we only consider the queries and sensors that are available during the current time slot. After finding the best allocation scheme, the cost of each

selected sensor is shared among queries that are answered using the measurement from that sensor.

We solve the myopic multi-query optimization problem for the following query types:

- Single-sensor point queries, for which we provide optimal and approximate solutions.
- Multiple-sensor one-shot queries including spatial aggregate queries, queries over trajectories, multiple-sensor point queries, etc.
- Continuous queries
    - Location monitoring queries
    - Region monitoring queries
- Mix of the above query types

The algorithms that we used for achieving utility-based optimization for the abovementioned query types are available in [Riahi 2013].

3.4.2 Cost Computation

Sensors owners participate in the system as long as the resource consumption on their devices as well as their location privacy loss are compensated. In this regard, each sensor asks for a certain price in return for providing a measurement to the aggregator. Therefore, the cost of obtaining a measurement from sensor s which is located at l_s, consists of two components as demonstrated in the following equation:

$$c_s(\mathcal{E}_s, H_s, l_s) = c_s^e(\mathcal{E}_s) + c_s^p\big(p_s(H_s, l_s)\big),$$

where $\mathcal{E}_s$ is the remaining energy, and H_s is the history of revealed locations of s. c_s^e is a function that gives the energy cost of taking a measurement and transmitting it to the aggregator, and c_s^p is a function that calculates the cost of the sensor's privacy loss due to revealing its location. The privacy loss is computed by the function p_s. We do not impose any restrictions on the form of these two functions. When this cost function is not available from the sensors, a default cost function can be assigned by the scheduler to the sensor. Note that $\mathcal{E}_s$ can also represent the energy consumption rate of the sensor depending on the type of sensors. The function must be selected by the scheduler considering available constraints of the sensors and the energy/bandwidth optimization objective.

3.4.3 Valuation functions

Generally, the value of a sensor reading for an application is a function of the quality of that sensor reading and the quality of the sensor readings obtained so far. The number of samples required for finding the value of a phenomenon depends on the phenomenon itself and the trustworthiness of the sensors. For example, it might be necessary to take redundant measurements to assess the trustworthiness of a particular sensor that can be used for providing the measurements. For instance, a single-sensor point query q might have the following valuation function:

$$v_q(s) = \begin{cases} B_q \theta_{q,s}, & \theta_q^{min} \leq \theta_{q,s} \leq 1 \\ 0, & \theta_{q,s} < \theta_q^{min}, \end{cases}$$

where $0 \leq \theta_{q,s} \leq 1$ is the quality of the sensor reading, θ_q^{min} is the minimum acceptable quality by the query, and B_q is the query budget. This implies that the user is willing to pay B_q for a sensor reading with the highest possible quality.

The quality of a sensor reading depends on the distance of the sensor from the queried location (more accurately, it depends on the correlation between the phenomenon value at the queried location and the location of the sensor,) the inherent sensing inaccuracy, and the trustworthiness of the sensor. We assume that this dependency is given by a user-defined function $v_q(s, l_q)$, where l_q is the queried location. The following is an example of such a function:

$$v_q(s, l_q) = \begin{cases} (1 - \gamma_s)\left(1 - \frac{|l_s - l_q|}{d_{max}}\right)\tau_s, & |l_s - l_q| \leq d_{max} \\ 0, & otherwise, \end{cases}$$

where γ_s is the inaccuracy of s measured in percentage of the value range of the sensor, $0 \leq \tau_s \leq 1$ is the trustworthiness of s, l_s is the current location of s, and d_{max} is the maximum distance in which the sensors can be considered to provide data.

For spatial aggregate queries, which need more than one sensor, and sample valuation function could be the following:

$$v_q(S_q) = B_q \mathcal{G}_q(S_q) \frac{\sum_{s \in S_q} \theta_s}{|S_q|},$$

where $\mathcal{G}_q$ is a function that calculates the coverage of the selected sensors. A simple coverage function can calculate the fraction of the area covered by the sensors, while a more general function might also take into account the dispersion or the importance of the locations that are covered by the selected sensors.

3.4.4 Experimental Evaluation

We used a real mobility dataset from Nokia campaign in Lausanne, Switzerland. The simulations are run for 50 time slots. Figure 11 shows the average utility per time slot achieved by different algorithms when we have only point queries. Figure 12 illustrates the average utility per time slot achieved by of our algorithm compared to a baseline algorithm for spatial aggregate queries. Similar results for location monitoring and region monitoring queries are illustrated in Figure 13 and Figure 14, respectively. Figure 15 shows the average utility per time slot achieved by our multi-

query data acquisition algorithm and a baseline algorithm when a mix of queries of different types is available.

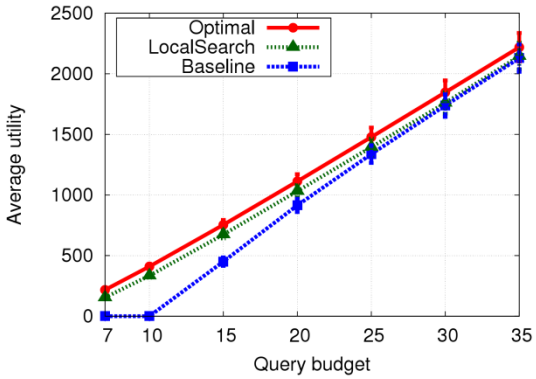


Figure 11. Average utility per time slot having only point queries.

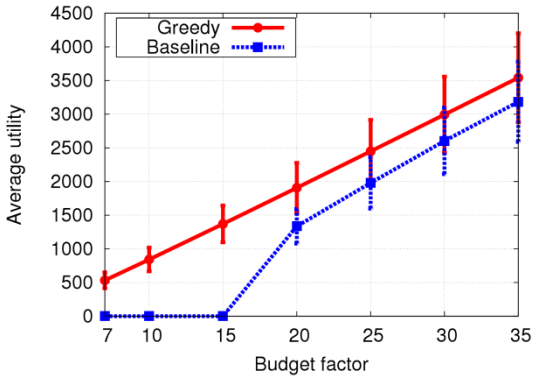


Figure 12. Average utility per time slot having only spatial aggregate queries.

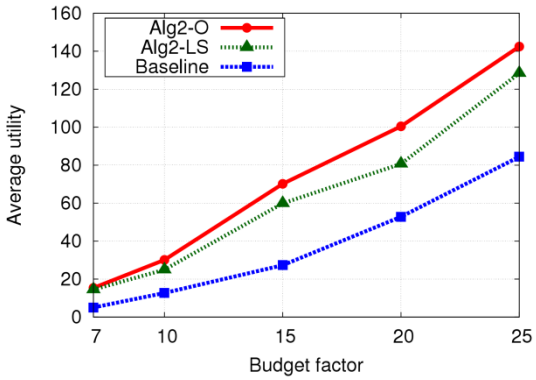


Figure 13. Average utility per time slot having only location monitoring queries.

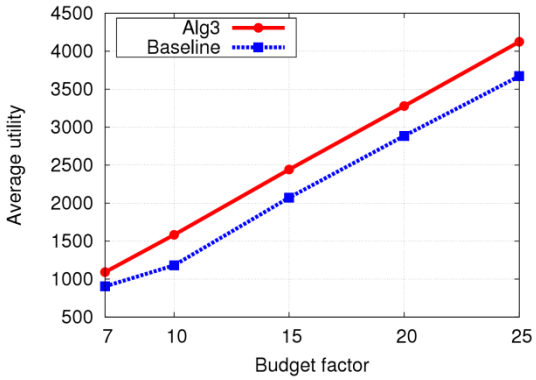


Figure 14. Average utility per time slot having only region monitoring queries.

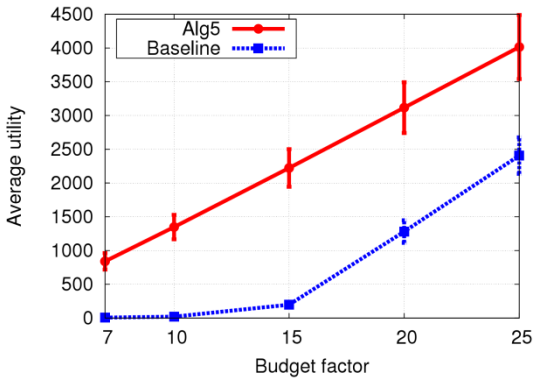


Figure 15. Average utility per time slot having a mix of point, aggregate, and location monitoring queries.

3.5 Efficient Sensor Data Collection

In the area of ICOs big advances have been realized to enable ICO control (mainly over the sensor networks data). However, full control over the data of multiple devices has not been implemented yet, nor intelligent data services have been deployed. In OpenIoT we seek for deployment of stream data processing components enabling the deployment over multiple infrastructures (openness feature). An efficient set of methods for data acquisition from heterogeneous sensors, both static and mobile, allows filtering of incoming sensor data, or selecting relevant sensor data sources.

In this context, we present two main contributions. The first (Section **Error! Reference source not found.**) is an approach for efficient sensor data acquisition, based on the utility-based optimization described in Section 3.4. The second is a context-aware acquisition and filtering approach for mobile sensors, detailed in Section 3.5.2.

3.5.1 Utility-based Sensor Data Acquisition

As we described in Section 3.4, based on utility functions, we can define optimization schemes that maximize the total welfare for sensor data acquisition. One specific outcome of this effort is a data acquisition framework that efficiently shares sensor data among queries of different types. This framework optimizes the usage of sensors, choosing them in such a way that the global utility is maximized.

In particular, the utility-based data acquisition approach is able to select a subset of sensors S' from the available set of sensor S, in such a way that a given utility function is maximized. In this way, the system avoids acquiring data from (virtual) sensors which are not needed by the queries posed by the users, or whose contribution to the total utility is marginal. The cost computation and valuation functions have been described in Section 3.4, and are provided as an input to the data acquisition algorithms described here.

Nevertheless, and as it was explained above, the algorithms for data acquisition vary depending on the type of queries that are received from users and/or applications. We describe below the main characteristics of these data acquisition algorithms, classified according to the type of query: single-sensor point query, multiple sensor one-shot query, continuous queries, and a query mix. These types of queries are commonly found in a pervasive sensor infrastructure such as the one in the OpenIoT context. The full description of the algorithms can be found in [Rihai 2013].

Single-Sensor Point Queries: In the context of OpenIoT, these queries are limited to observations that are available in one particular sensor (notice that the sensor may be virtual). For this type of queries, we can express the optimization of sensor allocation as a Binary Integer Linear Problem (BLIP). For his case, an ILP solver can find the optimal solution, if the input size is not too large. On the other hand, if the

input size is large, a Heuristic Scheduling approximation algorithm is proposed. This algorithm [Feige 2007], referred to as the Local Search algorithm, has been devised to solve non-monotone submodular functions, as it is the case with this optimization problem.

Multiple-Sensor One-shot Queries: In this case, the queries received have different data requirements: multiple sensor observations are needed to be able to answer them. This is the case for queries that operate over trajectories or over a spatial extent. Moreover, the optimization function must be able to take advantage of the possible overlapping of sensors readings (e.g. they might cover contiguous areas or have other different topological relationships) and the value that each sensor contributes to a query. This turns out to be a combinatorial problem: a sensor assignment that maximizes the overall benefit must be selected, out of all possible ones. The proposed greedy algorithm iteratively selects sensors that maximize the partial overall utility. This algorithm has been shown to be faster and outputs a better total utility if the utility functions are not submodular.

Continuous Queries: The proposed acquisition algorithms for continuous queries target location and region monitoring queries. In both, the continuous nature of the query implies a time period when the monitoring is performed, as well as a sampling time. These algorithms attempt to get sensor observations according to the frequency of the sampling time. Due to uncertainty, it is not guaranteed that data is acquired for the required sampling time, so data can be acquired at other times, but with a fraction of the expected value. In the case of the location monitoring queries, a point query is created at every time slot; then a set of sensors is selected for those point queries and for each sensor the correspondent payment is calculated.

In the case of region monitoring queries, sensor data is possible if the regions over which the queries are executed overlap. Several queries may share subsets of sensors (e.g. two queries requesting temperature values in the same area), or sensor can provide similar data (e.g. two sensors providing humidity measurements in the same location). A modified algorithm can take advantage of this, by providing a set of weighted costs of sensors. As an example, if a subset of sensors was already selected by another continuous query, then a weight of 0 can be assigned to that subset of sensors.

Query Mix: When the aggregator receives queries of different types, it has the possibility of sharing the sensors among them and hence increasing the total utility. Indeed, since individually finding an optimal set of sensors for multiple point or aggregate queries is NP-Complete, finding the optimal set of sensors for the combination of queries is also NP-Complete. The proposed algorithm for the query mix selects sensors by exploiting the commonalities of the queries posed to the system. It first generates point queries for location and region monitoring queries. Then, all queries are provided to the greedy algorithm used for multiple sensor one-shot queries, so that it optimizes the total utility. Afterwards, the results of the point

queries are applied for continuous queries. As in this stage there might be queries sharing the same sensors (e.g. regions overlapping), the payments need to be adjusted accordingly. Finally, the selected sensors are requested to provide their observations.

3.5.2 Context-Aware Acquisition and Filtering of Sensor Data in Mobile Environments

Publish/subscribe middleware offers the mechanisms to deal with the challenges related to continuous context-aware and energy-efficient acquisition and filtering of sensor data in mobile environments, specifically in scenarios requiring opportunistic mobile sensing that can potentially generate huge volumes of sensor data. Note that this data needs to be transmitted into the cloud over mobile devices for which battery and bandwidth are limiting resources. Thus we need to devise strategies to minimize the number of data transmissions to the cloud while maintain adequate sensing coverage for mobile sensing applications. Publish/subscribe middleware provides the means for selective acquisition of sensor data from mobile wearable sensors as well as filtering of sensor data on mobile devices prior to its delivery into the cloud for further processing.

In this subsection we present the main concepts of a publish/subscribe component running on mobile devices entitled Mobile Publish/Subscribe (MoPS). MoPS enables selective sensor data acquisition and filtering in IoT environments where mobile devices are applied as gateways for collecting and transmitting sensor data into the cloud, while at the same time mobile devices receive the data of interest from the cloud. In contrast to existing centralized database solutions which typically send all sensed data into the cloud, MoPS supports *flexible and controllable acquisition of data* and its subsequent transmission into the cloud only in situations when the sensed data is indeed required by the the back-end system, i.e., the cloud. In other words, the data should be produced and transmitted to the cloud only if it is valuable, e.g., there is current interest by system users to be alerted about certain events, or the data is needed for the data-mining tasks.

MoPS provides *content-based filtering of sensor data on mobile devices* based on context, e.g., current data needs specified by application users, sensing coverage, available bandwidth, or QoS-specific parameters defined by an application. Moreover, it can even suppress the sensing process on wearable sensors. Similar to [Sadoghi2011], MoPS supports a rich predicate language with an expressive set of operators for the most common data types: relational operators, set operators, prefix and suffix operators on strings, and the SQL BETWEEN operator. Hereafter we explain the MoPS model and underlying design principles. Further details on MoPS design and implementation are available in deliverable D3.4.1.

Publish/Subscribe Model. The MoPS model comprises a set of publishers, P_i, and a set of subscribers, S_j, that interact over a hierarchical two-tier publish/subscribe network composed of *mobile brokers*, MB_k, and a cloud broker, *CB*. An example

model is shown in Figure 16. Publishers, e.g., wearable or built-in sensors, *publish* data items and send them either to mobile brokers or directly to a cloud broker. Subscribers, e.g. processes on mobile devices, can activate and dismiss subscriptions by sending messages *subscribe* and *unsubscribe* to mobile or cloud brokers, which in turn use the message *notify* for push-style delivery of matching data items, i.e., items that satisfy subscription constraints, to subscriber processes.

A cloud broker is responsible for efficient matching of data items to active subscriptions as well as their subsequent delivery to either subscribers, mobile brokers, or other remote services, i.e., components that have defined matching subscriptions.

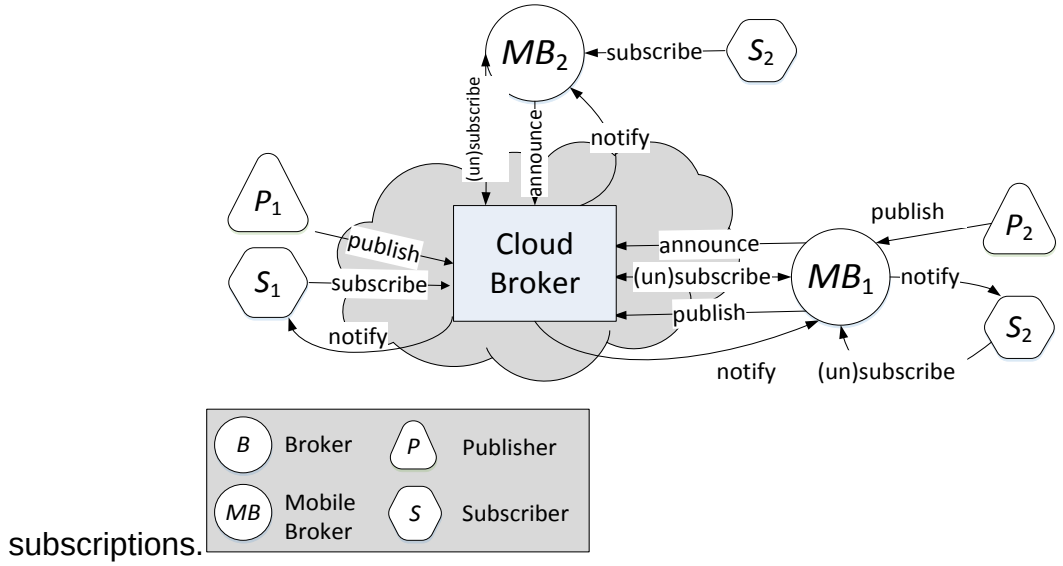


Figure 16. Publish/subscribe model and interaction.

The main novelty of the MoPS model compared to existing publish/subscribe solutions is the implementation of mobile brokers running on mobile devices such as smartphones and tablets. After their initial registration with the cloud broker, mobile brokers can *announce* the type of data for publishers which they represent. For example, P_2 in **Figure 16** is, e.g., a wearable gas sensor detecting levels of nitrogen dioxide (NO2) and ozone (O3). After MB_1 detects P_2 because they exchange signalling information over a Bluetooth connection, MB_1 can define the type of data items to transmit to its cloud broker B_2 in the future. MB_1 sends a message *announce*(NO2,O3,*x*,*y*), where *x*=45.81302 and *y*=15.97781 represent MB_1's current geographical latitude and longitude. The reason for creating the announce message is the following: We need to activate subscriptions from the cloud broker on MB_1, but only those that can potentially match future publications created by P_2. Obviously, as it is not desirable to activate all subscriptions from the cloud on a single mobile device, the announce message is compared to existing subscriptions on B_2. For example, B_2 identifies subscription s_i=[NO2 > 40μgm^{-3} AND 45.81<lat<45.82 AND 15.96<long<15.98] as a subscription potentially matching future publications of P_2. Thus, B_2 sends a message *subscribe* to activate s_i on MB_1. Further on, MB_1 publishes P_2's data items into the cloud, but only those that match s_i.

Selective and flexible data acquisition. By reusing the inherent features of the two-tier publish/subscribe model, we provide a flexible mechanism to control the sensing density over a predefined area covered by traces of mobile internet-connected objects (MIOs). It requires an orchestration of the sensing process with activation of adequate subscriptions on mobile brokers, as instructed by the back-end cloud system based on the integrated crowdsensed data. If we assume the density demand is predefined for an area as required by the application logic, MIOs residing in this area during a certain time interval can be instructed either

(1) to transmit the sensed data into the cloud as additional data samples are needed within this area for the particular time interval, or

(2) to restrain from such transmissions since the application has already acquired sufficient data samples for the area.

This is the main mechanism for frequency reduction of data transmissions from MIOs into the cloud which has the potential to greatly reduce energy consumption on MIOs. Consider the following example in **Figure 17**. It depicts movement traces for three MIOs m_1, m_2 and m_3 within a certain area, and denotes time intervals [t_{11}, t_{12}] and [t_{21}, t_{22}] within which the two MIOs perform data transmissions into the cloud (they are marked by the symbol $\times$), while m_3 does not perform any transmissions. MIOs perform transmissions at marked places because during the two time intervals subscriptions matching the data acquired by m_1 and m_2 are active on those MIOs. This does not impose any constraints on the sensing process as it largely depends on MIO interaction with sensors in its vicinity. For example, if the sensing process is pull-based, an MIO can invoke it periodically during the subscription activity periods. If sensors are configured to perform periodic sensing, mobile brokers residing on MIOs ignore the sensed data while it does not match any of the active subscriptions.

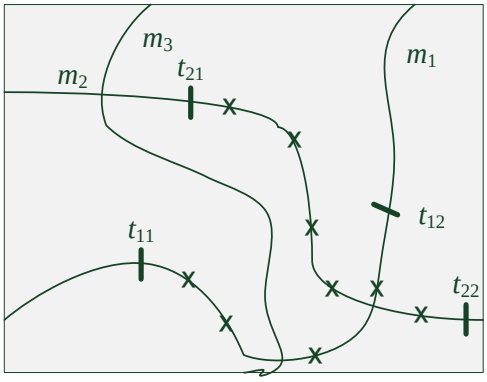


Figure 17. Movement traces and data transmissions.

Let us further explain who controls the activation of subscriptions on MIOs and how the sensing density is controlled. The back-end cloud system is notified when an MIO enters the depicted geographical area since MIOs are configured to announce their available data sources when entering the area. This requires periodic GPS positioning on MIOs which is potentially energy-greedy, but if other network-based techniques are available for determining MIO location, this process should not

represent a major obstacle for application adoption. In addition, mobile devices need to be aware of area boundaries that are important for the application logic. Since the back-end system is aware of the data samples already acquired over an observed area, it can decide whether to ship matching subscriptions to MIOs or not. In our example in **Figure 17**, the application logic has decided that there are sufficient measurements acquired for the depicted area from m_1 and m_2, and thus subscriptions were not forwarded to m_3.

Potential gains due to filtering of redundant data. To gain an insight into potential energy gains due to flexible data acquisition and filtering of redundant data, we have performed an analysis based on a real data set, the Mobile Data Challenge (MDC) data set collected during the Lausanne Data Collection Campaign from October 2009 until March 2011 [Laurila 2012]. The analysis is done such that we have randomly selected one day data traces for each of the 38 participant logs available in the MDC data set. Each such data trace represents an MIO movement over one day where we associate user locations with GSM cell identifiers. Two users are collocated if they reside with the same GSM cell during the same time interval, when they can potentially create redundant measurements. Our next assumption is that users carry wearable sensors with periodic readings generated once in a minute or once every five minutes. In addition, we assume that for our approach the required number of daily measurements within a cell equals 30.

Table 4. Energy gains due to flexible data acquisition.

	No. of cells	1 min	5 min	Our approach
1	822	40330	9344	2013
2	870	39686	9188	2011
3	942	41153	9884	2086
4	888	42068	9977	2071
5	777	39267	9250	1807

We have performed 10 iterations of the experiment with randomly selected daily traces from 38 different users and **Table 4** depicts our results for 5 selected iterations. The second column lists the number of different cell identifiers found in all traces. It varies from 777 to 942 different cells which tells us that there is not much overlap in user movement (at most 5 to 9 users are collocated in the same cell in all our experiments). The third and the fourth column list the number of daily data measurements if sensors generate periodic readings once per minute or 5 minutes, while the fifth column lists the number of such readings with our approach. One can see that our approach generates only around 5% to 6% sensor readings and data transmissions compared to 1/60 Hz measurements and 20% to 25% such readings compared to 5/60 Hz measurements. Thus, based on this preliminary analysis with unfavourable movement traces with low collocation probability, one can conclude that potential energy gains are significant.

Modelling the number of published messages while varying the sensing coverage. Here we investigate the number of messages generated by the MoPS approach w.r.t the area covered by mobile sensing. We assume that a geographic area can be divided into smaller location areas such as GSM cells and that a user mobility model is purely random. Subscribers or application logic can define interest in part of the geographic area and we want to compare the MoPS approach with a traditional publish/subscribe approach which contributes all acquired sensor data to the cloud broker. We define an analytical model to assess the number of transmitted publications comparing the CUPUS approach with the traditional approach to estimate potential gains in the number of transmitted messages from mobile phones to the cloud which directly influence energy consumption on mobile devices. By lowering the number of publications sent from a mobile device to the cloud, we can reduce the consumption of two key resources on a mobile device, the battery life and network bandwidth.

To calculate the savings in terms of the number of transmitted messages we use the following parameters, which can be estimated for real applications:

- n - the total number of publishers
- c - the total number of cells
- c_s - the number of cells with at least one subscription
- P_i - the number of publications generated by the i-th publisher
- c_i - the number of cells through which the i-th publisher has passed

In our analysis we are assuming that the number of cells c is constant and that cells do not overlap. Additionally, we assume that subscriptions are moving and are not fixed to specific cells, but such that a proportion c_s of cells with at least one subscription is constant during the observed experiment.

The savings can then be calculated as the percent decrease in the number of transmitted messages of our solution compared to the traditional one in which publishers are publishing all available data objects to the rest of the system, while with our solution only publications of interest to one or more users are published to the rest of the system:

$$\boldsymbol{S = \frac{M_{trad} - M_{MoPS}}{M_{trad}}}.$$

Since our solution generates additional control messages (i.e. announce messages and responses to announce), we need to add their number to the number of transmitted publications to calculate the total number of exchanged messages in our solution. The number of messages generated by a single user M_i is equal to the sum of his/her useful publications (i.e., publications that are delivered to subscribers) and control messages (announce messages with replies to them), and can be calculated as $\boldsymbol{M_i = P_i^u + A_i = P_i \cdot r_s + 2c_i}$, where $\boldsymbol{P_i^u}$ is the number of transmitted useful publications by the i-th publisher that is calculated as the product of the number P_i of

publications generated by the *i*-th publisher and $\boldsymbol{r_s = \frac{c_s}{c}}$ is the probability that a randomly selected cell has at least one subscription. The number of control messages A_i is defined as the number of cells through which a publisher is passing c_i, where in our case we can compare it with the number of GSM cell handovers that are made during publisher movement.

Finally, from the previous equations we get the following percent decrease in the number of transmitted messages for our solution:

$$\boldsymbol{S = \frac{M_{trad} - \sum_i^n M_i}{M_{trad}} = \frac{\sum_i^n P_i - \sum_i^n (P_i \cdot r_s + 2c_i)}{\sum_i^n P_i}}.$$

Hereafter, we analyse the number of transmitted messages in our approach when compared to the traditional approach. Table 5 shows the default parameter values used in the analysis. We analyse the influence of parameters r_s, P_i and c_i on the percent decrease *S*. For each analysis, we changed a single parameter, while all other parameters are fixed to default values in Table 5.

Table 5. **Default parameter values.**

Parameter	Symbol	Value
the number of cells	*c*	1500
the number of publishers	*n*	60
the percentage of cells with subscriptions	r_s	0.5
the average number of user publications	P_i	1000
the average number of cells through which a publisher has passed	c_i	100

Figure 18 shows how the percent decrease changes with increasing percentage of cells with subscriptions r_s. As we can see, the percent decrease falls linearly with r_s. By increasing the value of parameter r_s, we increase the number of cells for which there is interest from subscribers. As expected, the advantage of our approach drops when increasing r_s due to the drop in retained publications. Obviously, if all cells are covered by subscriptions, there is no value in data filtering on mobile phones as announce messages represent an overhead: Our approach drops to 0 when r_s reaches 0.8, but it can cause significant savings when r_s is in the range from 0 to 0.5.

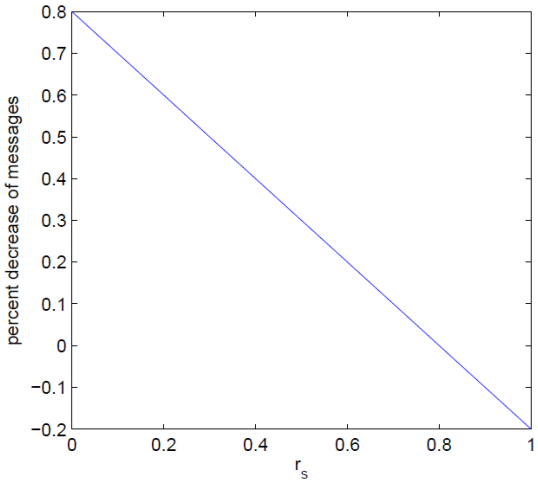


Figure 18. Percent decrease in the number of messages for different percents of cells with subscriptions.

In **Figure 19** we can see how the percent decrease changes when we increase the number of average publications per publisher P_i . As we can see in the figure, the percent decrease grows sublinearly with P_i. By changing the value of parameter P_i we model the frequency of publication production. Since our approach reduces the number of transmitted publications, by increasing P_i, the gain of our approach also grows under the assumption that r_s=0.5.

From the previous analysis it can be concluded that the data filtering approach on mobile devices can bring significant gains when the sensing area is below 50%. Further savings are possible by filtering redundant data within highly covered areas.

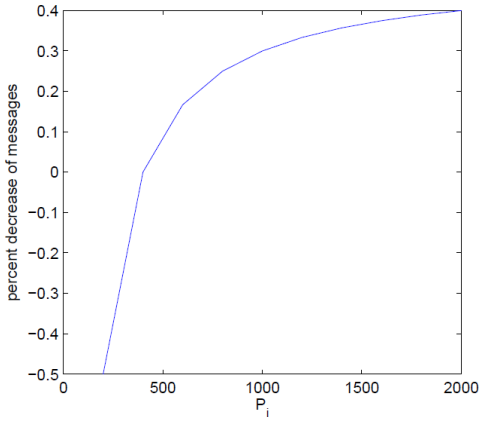


Figure 19. Percent decrease in the number of messages when increasing the number of average publications per publisher P_i.

Figure 20 shows how the percent decrease changes when increasing the number of cells through which an average publisher has to passed through. As we can see in the figure, the percent decrease drops linearly with c_i. By increasing the value of parameter c_i we model the speed and mobility of publishers. Since our approach generates additional announce messages when publishers are changing cells, obviously the advantage of our approach drops when increasing c_i.

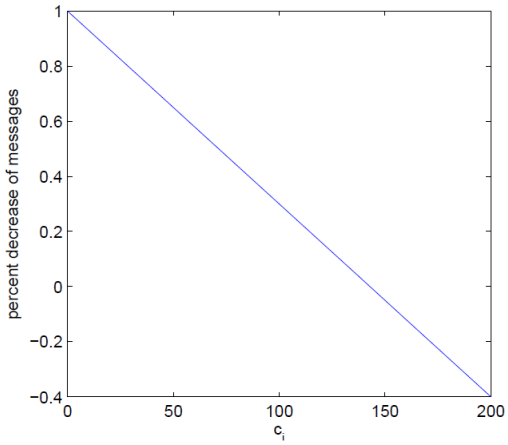


Figure 20. Percent decrease in the number of messages when increasing the number of cells through which a user passes through.

3.6 Request Type Optimization

Depending on the type of queries that users and applications dispatch to the OpenIoT infrastructure, access and processing of streaming data resources can be optimized in different ways. In particular, the dynamic nature of the data coming from sensors and ICOs, calls for efficient query processing mechanisms that go beyond traditional database management systems capabilities.

Moreover, given the potential diversity of sensor data sources, it is needed to represent and query the ICO data through a holistic model that reflects the application domain. Semantic Web and Linked Data technologies can answer to some of these requirements, as they provide well-defined models (in the form of ontologies) that can be interlinked, queried, and reasoned upon. Nevertheless, existing Linked Data platforms are designed for static data storage and not suited for streaming data processing,

To cope with dynamic streams of data coming from ICOs, in the OpenIoT project we use LSM [Le-Phuoc 2011], a middleware with functionalities to transparently cater for dynamic stream information [Nguyen 2012] and tailored to existing distributed sensor infrastructures: from Twitter streams down to resource-constrained sensing hardware. In the remainder of this section we highlight the optimization techniques present in LSM, especially for efficient query and stream data processing.

3.6.1 Efficient Query Processing

The linked stream data model brings several advantages in data correlation operations. The first advantage comes from the data acquisition and data distribution. The graph-based layout gives the data processing operators the global view of the whole dataset. Therefore, the query processor can filter the irrelevant data to a query much earlier than the log-file approach does. Traditionally, the monitoring data

recorded in separated log files are partitioned by in individual services, processes, etc., thus, cross-correlating the relevant data items among them needs to load all the data into a relational storage before carrying out the correlation.

The push-based and incremental processing model of linked stream processing engines provides much better performance than that of traditional relational database engine. Because a query over the log streams on relation database is performed in pull-based and one-shot fashion whereby any new snapshot of log stream needs the full computation. Thanks to the push-based and RDF triple data model, the log data can be pushed gradually per triples or a set of triples into the Data Correlation Engine. This helps to avoid the overload of matching schema and data loading when receiving a big monitored log file.

To meet the query processing demand of Data Correlation Engine, we evaluated a Continuous Query Evaluation over Linked Stream (CQELS) engine [Lephuoc 2011]. This engine can consume very high throughput from log streams and can have access to big persistent triple storages with millions of triples. The current version can deal with thousands of concurrent queries corresponding to service matching policies registered.

In OpenoT we aim at using a declarative language for defining stream processing functionalities by using query-based data acquisition operator is used to collect or receive data from data sources or gateways and can be pull-based or push-based. By using SPARQL/CQELS the data transformation and alignment can be done to produce a normalized RDF output format, thus a streaming operator streams the outputs of the final operator of a workflow to the consuming stream data applications. SPARQL/CQELS provides the engine for processing Linked data stream and Linked data. It contains a definition of the language specification and the engine for processing the input data.

The LSM architecture functionality is illustrated in Figure 21. It is divided in layers that together cover the entire process, from data acquisition, to Linked Data, publishing and access, until storage and applications by means of stream processing and correlated data.

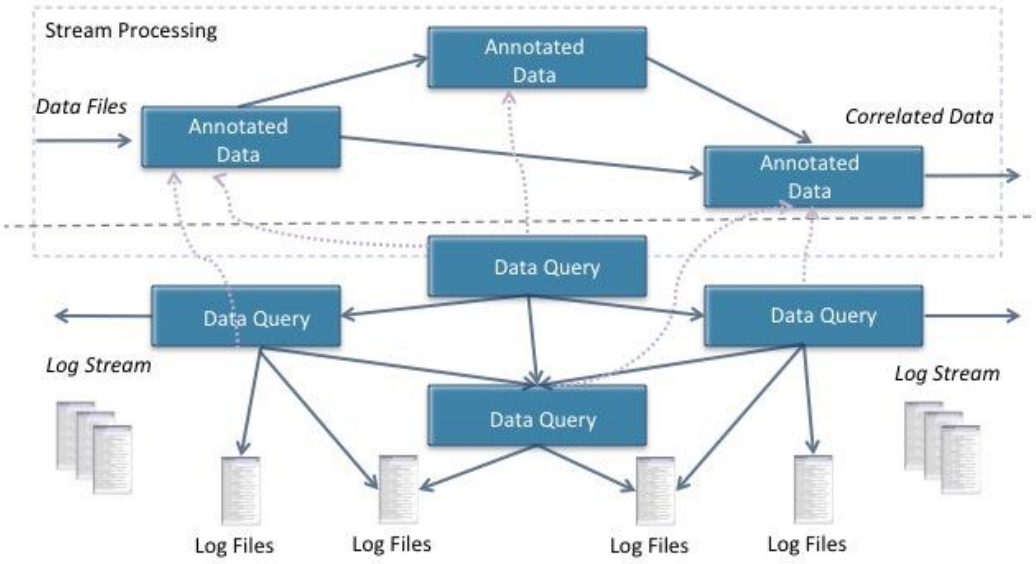


Figure 21. Linked Data Functionality by means of Linked Data in LSM.

3.6.2 Efficient Stream Data Processing

In OpenIoT we are witnessing more and more senor data services that are based on cloud computing models, which can typically lead to unprecedented economies of scale. These cloud computing infrastructures offer a pay-as-you-go model, as well as standard software stack for various applications.

While OpenIoT takes into account existing tools and techniques for the virtualization of computing resources, it also considers the possibility and the extent to which ICOs can be virtualized, despite limitations imposed by their geographical locations, administrative ownership and functional capabilities. OpenIoT indeed advocates the creation of virtual sensors through the X-GSN middleware, which can encapsulate internet-connected objects. On a higher level, the users of the OpenIoT cloud are able to develop applications that leverage information from multiple sensors, actuators and other devices. This abstracts users from specific ICOs, as they provide their data requirements through high-level queries (e.g. SPARQL) in terms of well-defined ontological models (e.g. SSN ontology). The Linked Sensor Middleware (LSM) is the OpenIoT component that is in charge of handling these queries. This is a first of a kind extension of existing cloud computing infrastructures: using algorithms and strategies developed in OpenIoT, end-users are able to configure, deploy and use IoT based services.

The use of near-real time stream data is a key enabler and driver in such diverse application domains as smart cities, home automation, ambient assisted living, or recommender systems. As on the Web, access to and integration of information from large numbers of heterogeneous sources under diverse ownership and control is a resource-intensive and cumbersome task without proper support.

In OpenIoT we have analysed LSM and others, e.g., SPARQL extensions to query RDF streams: C-SPARQL, EP-SPARQL, CQELS, SPARQLStream, or middleware that has been built for streaming data processing, e.g. SPITFIRE, GSN, etc. Still, widespread access to real streams does not exist at the same level as for Web resources. LSM, the Linked Stream Middleware[6], addresses this problem, providing access to more than 100,000 stream sources via a RESTful interface and a SPARQL/CQELS endpoint. However, to the best of our knowledge, no general-purpose infrastructure to support existing lower access thresholds for users and developers has been developed.

In OpenIoT, the usage of LSM enables efficient query processing over both static data (e.g. sensor metadata) and also streaming data (e.g. observations). LSM transforms the data from virtual sensors into Linked Data stored in RDF. A SPARQL query is a so-called one-shot query, and such queries typically refer to queries about sensor metadata and historical sensor readings. The SPARQL endpoint of LSM provides the interface to issue these types of queries. The currently deployed RDF triple store by LSM, OpenLink Virtuoso, provides a Linked Data query processor that supports the SPARQL 1.1 standard.

SPARQL queries are executed once over the entire collection and discarded after the results are produced, but queries over Linked Stream Data are continuous (registered in the system, and continuously executed as new data arrives). For processing continuous queries over Linked Stream Data, the LSM provides the CQELS engine [Le-Phuoc 2011]. The query processing in CQELS is done in a push-based fashion, i.e., data entering the query engine triggers the processing. The continuous queries are expressed in the CQELS language, which is an extension of the SPARQL 1.1 standard.

3.7 Energy Efficiency and Bandwidth Optimization

ICOs are often used in remote monitoring and control applications, where software running on general-purpose computers “pull” information from remote sensors and “push” actuations into the network. The ICO themselves form a multi-hop network communicating with one or more access points that interface between application software and the ICO network. Therefore, two resources that are scarce are the ICO are energy and link bandwidth.

The energy efficiency in ICO becomes an issue because each node in the network is equipped with a battery, but it is sometimes quite difficult to change or recharge batteries. Therefore, the crucial question is on how to prolong the autonomous ICO lifetime as much as possible. Hence, maximizing the lifetime of an ICO network through minimizing the energy consumption is an important challenge since sensors

6 Linked Stream middleware (LSM), lsm.deri.ie.

cannot be easily replaced or recharged due to their ad-hoc deployment in distant locations and hazardous environments.

The bandwidth optimization in ICO becomes an issue because the network can be concurrently used for different applications (measurements). As an example consider a network monitoring temperature, light and noise in company offices and production halls. In such applications, the relative importance of ICO data streams often depends on the type and values of the data being sensed, and on how data from different streams is correlated with each other. For example, if the goal of temperature monitoring application is to actuate heating or cooling then it would make sense to allocate more network bandwidth for data streams coming from occupied rooms compared to empty rooms. As a more extreme example, if sensors in an area detect abnormally high temperature, it may signify a disastrous event like a fire, in which case it would be prudent to allocate almost all of the bandwidth to those streams. Thus, as such ICO shared networks grow in size, they require a bandwidth allocation method, by which the nodes can decide how to allocate network bandwidth to the streams. The allocation method has to handle traffic that exhibits a high degree of spatial correlation, when a group of nodes in close proximity all detect an event of interest. Thus, it has to be able to change bandwidth allocations in the network depending on observed phenomena.

3.7.1 Energy and Bandwidth Consumption on MIOs

In environments with MIOs and smartphones as described in Section 3.5.2, the process of pushing messages from the cloud to smartphones can incur large energy costs. A recent study shows that periodic transfers in mobile application which account for only 1.7% of the overall traffic volume contribute to 30% of the total handset radio energy consumption [Qian2012]. Thus here we investigate potential solutions for sending sensor readings to user smartphones and evaluate experimentally the incurred energy and bandwidth consumption.

Hereafter we briefly report three potential solutions that have been implemented and tested to enable delivery of notify messages in the MoPS system: 1) persistent TCP connection, 2) connection-less communication over HTTP where a REST web service is running on a mobile phone, and 3) REST web service with Google Cloud Messaging.

Persistent TCP connections are the simplest mechanism to implement, but can cause significant overhead as keep-alive messages are needed to maintain an active connection which prevents the processor from going into a sleep mode.

Connectionless REST-based communication between a mobile device and the cloud is an alternative to permanent TCP connections. Both the mobile device and server need to run a REST service: Whenever they want to communicate, they send HTTP messages to the REST service entry-point. In comparison to TCP connections, this mechanism is one step closer to push-based communication where situations of

temporary connection losses and failed handover do not affect the communication mechanism.

This mechanism does not allow a power save mode, but reduces the generated traffic over wireless interfaces and reduces the number of open connections. REST-based mechanism allows a mobile service to use a single entry point for all incoming messages, regardless of the sender, while the previous approach uses separate TCP connections for each sender.

For a fully implemented push-based message delivery mechanism in mobile environments we have used the Google Cloud Messaging (GCM) service. GCM is a service provided by Google running as an intermediary between application servers (cloud-based brokers in case of our prototype) and mobile devices running the Android OS. GCM uses a simple format for messages limited to 4 KB. A mobile service does not need to be in active state to receive such notifications: The Android OS will start or wake up the service upon a received message. The mechanism does not create, handle or destroy any additional connections, which makes it a true push-based communication mechanism without additional overhead. Since the support for the GCM service is an integral part of the Android operating system, GCM only requires that a radio interface is online, and allows the processing unit to go to power save mode. The GCM mechanism is used by various Google applications on mobile devices and reuses the same connection for the delivery of all messages, thus reducing the communication overhead to a minimum. The main drawback is limited availability (only for AndroidOS) and dependency on a third party solution.

Experimental evaluation. In our evaluation scenario the previously listed communication paradigms are tested such that we send sequentially notify messages to smartphones, and measure battery power consumption and generated network traffic at the wireless interface of a mobile device. Measurements are performed on a Samsung Galaxy S4 Android phone. The power consumption of a mobile device is measured with the PowerTutor application, and network traffic monitoring is performed with the TrafficMonitor application. All other services, which could potentially use the GCM for its purposes (e.g. Gmail application, other Google's services) were stopped during the evaluation phase.

At the beginning of the evaluation scenario, a mobile service registers itself at the MoPS server, such that the server is aware of a mobile service and of the mobile device address. After the registration, the mobile service no longer sends any data, because evaluation scenario is focused on the resource consumption for various receiving paradigms. The server generates a random data set of notification items, and sends them to the registered mobile device. A data item consists of five numbers, where each number is written with double precision, so a data item has the size of 40 bytes. Small data items were used because we wanted to analyse the receiving paradigm overhead. Larger amount of data would mask the overhead resource consumption, because most of the resources would be spent to transfer the data.

In first energy consumption test, the server has sent 1000 data items with an average interval of 1 second between two consecutive notify operations on a Wi-Fi interface. In this case the phone did not enter a power-save mode. The measured values of energy consumption are shown in Table 6. The power consumption is measured in mili Watts, the duration of each paradigm runtime necessary for receiving the entire data sets in expressed seconds, and the consumed energy expressed in Joules. All three paradigms need approximately the same time for receiving 1000 data items. The GCM paradigm is the most favourable technique for sending notifications as it consumes almost 50% of the energy required for TCP-based solution, while REST has an overhead of almost 20% compared to TCP (Figure 22).

Table 6. Energy consumption on a Wi-Fi interface for receiving 1000 data items

Communication paradigm	Power consumption [mW]	Runtime [s]	Energy consumption [J]
TCP	103.6	1034	107.12
REST	118.57	1053	124.85
GCM	53.27	1041	55.46

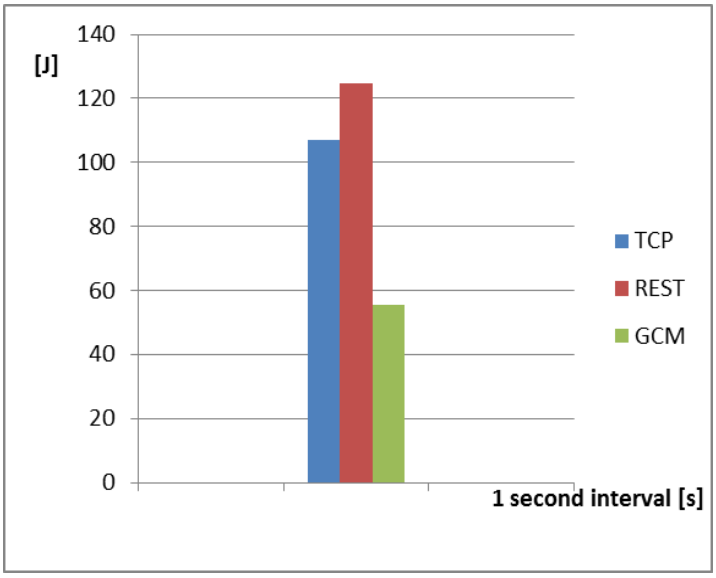


Figure 22. Energy consumption on a Wi-Fi interface for receiving 1000 data items.

The second energy consumption test is done by sending 100 data items, with an average interval of 10 seconds between each notify operation. In this case the smartphone did enter a power-save mode between each receive operation. Results of the second test are shown in Table 7. As one can notice the GCM paradigm once again has the best performance, but in this test other two paradigms have much better results than in the first test scenario (Figure 23). In general, the GCM service shows the best results regarding energy consumption because no additional network connections are needed while the processor can go to the power save mode.

Table 7. Energy consumption on a Wi-Fi interface for receiving 100 data items

Communication paradigm	Power consumption [mW]	Runtime [s]	Energy consumption [J]
TCP	67.73	1031	69.83
REST	77.67	1049	81.47
GCM	45.73	1017	46.51

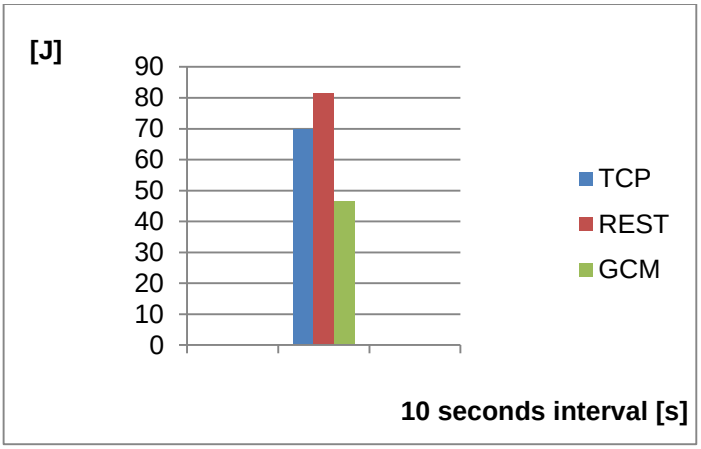


Figure 23. Energy consumption on a Wi-Fi interface for receiving 100 data items.

Parallel with energy consumption tests, we also measured the bandwidth consumption with TrafficMonitor application on the phone Wi-Fi interface. In the first bandwidth consumption test, the server sent 1000 data items, with a 1 second interval between each transmission. The TCP-based solution generates the least amount of traffic, and our REST-based solution generates the largest amount of traffic (approximately 5 times larger than pure TCP) as expected since entities communicate using the HTTP protocol. The TCP paradigm provides the best results because it introduces the least overhead. In addition to our data set, the data transferred through the GCM connection also contains the identifier of the intended recipient, while the REST solution is built on top of HTTP (Figure 24). The REST paradigm generates much more traffic than the other two, especially for upload (i.e. upload) as shown in Table 8.

Table 8**.** Bandwidth consumption on a Wi-Fi interface for receiving 1000 data items.

Communication paradigm	Total bandwidth [kB]	Download [kB]	Upload [kB]
TCP	264.13	256.51	7.62
REST	1402.88	1293.29	109.59
GCM	973.81	958.73	15.08

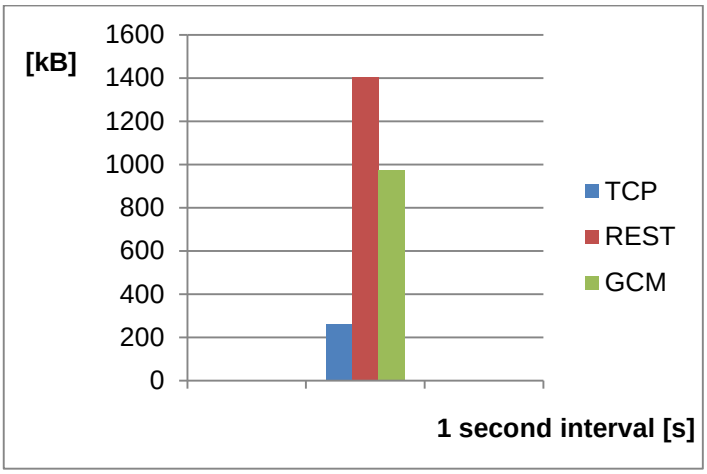


Figure 24. Bandwidth consumption on a Wi-Fi interface for receiving 1000 data items.

The second bandwidth consumption test was done by sending 100 data items, with an average interval of 10 seconds between each data transmission. The results of the second test are shown in Table 9. The TCP-based solution once again generated the least amount of traffic, and REST generated the largest amount of traffic (Figure 25).

Table 9**.** Bandwidth consumption on a Wi-Fi interface for receiving 100 data items.

Communication paradigm	Total bandwidth [kB]	Download [kB]	Upload [kB]
TCP	52.82	45.05	7.77
REST	986.77	961.29	25.48
GCM	203.24	189.67	13.57

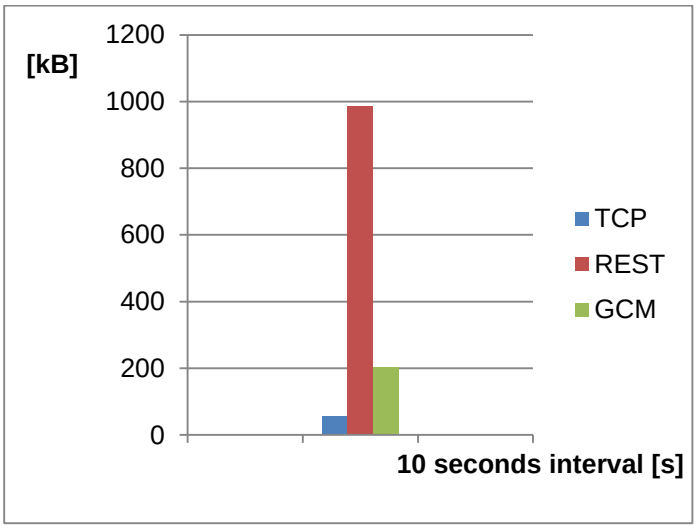


Figure 25. Bandwidth consumption on a Wi-Fi interface for receiving 100 data items.

After testing all three paradigms for battery consumption, power consumption and generated network traffic we can conclude that the TCP solution generates the least amount of traffic on a Wi-Fi network interface and the GCM paradigm consumes the least energy, compared to the other two paradigms, especially when the time interval between two consecutive data transmission is large enough such that the processor can go to the power save mode. The REST paradigm consumes the most energy and generates the biggest traffic on a Wi-Fi interface, so we can conclude that the REST paradigm is ineffective in terms of energy consumption and generated network traffic.

3.7.2 Bandwidth Optimization through Indirect Sensor Control

The most commonly used resource and therefore the most significant source of bandwidth consumption on the OpenIOT platform, is expected to be the data streamed from the sensors to the users. This section describes an optimization strategy that addresses this issue and has been developed in the context of **T5.2 Resource Sharing and Management.**

The module responsible for streaming from sensors to the LSM is X-GSN. In the current implementation of X-GSN, once a sensor is activated it streams data continuously whether the data is actually needed from a service or not. This results in a misuse of available bandwidth. In order to address this issue, a module that applies Indirect Dynamic Sensor functionality has been implemented on top of the X-GSN module.

As the name of the module implies, the control (activation/deactivation) of a sensor is not to be controlled directly from the user. Rather, a user **announces** the creation of a service which makes use of a group of sensors, to the Request Definition module. The request is forwarded to the Scheduler which in turn creates a SPARQL triplet of a “serviceID HAS sensorID” format on the LSM, which is represented by the sensorServiceRelation entity, stating which sensors a particular service is intending to use.

At the same time, a periodic timed task is running on the X-GSN module, which is responsible for direct sensor management, querying the particular triple on the LSM repository, in order to determine which sensors are being currently announced/ requested by users. The task compares the query results from the triplets, with the list of virtual-sensors that are currently active on the X-GSN module. Then X-GSN activates virtual sensors that have been found by the query but are not active on the module and deactivate the virtual sensors that are active on the module but have not been found in the query.

This process is illustrated in the sequence diagram in Figure 26 and flow chart diagram in Figure 27.

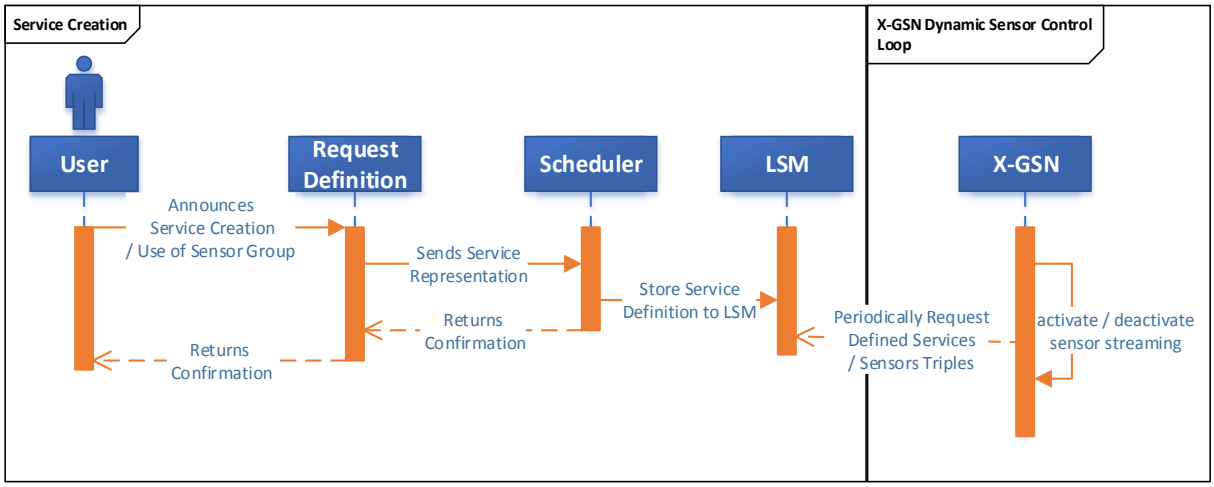


Figure 26. Indirect Dynamic Sensor Control Sequence Diagram.

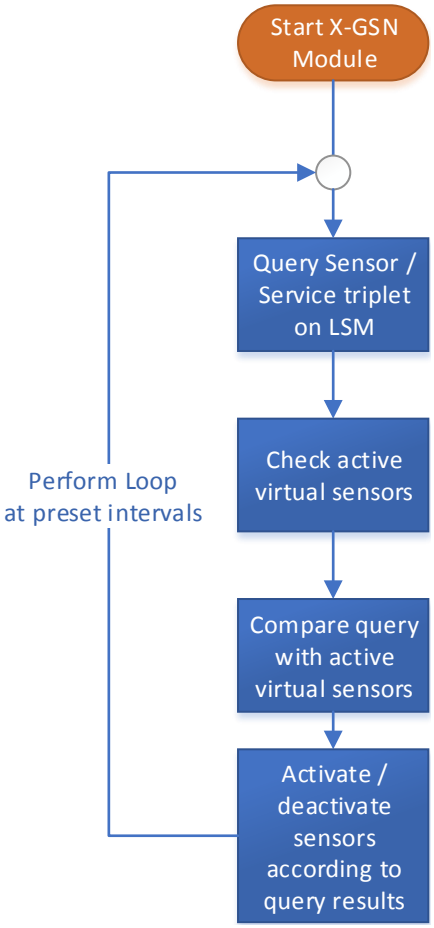


Figure 27. Indirect Dynamic Sensor Control Flow Chart.

The above process is expected to result in significant bandwidth conservation, since sensors streams data only when they are actually used, as opposed to them streaming on a constant basis.

3.7.2.1 Sensor Use Identification

To implement the Dynamic Sensor Control functionality, the X-GSN module needed support a basic API that activates/deactivates virtual-sensors programmatically. The original GSN framework does not support such an API, and sensor activation / deactivation is performed by copying or removing accordingly the virtual sensor .xml and metadata files, from the **virtual-sensors** directory within the GSN source folder.

In order not to temper with the existing code, an independent module for has been implemented that performs the Dynamic Sensor Control functionality.

The module's functionality can be described briefly as follows:

- By querying the LSM, an ArrayList<String> of active sensor names are obtained
- Then the module scans the **virtual-sensors** directory for all .xml files constructs a HashMap<String, File>, that maps **sensor id Strings** which are obtained from the name property in the <sensor>.xml.metadata file, with **File Objects** that correspond the virtual-sensor names
- The module scans the **available virtual sensors** from the **LSM** folder in the X-GSN module, again mapping **sensor id's** with **corresponding files.**
- The ArrayList<String> obtained from the query on the LSM is compared with the first HashMap<String, File>. Any sensor names located in the HashMap but not in the SPARQL query ArrayList are deactivated. This is performed by deleting the corresponding .xml and .xml.metadata files from the directory
- Finally the ArrayList<String> obtained from the SPARQL query is compared with the **available sensors** and activates them by copying the corresponding files to the **virtual-sensors** directory
- This functionality is embedded in a TimerTask class (the DynamicControlTask class) that is executed in predefined intervals

4 PROTOTYPE IMPLEMENTATIONS

In the previous sections, we have described the techniques, algorithms and principles that we have conceived, designed and proposed for the self-management and optimization framework of OpenIoT. We have supported our design choices and algorithms with experimentation and evaluation over proof-of concept prototypes, where applicable.

In this section we provide details about the prototypes that actually implement the techniques and algorithms presented previously, within the components of the OpenIoT architecture. This includes functional specifications and a summary of the design decisions and technical details needed to adapt, modify or configure the OpenIoT modules concerned. Specifically, we include:

- The Utility-based optimization implementation. It implements the cost and valuation functions introduced in Section 3.4 and the acquisition algorithms in Section 3.5.
- The Dynamic Sensor Control module, which implements the control of X-GSN virtual sensors for Bandwidth optimization, as explained in Section 3.7.2. The implementation details of the data acquisition and filtering mechanism for mobile devices as specified in Section 3.5.2 are available in deliverable D3.4.1.
- Caching Scenarios prototype. It describes the simulator that calculates cache costs associated with accessing a cloud data-store, in combination with a local caching solution, following Section 3.2.3.5.
- Cloud optimization implementation. It specifies the implementation of the integration of LSM and X-GSN including the cloud optimization based on view-models using memory indexes and Map Reduce-based query processing.

We have elaborated **Table 10**, which shows how the different techniques explained in Section 3 relate to the implementation descriptions in this section. In the case of the mobile publish subscribe system (MoPS) that addresses efficient data collection and bandwidth optimization for mobile devices, the implementation is further described in Deliverable 4.5.1. For LSM, full implementation details have already been provided in Deliverable 3.3.1.

Table 10. Prototypes and module implementations vs. OpenIoT Management and optimization Techniques.

Prototypes and module implementations vs. OpenIoT Management and optimization Techniques	Utility-based optimization	Dynamic Sensor Control	Caching Scenarios Simultator	Cloud Optimization	Mobile Publish Subscribe	LSM CQELS
Efficient Scheduling		X	X			
Cloud Optimization				X		
Utility-based Optimization	X					
Efficient Sensor Data Collection	X				X	
Request Types Optimization						X
Energy Efficiency and Bandwidth Optimization		X			X	

4.1 Utility Based Optimization

In this section we present the functional specification of the utility-based optimization in OpenIoT described in Section 3.4**.**

4.1.1 Functional Specification

In OpenIoT utility-based data collection and query processing is performed in a subcomponent of SD&UM. We refer to this subcomponent as Utility-based Optimizer (UBO). **Figure 28** depicts the high level functional architecture of utility-based optimization in OpenIoT. UBO periodically retrieves the available queries from the OpenIoT cloud database, the metadata of sensors in the regions requested by these queries and the trust score of these sensors. Trust scores of sensors are calculated by the trust assessment component described in Deliverable 5.2.1. Given these information, UBO performs utility-based sensor selection to identify the sensors that are used to answer the queries. After selection of sensors it might be necessary to rewrite the SPARQL queries in order for them to read from selected sensors. These rewritten queries are denoted by **Queries*** in **Figure 28**. The frequency of running UBO optimizations is read from a configuration file. However, this frequency can be updated based on the scheduling information of the requests that arrive to SD&UM.

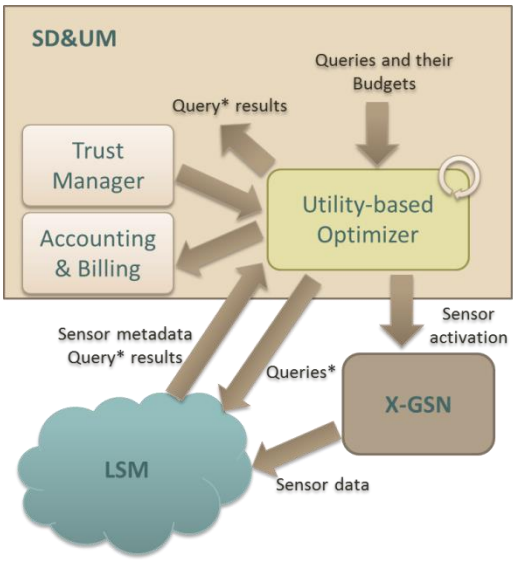


Figure 28. High level functional architecture of utility-based optimization.

Figure 29 shows the functionality of UBO in more details. In the following we describe the steps that are taken in each execution round of UBO.

1. All the OpenIoT Service Model Object (OSMO) objects available from the Service Delivery and Utility Manager (SD&UM) are retrieved.
2. The SPARQL queries in OSMO objects are parsed and the required point queries and spatial aggregate queries are created for each OSMO object. For example, a query asking for a reading from sensors, is translated to a point query asking for a sensor reading at the location of sensor *s*. A query asking for the average value of sensor readings from a set of sensors *S*, is translated into a spatial aggregate query asking for the average sensor reading in a rectangular area that contains all the sensors in *S*.
3. The metadata of sensors which fall into *the enclosing queried region* are retrieved from the Directory Service. The enclosing queried region is the smallest region that contains all the regions defined in spatial aggregate queries and the regions defined around the locations queried by point queries.
4. Utility-based multi-query optimization algorithm explained in Section 3.4 is run against the extracted queries and the metadata for available sensors. The result of this step is a set of sensor IDs.
5. New SPARQL queries for each OSMO object are created from the original SPARQL queries based on the selected sensors.
6. If a selected sensor is not activated, a message is sent to X-GSN to activate the sensor and push its data to LSM.
7. The new SPARQL queries are executed by forwarding them to the Directory Service SPARQL interface.
8. The query execution results are forwarded to Request Presentation.
9. The cost of sensor readings is split among the queries and the corresponding accounting information is sent to the Accounting & Billing module.

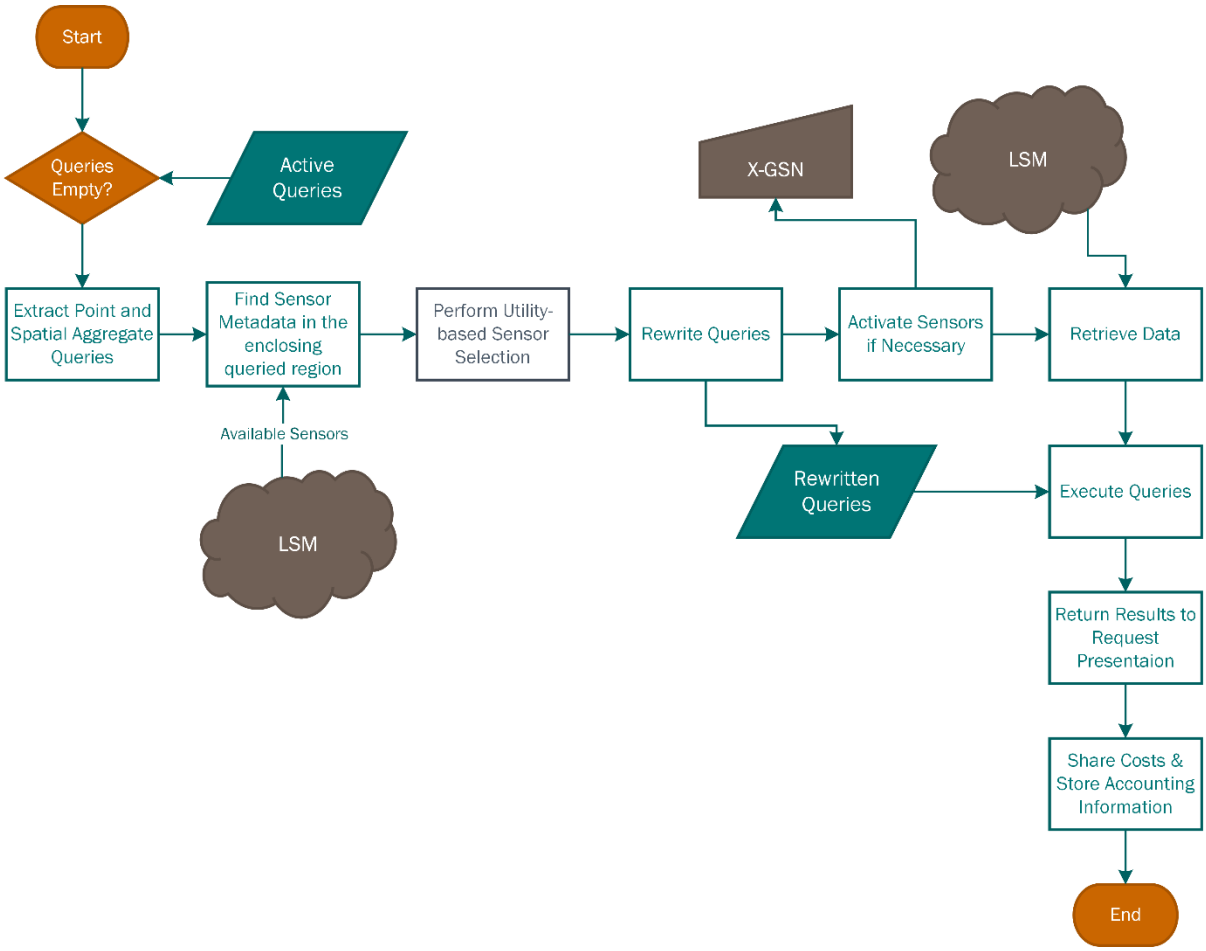


Figure 29. Utility-based query execution.

4.1.2 Required Information about Sensors and Queries

The UBO assigns a *cost* to each sensor. The cost value can be specified in the sensor metadata, which is accessible to the UBO through **SensorType** objects. If the cost information is not available in the metadata, we assume that at least information about energy consumption of sensors is available in their metadata. Based on this information, the UBO can assign a reasonable pre-defined cost to the corresponding sensors.

In OpenIoT heterogeneous stationary or mobile devices are available. The device owners, especially mobile device owners, can be concerned about possible leakage of their privacy by providing data about themselves or about their surroundings. In order to minimize privacy threats or to manage the level of privacy leakage, privacy protection mechanisms are employed on the devices. OpenIoT cannot impose any specific privacy protection mechanisms on the sensing devices. However, the privacy requirements of the device owners must be considered while utility-based data collection and query processing is performed. In order to achieve this important requirement, we assume that the cost reported by the sensing device already

includes the cost of possible privacy loss as mentioned in Section 3.4.23.4.2. When the cost information is not provided by the sensing devices, the assumption is that the device owners are not concerned about their potential privacy loss; hence in assigning the default cost only the energy consumption of the device is taken into account.

When a user defines a new service through the Request Definition module, he/she can assign a maximum budget for obtaining the query results. This maximum budget along with a suitable predefined query-type specific quality assessment function determines how much the data collected for answering the defined query is worth. Examples of these functions can be found in Section 3.4.3. The budget information is stored in each service's OSMO object. If the query budget is not specified, the average sensor reading cost is used as the budget of the query. If the query is scheduled to run continuously, in each query execution round, this average cost is considered as the budget.

4.2 Dynamic Sensor Control Module

4.2.1 Main Released Functionalities and Services

In the context of **T5.2 Resource Sharing & Management,** a Dynamic Sensor Control module has been developed in order to extend the X-GSN module's functionality. The functionality of this module as described in Section 3.7.2 periodically queries the LSM for the active sensors and activates/deactivates the relevant virtual sensors on the X-GSN module. In order to implement the module it was necessary to provide and extension to the API of X-GSN that would perform these queries that identify the currently active sensors. This module is explained in further detail in 3.7.2.1.

4.2.2 Download, Deploy and Run

The current module is embedded in the X-GSN module, therefore the process to download, install and run this module is already handled when performing the same process with X-GSN. Refer to deliverable D4.3.1 for specific details.

4.2.2.1 Source Code Analysis

This section describes the architecture of the Dynamic Sensor Control code. Figure 30 represents the UML class diagram that facilitates the Sensor Use Identification functionality (identify the X-GSN sensors that are used in the queries, and activate/deactivate virtual-sensors accordingly, as in Section 3.7.2.1)**:**

In Figure 30 the UML depicts the classes comprised in the Dynamic Sensor Control Module. The Parser interface and the SensorParser implementation class are in charge of parsing the RDF metadata in LSM, and extract the identifiers of the virtual sensors. The DynamicControlTask class encapsulates the operations of activation and de-activation of virtual sensors. It has a SparqlClient attached to be able to pose SPARQL queries, get results (which can be later parsed) and get the virtual sensor identifiers to activate or de-activate.

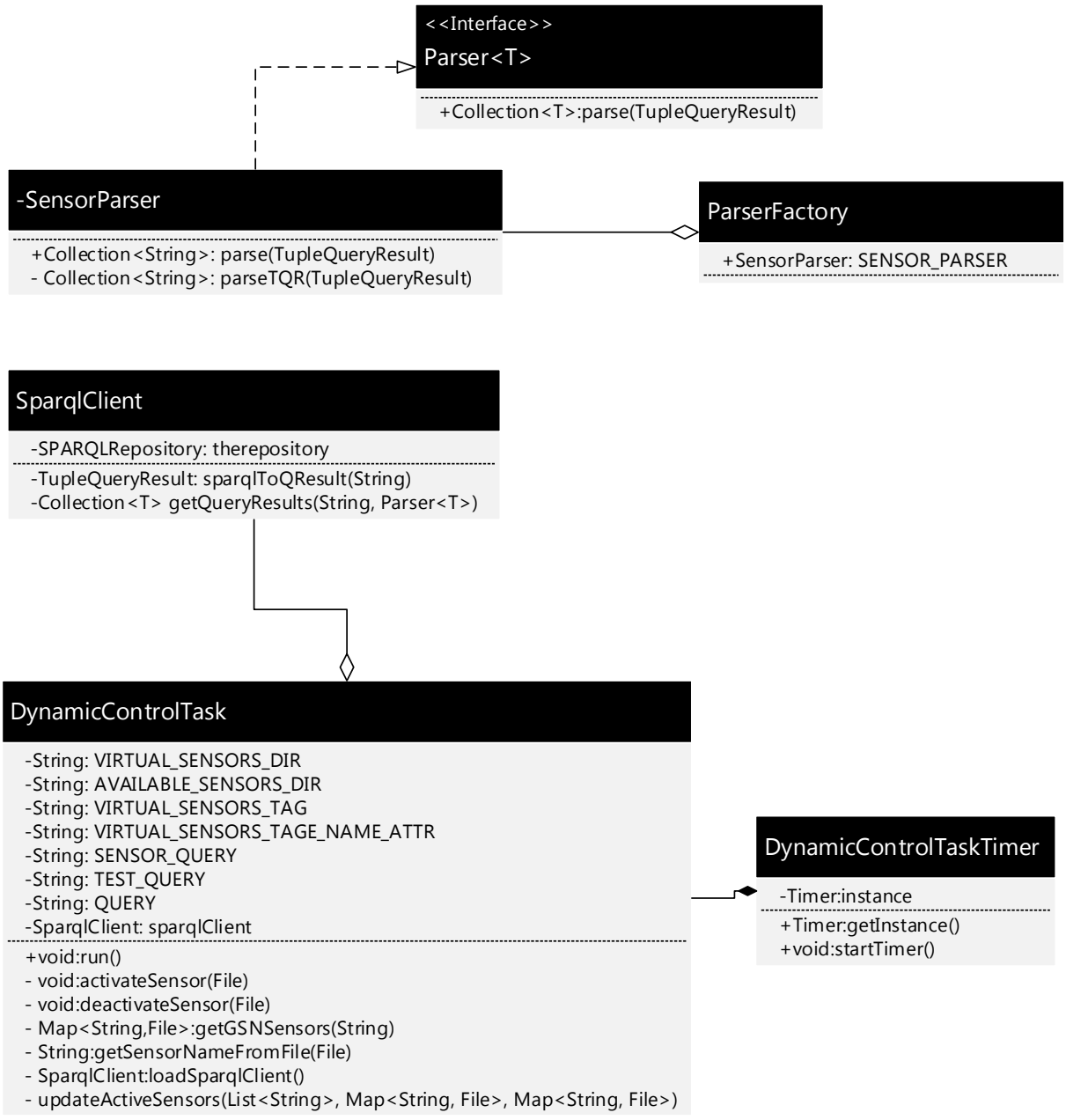


Figure 30. Dynamic Sensor Control UML Diagram

Also, the following tables analyse various class methods of the module's components.

<<Interface>> Parser<T>

Service Name	Input	Output	Info
parse	TupleQueryResult	Collection<T>	This method should be implemented by any class implementing the Parser<T> Interface

The Parser<T> interface is part of the Strategy Design Pattern that is used to manage various Parser subclasses that are used from the SparqlClient. Each class implementing this interface returns a Collection of the type defined at runtime.

ParserFactory

Service Name	Input	Output	Info
SENSOR_PARSER	void	SensorParser	Sensor parser implements the Parser<String> interface meaning that the parse method returns a Collection of Strings.

ParserFactory implements a Static Factory Design pattern that creates objects that implement the Parser<T> interface.

private inner class SensorParser

Service Name	Input	Output	Info
parse	TupleQueryResult	Collection<String>	Simple calls the parseTQR method
parseTQR	TupleQueryResult	Collection<String>	Implementation of the actual parsing functionality

SensorParser implements a Parser<String> interface and is the only concrete implementation at the moment. Sensor parser parses TupleQueryResults and returns Strings that represent a sensor ID. The parser is executed over the SPARQL query results from LSM.

DynamicControlTask (Singleton)

Service Name	Input	Output	Info
run	void	void	This is the main method of the class that is executed once a timer starts it
activateSensor	File	void	Copies active sensors from the LSM directory to the

			virtual-sensors directory
deactivateSensor	File	void	Deletes inactive sensors from the virtual-sensors directory
getGSNSensors	String	Map<String, File>	Retrieves GSN sensors from the specified path
loadSparqlClient	void	SparqlClient	Loads the sparql client
getInstance	void		Retrieves the singleton instance

DynamicControlTask is the class providing the main dynamic sensor control functionality. Since it is desirable that only a single task of that type is running at a given time, it is implemented as a singleton. Other than that its' main responsibilities are using the SparqlClient class to query the LSM for active sensors and activate/deactivate the corresponding virtual sensors. The activateSensor and deactivateSensor method implements the copy and activation of sensors, if they are announced in LSM. Conversely, the deactivate Sensor deletes inactive sensors in X-GSN, thus optimizing the use of resources in the system.

DynamicControlTaskTimer (Singleton)

Service Name	Input	Output	Info
getInstance	void	Timer	Retrieves the singleton instance
startTimer	void	void	Initiates the timer

The DynamicControlTaskTimer class simply starts/cancels the timed schedule for the DynamicControlTask class. Similarly to the DynamicControlTask, we only want a single Timer to be active. Therefore, this class is also implemented as a singleton.

SparqlClient

Service Name	Input	Output	Info
getQueryResults	String, Parser<T>	Collection<T>	This method receives a string query and a Parser<T> with which the query results are parsed. It retrieves the results from the LSM, inserts them into the parser and returns a Collection of objects specified by the parser algorithm that is selected.

sparqlToQResult	String	TupleQueryResult	This method receives a Sparql query in String form and returns a TupleQueryResult which can then be inserted into a parser

Finally, the SparqlClient class is responsible for establishing a connection with the LSM database, in order to perform queries and return results.

4.2.2.2 Configuration

Concerning the Dynamic Sensor Module itself, there is a limited set of functionalities that can be configured in the **conf/lsm_config.properties** file. The file contains the following lines that concern the specific module:

```
#DynamicControl
functionalGraph = http://lsm.deri.ie/OpenIoT/guest/functionaldata#
endPoint = http://lsm.deri.ie/sparql
virtualSensorsDir = virtual-sensors
availableSensorsDir = virtual-sensors/LSM

dynamicControl = true
#enter frequency of dynamic sensor control in minutes
dynamicControlPeriod = 5
```

The properties (**Table 11**) that can be configured are the following:

Property	Explanation
functionalGraph	The link to the RDF Graph that is to be queried
endPoint	An LSM endpoint that is used to establish the connection for the SparqlClient class
virtualSensorsDir	This is the folder where active virtual sensors (xml files) are expected to be found
availableSensorsDir	This is the folder where available virtual sensors (xml files) are expected to be found
dynamicControl	This property states if the Dynamic Control actives (True) or inactive (False). The default value is true
dynamicControlPeriod	This property states the time interval, which the LSM query for active sensors (in minutes).

Table 11. Dynamic Sensor Control Properties.

4.3 Caching Scenarios Simulation Prototype

In the context of **T5.2 Resource Sharing & Management,** a "Caching Scenarios Simulation Prototype" has been developed. The purpose of the prototype is to simulate the costs associated with accessing a cloud data-store, in combination with a local caching solution. It is expected, that the simulator assists in estimating the cost-efficiency of such a system, depending on the average request load and the caching solution that is implemented.

4.3.1 Main Released Functionalities and Services

The above prototype simulator has been developed as an MS Excel Workbook. It is separated into three distinct worksheets:

- Instructions
- User Input & Simulation Chart
- Chart Calculation Parameters

In further details the distinct workbooks have the following functionalities.

Instructions

In **Figure 31** the simulator introductory screen is displayed, which explains the functionality of the worksheet and how to use it. The particular worksheet is locked entirely and cannot be edited. Further detailed instructions, on using the prototype are included in the worksheet itself.

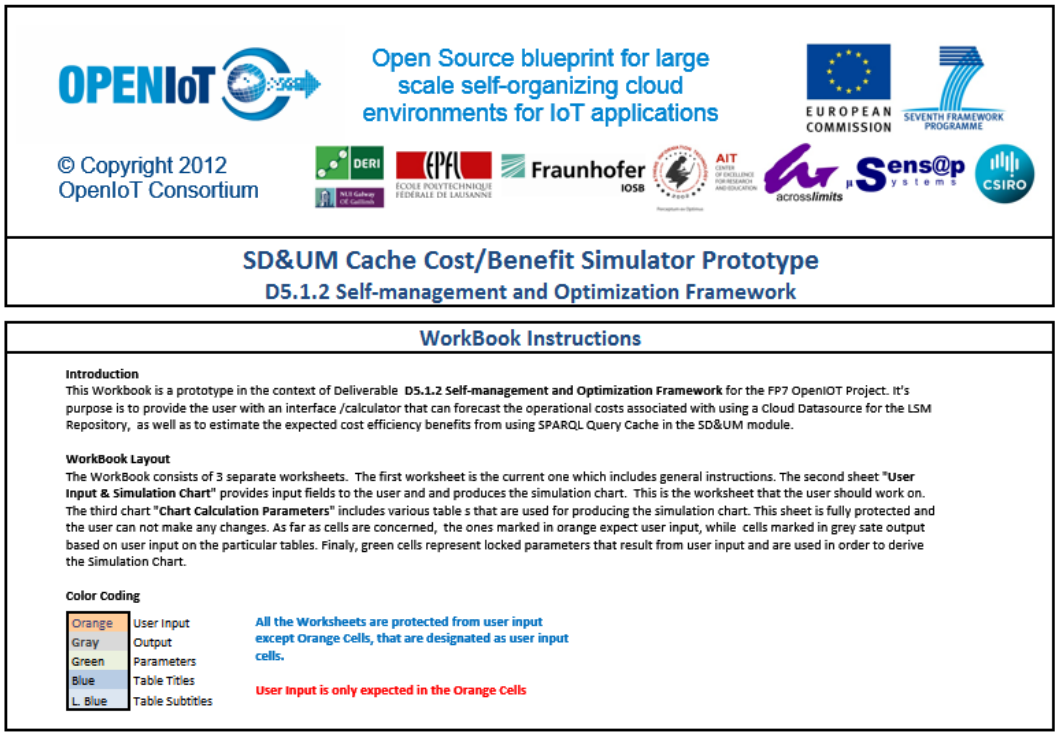


Figure 31. Caching Simulation Introductory Screen.

User Input & Simulation Chart

This worksheet is where the user can provide actual input and view the results on the produced bar chart. The following figure (**Figure 32**) displays the relevant input cells

Cost Per Request	
$ / 1000 requests	0,0050
EUR/USD Rate	1,35
€ / 1000 requests	0,00370

Calculation of €/hour cost for Cloud Datastore			
Estimated KRPH	600	1450	2000
€/hour	2,22	5,37	7,41

Caching Server Cost /Unit	
Server Disk Capacity (TB)	5
Unit Cost(€)	3500
Lifespan (years)	3
PV Discount Rate (%)	5%
Server Maintenance/year (€)	1500
Energy cost / year (€)	1000

Server Cost / Year (Present Value)	
Cache Size Required (TB)	20
Servers Required	4
Cost / Year (PV 5% rate)	22.635,00 €

Acronyms

PV = Present Value

TB = Terabytes

KRPH = Thousand Requests per Hour

Figure 32. Caching Simulator User Input.

In the particular figure the orange cells are the ones expecting input from the user, while the grey cells provide the related output. These inputs/results are used in combination with other hard coded parameters in the **"Chart Calculation Parameters"** worksheet in order to provide the resulting chart. Again this worksheet is locked for editing besides the orange cells.

Chart Calculation Parameters

This worksheet contains parameters that are used to create the chart in the **"User Input & Simulation Chart"** worksheet**.** Certain parameters are obtained from the above worksheet, others are hardcoded in order to facilitate various calculations.

An indicative screen of the particular worksheet is displayed in **Figure 33**.

Chart Calculation Parameters

The following tables are used to calculate the costs per hour/year incurred from the use of the Cloud Datastore. These costs are calculated for the assumed cache miss rates on the respective column on **Table 1.** Then the costs are calculated or the estimated number of requests in thousands on **Table 2.** Finally, **Table 3** is used top prepare the data for the chart in **the User Input & Simulation Chart Worksheet.**

Hours in year	8640

Table 1

Datastore Request Costs / Cache Miss Rate				
		600,00	1450	2000
a distribution	Cache Miss	€ / hour		
no cache	1,00	2,22	5,37	7,41
linear	0,50	1,10	2,66	3,67
0.1	0,36	0,80	1,93	2,67
0.3	0,28	0,62	1,50	2,07
0.5	0,22	0,49	1,18	1,63
1.0	0,16	0,36	0,86	1,19
2.0	0,12	0,27	0,64	0,89
4.0	0,10	0,22	0,54	0,74

Table 2

	DataStore RPH / Year Cost Calculation (Thousands)					
	600		1450		2000	
a distribution	€ / Hour	€ / Year	€ / Hour	€ / Year	€ / Hour	€ / Year
no cache	2,22	19.200 €	5,37	46.400 €	7,41	64.000 €
linear	1,10	9.504 €	2,66	22.968 €	3,67	31.680 €
0.1	0,80	6.912 €	1,93	16.704 €	2,67	23.040 €
0.3	0,62	5.376 €	1,50	12.992 €	2,07	17.920 €
0.5	0,49	4.224 €	1,18	10.208 €	1,63	14.080 €
1.0	0,36	3.072 €	0,86	7.424 €	1,19	10.240 €
2.0	0,27	2.304 €	0,64	5.568 €	0,89	7.680 €
4.0	0,22	1.920 €	0,54	4.640 €	0,74	6.400 €

Table 3

	Chart Preparation Table			
cache miss %	600	1450	2000	Server Cost
no cache	19.200 €	46.400 €	64.000 €	0 0 0
50%	9.504 €	22.968 €	31.680 €	22635 22635 22635
36%	6.912 €	16.704 €	23.040 €	22635 22635 22635
28%	5.376 €	12.992 €	17.920 €	22635 22635 22635
22%	4.224 €	10.208 €	14.080 €	22635 22635 22635
16%	3.072 €	7.424 €	10.240 €	22635 22635 22635
12%	2.304 €	5.568 €	7.680 €	22635 22635 22635

Figure 33. Caching Simulator Chart Calculation Parameters.

The figure above shows three tables. Their contents are described as follows:

- **Table 1** includes hard coded cache miss parameters derived from the results of the research paper "*Improving the Performance of Semantic Web Applications with SPARQL Query Caching*" [Martin 2010]. Additionally, it includes cost calculations for each rph category depending on the cache miss rate.
- **Table 2** calculates the costs per year resulting from the above parameters.

- **Table 3** finally, simply distributes Table 2 parameters in such a way in order to create a combined stacked bar chart in the **"User Input & Simulation Chart"** worksheet.

4.3.2 Download, Deploy and Run

The workbook for the prototype is downloadable from URL: https://websvn.deri.ie/wsvn/openiot/Deliverables/D512/ , filename: OpenIOT-D512-Cache Cost Evaluation Simulation Prototype.xlsx.

The file is simply executed as a windows application and is ready to use.

4.4 Cloud Optimization Integration in GSN and LSM

4.4.1 Functional specification

Figure 34 illustrates the integration of model-view sensor data index and query modules and the current components of OpenIoT, namely, GSN node and LSM.

In GSN node, we add the functionality module shown in Fig. 4 (a) which is in charge of segmenting sensor time series on the fly and assigning the segments to corresponding nodes in vs-tree. GSN node should maintain a vs-tree for each sensor time series in memory. Instead of sending raw sensor data points to the cloud store in LSM, GSN node only pushes the segments including the registration node, time domain, value range and model coefficients of the segment, to the key-value store in LSM. Then the key-value store HBase resident in LSM materializes the segment into corresponding row of the model-view sensor data table.

Regarding querying model-view sensor data, our proposed hybrid query processing scheme is embedded into LSM shown in Fig. 34 (b). When a query comes to the LSM, in the first step the intersection or stabbing search on vs-tree in LSM delimits a set of nodes that may host qualified segments. Then, the MapReduce based query processing is invoked within the LSM cloud to fetch the potential qualified segments from the key-value store, filter predicate-addressed segments and return the gridded values as query results. Based on the current architecture of OpenIoT, the components in the dot-dash red blocks of Fig. 34 (a) and (b) need to be implemented.

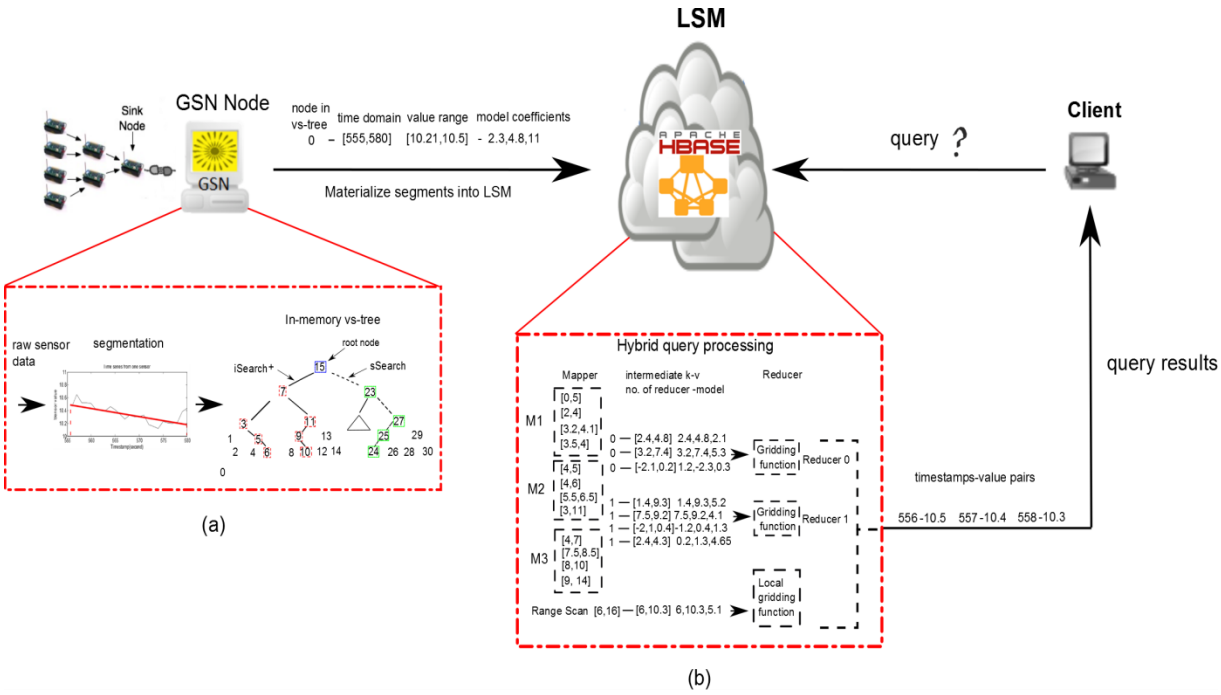


Figure 34. (a) GSN node. (b) Sensor data segments and KVI-index. (c) Key-value stores in LSM. (d) KVI-index and MapReduce based query processing.

4.4.2 Query specification

Model-view sensor data management only modifies the internal mechanism of indexing and querying sensor data, and therefore from the perspective of application side, end-users can submit queries as usual. The following categories of queries are supported by our model-view based approach in OpenIoT platform:

time point query: return the value of one sensor at a specific time point.

value point query: return the timestamps when the value of one sensor is equal to the query value. There may be multiple time stamps of which sensor values satisfy the query values.

time range query: return the values of one sensor during the query time range.

value range query: return the time intervals of which the sensor values are within the query value range. There may be multiple time intervals of which sensor values satisfy the query value range.

Concerning the query results, abstract functions of segments make little sense for end-users and hence the gridding phase is necessary in the query processing in order to generate user-friendly discrete data set as query results. Moreover, segment gridding helps eliminate the part of one segment that is outside the query range. Fig. 4 gives an example illustrating the query results from the hybrid query processing module in LSM are discrete data pairs representing the timestamps and sensor values.

4.4.3 Experimental evaluation

To show the feasibility of our approach, we have conducted a series of experiments described in this section. This proof-of-concept implementation is to be plugged to X-GSN, but we already show the feasibility of the techniques initially presented in Section 3.3. The results show important improvements in response time, compared to raw data value storage.

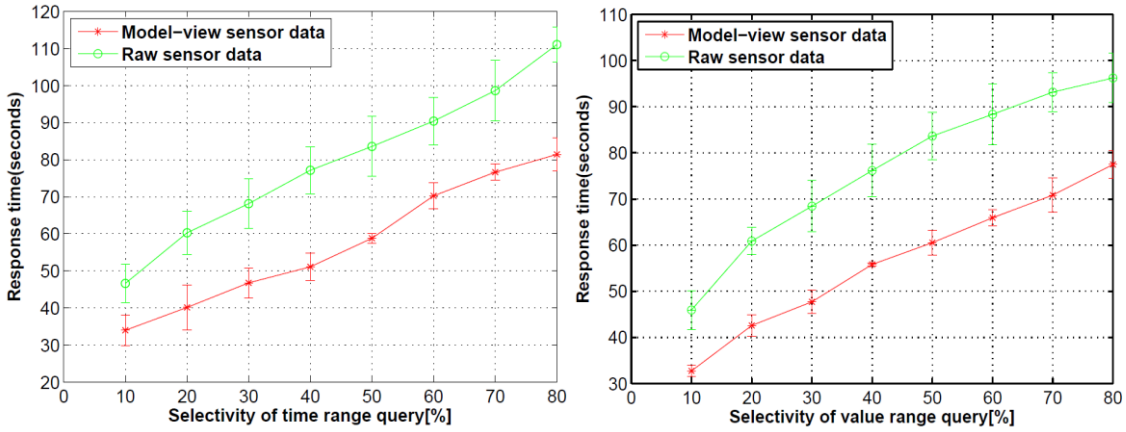
4.4.3.1 Setup

We employ accelerometer data from mobile phones as sensor data set. The size of raw sensor data is 22 GB including 200 million data points. After modeling, the modelled segments of the sensor data take 12 GB, while there are around 25 million modelled segments.

We developed our system using the versions of Hbase and Hadoop in Cloudera CDH 4.3.0. The experiments are performed on our own cluster that consists of 1 master node and 8 slaves. The master node has 64GB RAM, 3TB disk space (4 x 1TB disks in RAID5) and 12 cores, each of which is 2.30 GHz (Intel Xeon E5-2630). Each slave node has 6 cores 2.30 GHz (Intel Xeon E5-2630), 32GB RAM and 6TB disk space (3 x 2TB disks). Nodes are connected via 1GB Ethernet. In the experiment results, we refer to query selectivity as the ratio of the number of qualified modelled segments over that of total modelled segments.

4.4.3.2 Results

We compare the model-view sensor data query processing with conventional one over raw sensor data. Raw sensor data is a set of discrete data points each of which has associated timestamp and value. We create two tables, which respectively take the timestamp and sensor value as the row-keys, such that the query range or point can be used as keys to locate the qualified data points. The query processor invokes MapReduce to access the large size of data points for query results.



(a) Time range queries (b) Value range queries

Figure 35. Range query results.

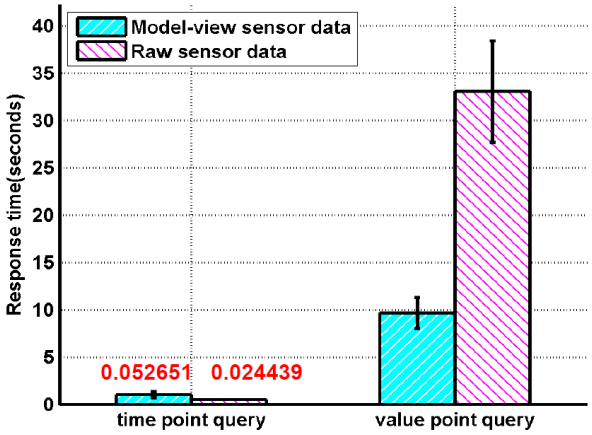


Figure 36. Point query results.

Figure 35. Range query results. (a) and (b) present the query response times for time and value range queries. As shown in Figure 35, model-view approach takes around 30% less time than the raw sensor data method for both time and value range queries. Although the raw sensor data based methods apply MapReduce to directly access the qualified tuples via the row-key based range scan, the amount of raw sensor data to process is much larger than that of the model-view approach. In Figure 36, the processing time of the raw data based method is 2x less than that of the model-view one in time point queries, because the raw data method can use the query time point as index key to directly access the relevant data points, while our hybrid approach requires to perform model filtering and gridding.

We also evaluate our KVI-Scan-MapReduce approach to compare with other model-view sensor data querying approaches. Moreover, we experimentally explore the factors that affect the performance of KVI-Scan-MapReduce. Please refer to [Guo 2013] for more details.

5 CONCLUSIONS

OpenIoT is working towards a blueprint framework and an associated middleware platform that could enable the on-demand formulation of IoT services over a cloud-computing infrastructure. Two of the main properties of the OpenIoT project are related to its ability to manage itself, towards optimizing the use of resources. In particular, as several users are serviced by OpenIoT and several services are concurrently running over the OpenIoT infrastructure, it is important to ensure that resources are used in an optimal way, which could also boost the availability and reliability of the infrastructure. To this end, OpenIoT employs a variety of optimization algorithms, which are structured within a framework for autonomic management of the OpenIoT infrastructure (i.e. without human mediation).

The deliverable has presented a number of algorithms and techniques that are employed for the management and resource optimization of the OpenIoT cloud platform. These algorithms target a number of different optimization objectives and employ a host of different mechanisms, in particular:

- Efficient scheduling functionalities are considered, mainly in order to ensure that OpenIoT streams data only in cases where these data have been requested and/or used.
- Caching mechanisms are prescribed with a view to accelerating access to sensor data that are frequently required, to sensor services that are frequently used, as well as to sensor data that reside in popular locations.
- Cloud optimization technologies are also presented, using model-based view for sensor query representation and processing.
- Utility-driven algorithms are also employed in order to maximize the net benefit (i.e. utility) measured as difference between the benefit of the provided information and the cost of maintaining the system in terms of energy consumption/bandwidth and the cost of ensuring privacy.
- Context-aware filtering for mobile environments, focused on efficiency on sensor data collection.
- Semantic Web and Linked data techniques are used in order to efficiently correlate different queries (e.g., sensor queries) to the OpenIoT system.
- The use of bandwidth allocation subject to spatial constraints is suggested in order benefit from spatial correlations and maximizes the energy efficiency of the network.

Several of the above algorithms are based on background research results of the partners, while other are tailored to the structure and the mode of operation of the OpenIoT system. In addition to describing these schemes, a specification and implementation design has been presented, identifying the components of the

OpenIoT architecture that host and/or support these mechanisms. Specifically, we have introduced the implementation details for the Dynamic Sensor control module, the Utility-based optimization, the Cloud optimization integration, and the simulation of caching scenarios. The project intends also to select some of the schemes for integration over the open source OpenIoT platform. The rest schemes would serve as exercises and projects for the open source community of the project, while also being excellent themes for (open source) promotion activities like summer of code.

6 REFERENCES

[Abadi 2005] D. J. Abadi, S. Madden, and W. Linder, "REED: Robust, Efficient Filtering and Event Detection in Sensor Networks", Proc. of the 31st VLDB conference, Trondheim, Norway, 2005, pp. 769-780.

[Ahmad 2004] Y. Ahmad and U. Cetintemel. Network-aware query processing for stream-based applications. In VLDB, 2004.

[Amazon Web Services], “Amazon Web Services, [Online]. Available: http://aws.amazon.com/s3/#pricing.

[Bizer 2009]: The Berlin SPARQL Benchmark . In: International Journal on Semantic Web & Information Systems, Vol. 5, Issue 2, Pages 1-24, 2009.

[Chand 2006] N. Chand, R.C. Joshi., M. Misra, “Cooperative caching strategy in mobile ad hoc networks based on clusters,” Springer Wireless Personal Communications, pp. 41-63, issue 1, October 2006.

[Chow 2007] C.Y. Chow, H.V.Leong, A.T.S. Chan, “GroCoca: Group-based peer-topeer cooperative caching in mobile environment,” IEEE Journal on Selected Areas in Communications, vol. 25, no. 1, January 2007.

[Clark 2003] Clark, D., Partridge, C., Ramming, J. C., Wroclawski, J. T. “A Knowledge Plane for the Internet” SIGCOMM 2003, Karlsruhe, Germany, 2003.

[Deshpande 2006] Deshpande, Amol and Madden, Samuel. MauveDB: supporting model-based user views in database systems. SIGMOD, 2006.

[Ding 2008] Ding, Hui and Trajcevski, Goce and Scheuermann, Peter and Wang, Xiaoyue and Keogh, Eamonn. Querying and mining of time series data: experimental comparison of representations and distance measures. VLDB Endowment, 2008.

[DMTF-CIM] DMTF, Common Information Model Standards (CIM]. http://www.dmtf.org/standards/standard_cim.php

[Feige 2007]U. Feige et al. Maximizing non-monotone submodular functions. In Proc. of FOCS, 2007.

[Google Inc.], “Google Cloud SQL Pricing,” [Online]. Available: https://cloud.google.com/pricing/cloud-sql.

[Guo 2012] Guo, Tian and Yan, Zhixian and Aberer, Karl. An adaptive approach for online segmentation of multi-dimensional mobile data. Proc. of MobiDE, SIGMOD Workshop, 2012.

[Guo 2013] Guo, Tian; G. Papaioannou, Thanasis; Aberer, Karl. Model-View Sensor Data Management in the Cloud. IEEE International Conference on Big Data 2013 (IEEE BigData 2013), Santa Clara, California, USA, October, 2013.

[Heinzelman 1999] Heinzelman, W.R., Kulik, J., Balakrishnan, H.: Adaptive Protocols for Information Dissemination in Wireless Sensor Networks. In: MOBICOM, Seattle, WA. (Aug. 1999) 174–185

[Intanagonwiwat 2000] Intanagonwiwat, C., Govindan, R., Estrin, D.: Directed Diffusion: a Scalable and Robust Communication Paradigm for Sensor Networks. In: MOBICOM, Boston, MA. (Aug. 2000) 56–67

[Laurila2012] Laurila, J. K., Gatica-Perez, D., Aad, I., Blom, J., Bornet, O., Do, T.-M.-T., Dousse, O., Eberle, J.,and Miettinen, M. The mobile data challenge: Big data for mobile computing research. In Proc. Mobile Data Challenge by Nokia Workshop, in conjunction with Int. Conf. on Pervasive Computing (2012).

[Lee 2006] K. C. K. Lee, W.-C. Lee, B. Zheng, and J. Winter. Processing multiple aggregation queries in geo-sensor networks. In Proceeding of the 11th International Conference on Database Systems for Advanced Applications (DASFAA), pages 20—34, 2006.

[Lephuoc 2011] Danh Le-Phuoc, Hoan Nguyen Mau Quoc, Josiane Xavier Parreira, and Manfred Hauswirth, The Linked Sensor Middleware: Connecting the real world and the Semantic Web, at 9th Semantic Web Challenge co-located with 10th International Semantic Web Conference – ISWC 2011, October 23-27, 2011 Bonn, Germany.

[Li 2004] Li, X., Huang, Q., Zhang, Y.: Combs, Needles, Haystacks: Balancing Push and Pull for Discovery in Large-Scale Sensor Networks. In: ACM SenSys, Baltimore, MD. (Nov. 2004)

[Li 2009] J. Li, S. Li, J. Zhu, “Data Caching Based Queries in Multi_sink Sensor Networks,” IEEE 5th International Conference on Mobile Ad-hoc and Sensor Networks, 2009.

[Maddan 2002] Maddan, S., Franklin, M.J., Hellerstein, J.M., Hong,W.: TAG: a Tiny Aggregation Service for Ad-Hoc Sensor Networks. In: OSDI. (Dec. 2002)

[Madden 2005] S. R. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong. Tinydb: an acquisitional query processing system for sensor networks. ACM Transactions on Data Base Systems (TODS), 30(1):122—173, 2005.

[Martin 2010] Martin, M., Unbehauen, J., and Auer, S. Improving the Performance of Semantic Web Applications with SPARQL Query Caching. *In* Proceedings of 7th Extended Semantic Web Conference ESWC 2010. June 2010.

[Meng 2008] Min Meng, Jie Yang, Hui Xu, Byeong-Soo Jeong, Young-Koo Lee and Sungyoung Lee, «Query Aggregation in Wireless Sensor Networks», International Journal of Multimedia and Ubiquitous Engineering, Vol. 3, No. 1, January 2008

[Mood 1974] Mood A.M., Graybill F.A., Boes D.C. (1974) Introduction to the Theory of Statistics (3rd Edition). McGraw-Hill.

[Olston 2003] C. Olston, J. Jiang, and J. Widom. Adaptive filters for continuous queries over distributed data streams. In SIGMOD, 2003.

[Papaioannou 2011] T. Papaioannou, N. Bonvin and K. Aberer. Scalia: An Adaptive Scheme for Efficient Multi-Cloud Storage

[Papaioannou 2011] T.G. Papaioannou and Riahi, M. and Aberer, K. Towards Online Multi-model Approximation of Time Series., MDM, 2011.

[Pietzuch 2006] P. Pietzuch, J. Ledlie, J. Shneidman, M. Roussopoulos, M. Welsh, and M. Seltzer. Network-aware operator placement for stream-processing systems. In ICDE, 2006

[Qian2012] Qian, F., Wang, Z., Gao, Y., Huang, J., Gerber, A.,Mao, Z., Sen, S., and Spatscheck, O. Periodic transfers in mobile applications: network-wide origin, impact, and optimization. In Proc. of the 21st international conference on World Wide Web, WWW'12, ACM (2012), 51-60.

[Ratnasamy 2002] Ratnasamy, S., Karp, B., Yin, L., Yu, F., Estrin, D., Govindan, R., Shenker, S.: GHT: A Geographic Hash Table for Data-Centric Storage. In: WSNA, Altanta, GA. (Sep. 2002)

[Riahi 2013] M. Riahi, T. G. Papaioannou, I. Trummer and K. Aberer. Utility-driven Data Acquisition in Participatory Sensing. 16th International Conference on Extending Database Technology (EDBT), Genoa, Italy, March 18-22, 2013.

[Serrano 2008] Serrano, J. Martin. 2008 “Management and Context Integration Based on Ontologies for Pervasive Service Operations in Autonomic Communication Systems”, PhD Thesis, UPC 2008.

[Serrano 2012] Serrano J. Martin “Applied Ontology Engineering in cloud Services, Networks and Management Systems”, Springer publishers, march 2012, Hardcover, p.p. 222 pages, ISBN-10: 1461422353, ISBN-13: 978-1461422358.

[Thiagarajan 2008] Thiagarajan, Arvind and Madden, Samuel. Querying continuous functions in a database system. SIGMOD, 2008.

[TMF-SID] SID - Shared Information Data model. http://www.tmforum.org/InformationManagement/1684/home.html

[TMN-M3050] Telecommunications Management Networks - Management Services approach - Enhanced Telecommunications Operations Map (eTOM) http://www.catr.cn/cttlcds/itu/itut/product/bodyM.htm

[TMN-M3060] Telecommunications Management Networks - Principles for the Management of Next Generation Networks. http://www.catr.cn/cttlcds/itu/itut/product/bodyM.htm

[Trigoni 2005] N. Trigoni, Y. Yao, A. J. Demers, J. Gehrke, and R. Rajaraman. Multi-query optimization for sensor networks. In Proceeding of the first IEEE International Conference on Distributed Computing in Sensor Systems (DCOSS), pages 307—321, 2005.

[Yao 2002] Y. Yao and J. Gehrke. The cougar approach to in-network query processing in sensor networks. SIGMOD Record, 31(3):9—18, 2002.

[Ye 2002] Ye, F., Luo, H., Cheng, J., Lu, S., Zhang, L.: A Two-Tier Data Dissemination Model for Large-Scale Wireless Sensor Networks. In: MOBICOM, Atlanta, GA. (Sep. 2002) 148–159.

OpenIoT 2013

