MONASH University

Performance and Cost Driven Data Storage and Processing for IoT Context Management Platforms

Alexey Medvedev

Faculty of Information Technology
Monash University

Supervisors:

Dr. Pari Delir Haghighi

Dr. Sea Ling

Professor Arkady Zaslavsky

Dr. Brano Kusy

Dissertation submitted for partial fulfilment of the requirements for the Degree of

Doctor of Philosophy

January 2020
Monash University

Copyright notice

Declaration

This thesis is an original work of my research and contains no material which has been accepted for the award of any other degree or diploma at any university or equivalent institution and that, to the best of my knowledge and belief, this thesis contains no material previously published or written by another person, except where due reference is made in the text of the thesis.

Signature:

Print Name: Alexey Medvedev

Date: 06/01/2020

To my family

Acknowledgements

I would like to thank all the people who provided support and advice during the last three years. First of all, I would like to thank my supervisors, Prof. Arkady Zaslavsky, Dr. Pari Delir Haghighi, Dr. Chris Ling, Dr. Brano Kusy and Dr. Maria Indrawan-Santiago. Arkady, thank you for believing in me and providing the chance for looking at life from an absolutely different perspective. Pari, without your control over the project scope it would be hard to meet the deadlines and other important criteria. Maria, thank you for finding time for technical discussions during the early stage of the project. Chris, thank you for your timely advices.

Dr. Alireza Hassani, it was a pleasure for me to participate in the CoaaS project side by side. Our discussions were slowly shaping my view on the problems we were facing and, also, lead to a number of publications. It was fascinating to see how our initial drawings on the blackboard were slowly becoming reality.

Dr. Gleb Belov, Dr.Sergei Polyakovskiy, thank you for your advices in the area of mathematical optimisation and statistical data analysis.

I also would like to express my special thanks to Monash University staff, especially Helen Cridland, Julie Holden, and Allison Mitchell. Your efforts in organisation of all the internal processes really helped me to focus on important things.

I would like to express my sincere gratitude to my friends, who I met and who were around during this journey, especially Dr. Victor Suturin, George Tyukavin and Dinislam Abdulgalimov.

And of course, I would like to acknowledge my family. Thank you for your support and love.

Publications arising from the project

This PhD research has resulted in thirteen peer-reviewed publications. These publications include three journal articles, and ten international conference papers. The list of these publications is presented below.

Hassani, A., **Medvedev, A**., Delir Haghighi, P., Ling, S., Zaslavsky, A., Jayaraman, P.P.: Context Definition and Query Language: Conceptual Specification, Implementation, and Evaluation. Sensors 2019, Vol. 19, Page 1478. 19, 1478 (2019).

Anagnostopoulos, T., Zaslavsky, A., Sosunova, I., Fedchenkov, P., **Medvedev, A**., Ntalianis, K., Skourlas, C., Rybin, A., Khoruzhnikov, S.: A Stochastic Multi-agent System for IoT-enabled Waste Management in Smart Cities. Waste Manag. Res. (2018).

International Conferences

Hassani, A., **Medvedev, A.**, Zaslavsky, A., Haghighi, P. D., Jayaraman, P. P., & Ling, S. (2019). Efficient Execution of Complex Context Queries to Enable Near Real-Time Smart IoT Applications. Sensors 2019, Vol. 19, Page 5457, 19(24), 5457.

Medvedev, A., Hassani, A., Haghighi, P.D., Ling, S., Indrawan-Santiago, M., Zaslavsky, A., Fastenrath, U., Mayer, F., Jayaraman, P.P., Kolbe, N.: Situation Modelling, Representation, and Querying in Context-as-a-Service IoT Platform. In: 2018 Global Internet of Things Summit (GIoTS). pp. 1–6. IEEE (2018).

Hassani, A., **Medvedev, A**., Delir Haghighi, P., Ling, S., Indrawan-Santiago, M., Zaslavsky, A., Jayaraman, P.P.: Context-as-a-Service Platform: Exchange and Share Context in an IoT Ecosystem. In: 2018 IEEE International Conference on Pervasive Computing and Communications Workshops (PerCom Workshops). pp. 385–390. IEEE (2018).

Sosunova, I., Zaslavsky, A., Anagnostopoulos, T., Fedchenkov, P., Sadov, O., **Medvedev, A**.: SWM-PnR: ontology-based context-driven knowledge representation for IoT-enabled waste management. In: Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics). pp. 151–162 (2017).

Fedchenkov, P., Zaslavsky, A., **Medvedev, A**., Anagnostopoulos, T., Sosunova, I., Sadov, O.: Supporting Data Communications in IoT-Enabled Waste Management. In: Lecture

Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics). pp. 163–174 (2017).

Medvedev, A., Indrawan-Santiago, M., Delir Haghighi, P., Hassani, A., Zaslavsky, A., Jayaraman, P.P.: Architecting IoT context storage management for context-as-a-service platform. In: GIoTS 2017 - Global Internet of Things Summit, Proceedings. pp. 1–6. IEEE (2017).

Anagnostopoulos, T., Zaslavsky, A., Kolomvatsos, K., **Medvedev, A.**, Amirian, P., Morley, J., Hadjieftymiades, S.: Challenges and Opportunities of Waste Management in IoT-Enabled Smart Cities: A Survey. IEEE Trans. Sustain. Comput. (2017).

Hassani, A., Delir Haghighi, P., Jayaraman, P.P., Zaslavsky, A., Ling, S., **Medvedev, A.**: CDQL: A Generic Context Representation and Querying Approach for Internet of Things Applications. In: Proceedings of the 14th International Conference on Advances in Mobile Computing and Multi Media - MoMM ’16. pp. 79–88. ACM Press (2016).

Medvedev, A., Hassani, A., Zaslavsky, A., Jayaraman, P.P., Indrawan-Santiago, M., Haghighi, P.D., Ling, S.: Data ingestion and storage performance of IoT platforms: Study of OpenIoT. In: Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics). pp. 141–157 (2017).

Medvedev, A., Zaslavsky, A., Santiago, M.I., Haghighi, P.D., Hassani, A.: Storing and indexing IoT context for smart city applications. In: Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics). pp. 115–128 (2016).

Abstract

Internet of Things is a very active research area with great commercialisation potential. Context-awareness in IoT applications has a profound impact on smartness, relevance, adaptability, dependability and flexibility of such applications. Moreover, context-awareness can be seen as a key to breaking the data silos barrier, which still constraints the development of IoT System-of-Systems. For this, a special-purpose platform needs to be researched and developed. The main aim of this platform is processing context requests coming from heterogeneous entities, thus, providing Context-as-a-Service. We refer to such a platform as a CoaaS platform.

The CoaaS platform will have to cope with potentially big data generated from billions of devices. The amount of context, metadata, annotations in IoT ecosystems equals and may even exceed the volume of raw data. At the same time, the CoaaS platform will have to process queries in near real-time.

In this dissertation, we address the challenges of building a context storage management system (CSMS) as a core component of the CoaaS platform. The requirements to context query processing time, service discovery and selection, data encapsulation and overall efficiency determine the need for organizing an internal horizontally scalable and high-performing storage subsystem as a part of CoaaS platform. However, no substantial R&D has been carried out on how IoT-scale context can be stored, indexed, retrieved and provisioned to various IoT services. This research is concerned with how to architect a scalable context storage management system that can cost-efficiently and effectively respond to context queries from CoaaS platform, maintain the agreed quality of service, and provide proactive adaptation and caching.

We exemplify, validate, and evaluate the usage of the proposed CSMS via a smart city use case study, but the proposed solution can be deployed in other application domains as well.

The thesis contains 204 pages, 74 figures, and 4 tables. This work resulted in thirteen peer-reviewed publications.

Table of Contents

List of Figures

List of Tables

Chapter 1: Introduction

1.1 THE INTERNET OF THINGS AND CONTEXT AWARENESS

The modern world is more technology-dependent and technology-driven than any time before. Connected Smart Objects are taking the place of ordinary things, opening a new area for innovations. Most of the organisations which own and manage physical infrastructure, have already realised the potential of the Internet-of-Things (IoT) technological stack for solving internal tasks of managing the infrastructure, as well as providing innovative services for customers. Still, there is an enormous hidden capacity for developing services that could be built on top of data, which is spread across the IoT silos owned by various organisations. These services can potentially make our daily life more efficient and comfortable.

The area of building applications which are dependent on real-time data about external entities is often referred to as *Context-Awareness* (CA) [1]; the relevant data, which might also be pre-processed and aggregated, is referred to as *context*. Despite the potential of context-aware computing for the IoT, the progress in the introduction of these kind of services in the real world is in its infancy. The main problems are the lack of universal acceptance, standardisation, and accessible technologies. Even if the relevant data is potentially reachable, it is hard for a software developer to find these data, not even mentioning the problem of processing data from thousands of sources when the data is represented in different formats and has a different level of precision or trust.

A more feasible approach to the problem can be the communication through a middleware platform, which facilitates interoperability between multiple IoT silos and serves as an aggregator and a redirector at the same time. The main functionalities of such a platform include searching for data sources, retrieving and caching relevant data, building aggregations on top of these data, and answering queries from context consumers. Accordingly, we can say that a middleware with the described functionalities provides Context-as-a-Service in an ecosystem, where any entity can provide context, (acting as a *context provider*), or query context, (acting as a *context consumer*).

Advanced middleware platforms which comprise functionalities of (i) IoT marketplace, (ii) gateways to multiple data sources, (iii) subscription mechanisms, (iv) features for performing aggregations, reasoning, analytical functions, and (v) advanced sensor data management, are called *Context Management Platforms* (CMP) [2].

We can distinguish three main phases of IoT evolution, which are (i) the M2M phase, (Machine to Machine), (ii) the IoT silo phase, and (iii) the IoT ecosystems phase [3]. To date, significant progress in building IoT silos has been achieved. Currently, we are entering the third phase of IoT evolution (the IoT ecosystems phase), which is characterised by numerous horizontal integrations between the IoT silos. At this stage, smart devices can communicate with each other spontaneously, without being locked in a silo of a company that produces the device or owns it. We provide a more detailed discussion of similarities and differences between IoT platforms and CMPs, as well as the discussion of IoT phases and the definition of the IoT ecosystem in Chapter 2. As context awareness plays a key role in enabling IoT ecosystems, Context Management Platforms are attracting substantial research efforts nowadays.

1.2 IOT PLATFORMS AND CONTEXT MANAGEMENT PLATFORMS

The discussion of IoT middleware is often formed around the term '*IoT platform*'. However, we need to highlight the difference between the terms *IoT platform* and *Context Management Platform*. Once the IoT started to gain momentum, it was immediately commercialised by software vendors. Eventually, as the vendors were searching for fast commercial outcomes, the term IoT platform was used for the software, which is an enabler of IoT silos. These platforms are designed to be governed by the owner, (e.g. company/developer of IoT applications and devices), with full control over the data, which pass through this platform.

However, for the IoT ecosystems phase, we need a different type of platform where the spontaneous horizontal integrations would be possible. For that, an IoT platform will need to treat all the participants equally, based on the established rules of context exchange. Recently, the terms *'Context Information Management' (CIM)* and *'Context Management Platform' (CMP)* have received the community recognition. Currently, the standardisation efforts in the area of CMPs and context query languages are led by the ETSI CIM working group [4], where the NGSI-LD language and the FIWARE [5] platform are the basis for the proposed standard. Consequently, in our research, we use the term CMP to refer to a platform designed for the needs of the IoT ecosystems phase. At the same time, CMPs process the same IoT data as the IoT (silo) platforms. Thus, in this research, we look at IoT platforms as a closely relevant field of study, and many principles are applicable in both fields.

For our research, we use a broad definition given by Dey: “Context is any information that can be used to characterise the situation of an entity. An entity is a person, place, or object that is considered relevant to the interaction between a user and an application, including the user and applications themselves” [6].

1.3 BIOTOPE PROJECT INSPIRED USE CASES

This research is related and contributes to the bIoTope project[1] [7], which aims to build an IoT open Innovation Ecosystem for Connected Smart Objects, (CSO), with the primary application domain of Smart Cities. The bIoTope project is a part of European Union's Horizon 2020 Programme[2]. Among the key objectives of the bIoTope project, the following are directly connected with the scope of this research: (i) enabling interoperability between smart objects and vertical IoT silos by developing standards for open API’s, (ii) enabling creation of novel intelligent context-aware services, (iii) establishing a framework which will facilitate access to IoT data with respect to security, privacy, and trust.

The meetings we held during the bIoTope general assemblies, and other collaborative meetings, considerably helped to scope the project and understand the roles and requirements of IoT ecosystem stakeholders. Based on the discussed scenarios, we could deduce the set of features, which are essential for a CMP. We discuss these features in Chapter 3.

The main scenarios were the smart mobility and smart waste management. The smart mobility use case was developed in collaboration with the BMW Group. The initial use cases contained such tasks as searching for a vacant carpark and searching for a charging point for an electric vehicle. In this use case, a smart car or the backend of the navigation system is requesting the contextual information from the CMP about the availability of parking facilities around the needed location. The providers of contextual information are registered in the CMP, when the query is executed, CMP retrieves the needed data from the providers, processes it and returns the resulting answer to the consumer. This scenario is used as an illustration of query processing in Chapter 5, more details can be found in [8] and [9].

Then, we proceeded to a more complex scenario, where the automatic preconditioning of an electric car was triggered, based on the subscription for situation monitoring. Preconditioning is the enabling of a heater or an air conditioning system in advance to prepare the perfect environment for the driver. The preconditioning procedure should be started

[1] https://biotope-project.eu
[2] https://ec.europa.eu/research/fp7/index_en.cfm

automatically when the driver is likely to use the vehicle. In the most simple case, CMP is monitoring the location of the user, and when the distance between the driver and the vehicle is decreasing and a driver has an event planned in the calendar, CMP sends a notification to start the preconditioning procedure. The details of the query for the preconditioning scenario can be found in chapter 4.

In general, there are two types of users in the scenarios above. The end user, (e.g. the driver of a car), is a secondary user from the CMP perspective, as these users do not interact directly with the platform. In fact, an application developer is the primary user of the platform, as it is the person/team who composes context queries, thus integrating the application/device of an end user to the ecosystem. The critical point that should be taken into account is that the primary user (developer), is not tightly coupled with the CMP platform. For this reason, it is hard to manually tune the middleware for optimal performance, like it is done by administrators in enterprise systems or IoT silos.

For instance, consider a developer, who is in charge of creating a context-aware application that suggests parking facilities to drivers. In order to develop such an application, there are several challenges that need to be addressed. At first, the developer needs to retrieve data from IoT-enabled parking facilities in near real-time. This challenge is hardened by the fact that the mentioned facilities are owned by different organisations and might follow different standards and protocols of data access and exchange. Moreover, in order to improve the quality of recommendations, these data need to be enriched by considering additional context, such as weather conditions, safety of the area and the user's profile.

A scenario with vehicle preconditioning requires defining how the monitoring of incoming events should be handled. In this scenario, for instance, events contain the current location of the vehicle's owner, sent by the smartphone. The process of monitoring involves the windowing and trend analysis functionality, as well as other CMP mechanisms.

In addition to smart parking, and charging and preconditioning scenarios, we considered other smart city use cases, including such domain as waste management and safety in the city. For instance, in Chapter 4 we illustrate the design of the proposed system based on a scenario and a corresponding query, which enable a smartphone application to find appropriate smart garbage containers around a certain location. An early work which describes how a CMP can enable the safety of school students during the pick-up times by facilitating the secure exchange of context between IoT devices of classmates, their parents, school, bus, and other entities is

described in [10].We found that enabling the described scenarios with existing CMP prototypes is burdensome. For this reason, as a part of a bIoTope project, our research group started the research and development of the Context-as-a-Service (CoaaS) platform to address the lack of modern context exchange middleware for the IoT.

1.4 CONTEXT-AS-A-SERVICE (COAAS) PLATFORM

The CoaaS platform aims to support application developers of CA applications in expressing their needs for contextual information in a more flexible way, and avoid the tedious effort to implement application-dependent event processing pipelines to detect situations. The CoaaS platform has significant differences with other research efforts made in the area of CMP. For instance, the query interface is based on a specifically designed Context Definition and Query Language (CDQL) [10]. There are also significant differences in the approach to data retrieval, storage and processing.

The main motivation behind developing CoaaS is providing a generic and standard way to define, advertise, discover/acquire, and query context. In other words, CoaaS facilitates context exchange between IoT entities. CoaaS is designed to follow the XaaS (Everything as a Service) paradigm. However, the approach is different from well-known SaaS (Software as a Service), PaaS (Platform as Service), or IaaS (Infrastructure as a Service) paradigms. Figure 1.1 depicts an overview of CoaaS platform in an IoT ecosystem. As it is shown, context consumers send their contextual requirements to CoaaS as context queries, which are represented in CDQL. CDQL supports two types of queries, PULL-based query, where the query is only executed once, and PUSH-based query which enables continual situation monitoring. We provide more detailed information about the CoaaS platform and CDQL language in Chapter 3, as the background for the main scope of this research.

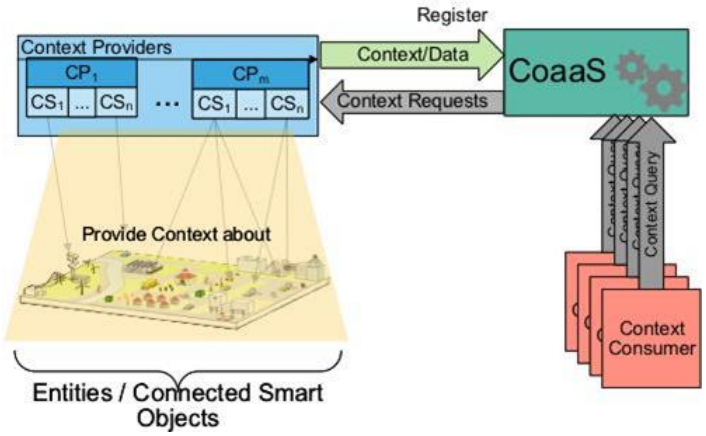


Figure 1.1 - An overview of Context-as-a-Service in IoT ecosystem

When a CoaaS platform is deployed for the operation in IoT ecosystem, none of the context consumers or context providers have full control over the platform. The platform provides access to contextual information by returning results of consumers' queries; thus the platform provides a service, which can be billed accordingly (e.g. per query).

1.5 CONTEXT STORAGE AND MANAGEMENT SYSTEM

The fundamental challenge that will be faced by CoaaS or any other CMP is to process and manage enormous amounts of context stemming from IoT context providers and other data sources. Some of this data will have to be stored for such purposes as acquiring historical context, mining patterns, access control, service discovery and other purposes. The need for storing and processing context stemming from IoT big data in near real-time, dictates the need for organising a specialised storage component in the CoaaS platform. We call this component a Context Storage Management System (CSMS). Researching, architecting, implementing and evaluating the CSMS is the main objective and contribution of this PhD project, which is described in this thesis.

Apart from the analysis and implementation of the main components, we have focused on researching the cache efficiency problem. As the CoaaS platform will operate in the cloud environment, we can potentially cache all the related data and refresh every data item at a high rate. However, running a system with such a strategy will require an infinite amount of monetary resources to afford paying for the infrastructure. At the same time, caching is necessary to reduce the number of expensive calls to remote providers as well as lowering the query serving time. Finding an optimal balance between the resource consumption and query execution time is an essential part of the project. Research, development, and evaluation of the cache management strategies and models are at the centre of the research scope.

1.6 CHALLENGES AND NOVELTY OF THE PROJECT

As discussed above, the transition from an IoT silo phase to IoT ecosystems phase occurs slowly. One of the main challenges for creating the middleware that will be able to facilitate the interaction in such an ecosystem is to get a clear view on the required functionalities and the forms in which these functionalities should be delivered.

Moreover, there are a number of technical challenges, which influence the flow of the CSMS project. These challenges are (i) the lack of common approach to modelling and querying, (ii) the gap between the modern data storage/processing technologies and existing

CMP prototypes, (iii) the need to serve queries based on the data both from the internal storage and from the context providers, and (iv) self-adaptation with a focus on cloud-based deployment.

Challenge (i) can be referred to as a war of standards. While many proposals are made, they lack the support of real successful large-scale integrations. In recent times, two distinct directions have been considered, which are the semantic modelling and the document-based mark-up modelling. A fusion of these approaches can be seen as the third option.

Challenge (ii) is related to the problem of bridging the gap between the modern technologies commonly accepted in the industry (such as databases, analytic frameworks, event processors) and the ways of context modelling and querying need to be dealt with. While few prototypes make use of modern data storage and processing frameworks, there is a need to develop a solution which can effectively harness these technologies, leaving the high-level access to the consumer through a well-balanced API.

We refer to the challenge (iii) as Not only Database – Not only Redirector (NoD-NoR) mode of CMP operation. In the database mode, all the data is always retrieved from the internal datastore to service the query. On the other hand, the redirector mode involves retrieving all the data from external sources. The NoD-NoR approach combines these two modes. This ability adds complexities at the development stage, but can also significantly improve the performance and cost-efficiency.

Challenge (iv) is related to the previous challenge, as well as to the loose coupling of consumers with the middleware. To understand which data should be kept in the internal storage and when it should be refreshed, a middleware needs to contain mechanisms that will be monitoring the query load and the behaviour of data sources. Based on the collected data, these self-adaptation mechanisms should produce the optimal strategy. We discuss these challenges in detail in Chapter 2. The challenges (i-iii) directly influence the functional and architectural sides of the R&D process, which are described in Chapters 3 and 4. The challenges (iii-iv) require research in the area of adaptive caching, prefetching, and task allocation. We tackle these challenges in Chapters 5 and 6.

One of the main differences of a CMP with many other types of data-centric systems is the lack of clearly defined pattern of data retrieval and ingestion. A typical database is usually designed and maintained to serve a particular application or a set of applications, and, consequently, can be tuned for specific loads. However, a modern CMP should automatically

adapt to the load, which is generated by context consumers (queries) and context providers (ingestion). We cannot rely on manual tuning of a CMP for a particular use case or a set of use cases. In general, a CMP should have a good performance in a wide range of use-cases.

The freshness of context and the latency of access to context, along with several other parameters, form the metric called Quality of Service (QoS). On the other hand, the price of services provided by the middleware is referred to as the Cost of Context (CoC). Linking these groups of parameters is achieved by establishing Service Level Agreements (SLAs) between the consumers, the middleware, and providers. Then, these parameters can be used to build cost-based models used for cache management, prefetching, and other self-adaptation tasks of the CMP.

The area of cache management for IoT data, where the freshness, cost, latencies, undefined patterns of access, and other specific parameters important for CMP operation in NoD-NoR mode are taken into account, are not well researched. The problem becomes even more complicated when multiple SLAs are taken into account, for instance, consumers can subscribe to a platinum, golden, or silver plan. We investigate this problem in Chapters 6 and 7.

1.7 RESEARCH QUESTIONS

Based on the analysis of existing projects in the area, along with the requirements of the CoaaS platform and the existing gaps in context storage management strategies, the following research questions are proposed:

RQ1 - How to architect a data storage and processing system that can effectively respond to context queries in the Context-as-a-Service (CoaaS) IoT platform?

RQ1.1 - What are the main factors and CoaaS requirements that can influence the design of a storage system?

RQ1.2 – How to design the needed software modules, and what functionalities should these modules provide to satisfy all the identified requirements?

RQ2 - How to balance CoaaS platform’s performance vs. cost while complying with CoaaS constraints?

RQ2.1 - What are the main caching strategies, efficiency criteria, and monitored metrics?

RQ2.2 - How to build a cost-efficiency model that can govern proactive caching for dynamic CoaaS loads?

RQ2.3 - How multilayered cache management techniques influence the performance of the CoaaS platform?

In this chapter, we provided the background discussion of the CMP field. We highlighted the research gaps and perspectives, focusing on bridging such areas as the modern data storage and processing frameworks, CSMS architectural requirements, and relevant self-adaptation techniques. In the next chapter, we present a comprehensive literature review, which forms the theoretical base for the project.

1.8 RESEARCH CONTRIBUTIONS

In this section, we provide a list of main contributions of the PhD project to the body of knowledge.

- An architecture of a novel data storage and processing system, which is capable of serving CoaaS (CDQL) requests as well as support the CoaaS platform requirements and use-cases.

- A set of context caching strategies, which can facilitate cost efficient CSMS operation, taking into account the possibility of multiple SLAs, as well as the CoaaS approach to context querying and situation definition.

- Mathematical methods to estimate and optimise the cost of CoaaS operation for a planning period to address multiple SLAs established to cover cost and time constraints. Other CoaaS unique features and constraints are also taken into account.

-Implementation and evaluation of a prototype of the proposed CSMS, including main modules, CDQL wrapper and mathematical models, which support the management of operations in CSMS.

1.9 THESIS STRUCTURE

In Figure 1.2, the structure of the thesis is presented.

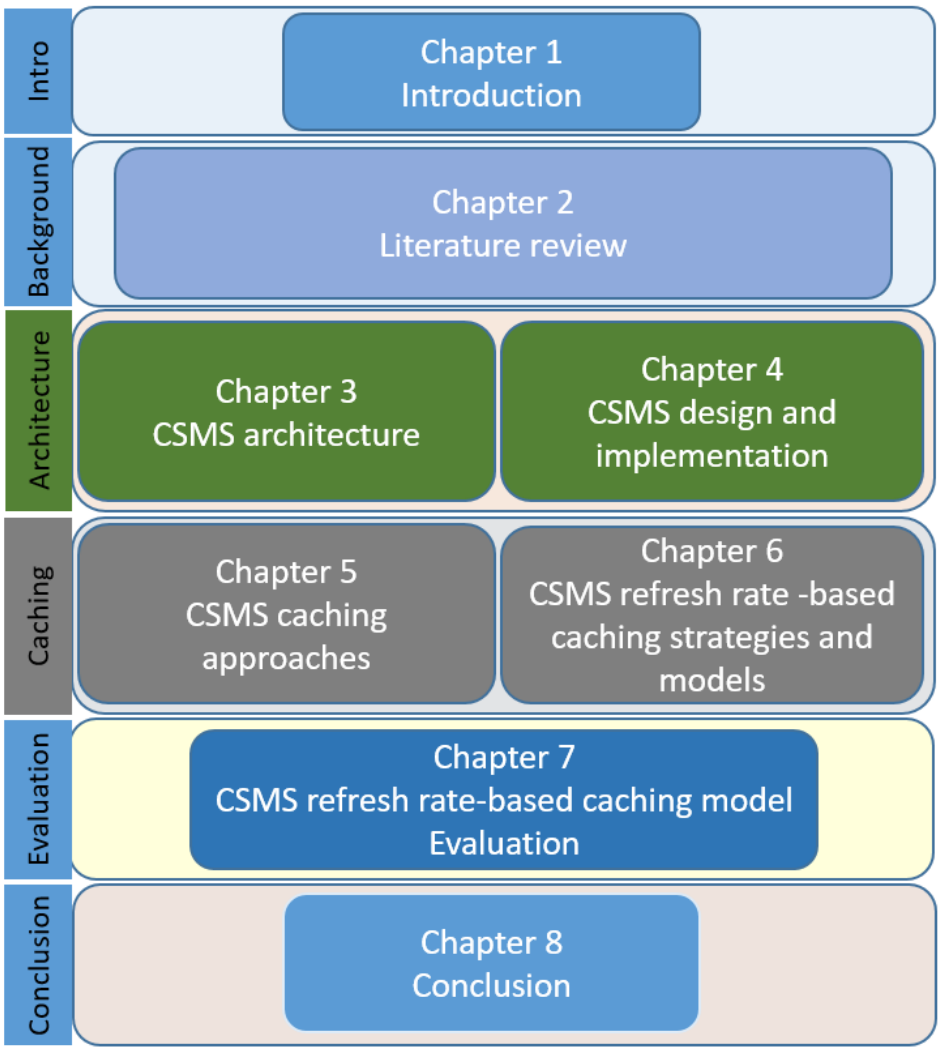


Figure 1.2 - Thesis structure

The background part includes the literature review. The architectural part contains the requirements to CSMS and the proposed architecture. The requirements arise from the analysis of use cases, CoaaS project background and the literature review. The architectural part also contains the description of the design and implementation of CSMS. The caching part contains the proposed caching approaches and CSMS refresh rate-based caching models. The evaluation part includes the validation and evaluation of the proposed models. The conclusion part concludes the thesis.

Chapter 2: Literature review

2.1 INTRODUCTION

The focus of this research is on the Context Storage Management System (CSMS) for the Context as a Service (CoaaS) platform. In this chapter, we present the analysis of publications, which are relevant for the CSMS component. The relevant fields include such areas as IoT context modelling, storage, processing caching and prefetching. We review both the IoT platforms and the Context Management Platforms (CMP), as they share numerous similarities. Additionally, we focus on self-adaptation mechanisms, which are useful for the efficiency and performance tuning of IoT and CMP middleware.

The chapter is structured in the following way: In Section 2.2, we present the background of IoT context management, main definitions used in the literature, and discuss the similarities and differences between the IoT platforms and CMPs. In Section 2.3, we analyse the main context modelling approaches, which have a direct impact on the storage and processing components of any CMP. In Section 2.4, we analyse architectures of popular IoT platforms and CMPs. Section 2.5 presents the discussion of the approaches to measure and compare the performance of IoT middleware. We also discuss the concept of Service Level Agreements (SLAs) used by cloud IoT platforms as a method to negotiate the balance of performance and cost of provided services. This section bridges the part where we analyse the CMP storage architectures with the analysis of techniques for the adaptive workload optimisation, which is presented in Section 2.6.

Among all self-adaptation techniques, we identified efficient cache management as the most important for the CoaaS platform at its current state of development. We present the analysis of traditional caching approaches and relevant theories for managing the cache in fixed-size systems in Section 2.7. Then, in Section 2.8, we analyse how the traditional approaches should be changed in order to fit the concept of the elastically scalable cloud-based systems. In Section 2.9 we analyse existing probabilistic techniques and strategies, which can be used by CMPs to plan the retrieval and caching of context information efficiently. Section 2.10 presents another group of methods, which are helpful in planning the task allocation, and section 2.11 concludes the chapter.

2.2 IOT CONTEXT MANAGEMENT BACKGROUND

In this section, we present our vision on the IoT evolution, along with the discussion of how context awareness is playing a crucial role in this evolution. We also discuss the definitions of context and our view on the problem. Next, we highlight the similarities and the differences between the IoT platforms and Context Management Platforms.

2.2.1 INTERNET OF THINGS EVOLUTION

As we mentioned in Chapter 1, we can distinguish three main phases of IoT, and we are now entering the third phase [3].

The **M2M phase** (Phase 1) refers to the stage when physical machines were upgraded with electronic communication technologies and could exchange the data with other machines in order to achieve a higher level of automation. However, the integration process required manual programming and tuning. This phase includes SCADA, HVAC, RFID, remote control, traffic control, and other industrial systems. Ordinary people had minimal direct contact with that kind of technologies; most of the settings were done by professionals manually. Communication of devices was mostly based on specialised proprietary protocols.

The **IoT silo phase** (Phase 2) started when commercial companies realised the potential of services, which connected devices can bring to the mass market. The leading enabler of this market was the appearance of smartphones and wearable devices. Mostly, the communication technologies used in the devices of this era were based on the TCP/IP and Bluetooth protocols.

During the silo phase, IoT technologies were instantaneously integrated into the production cycle of companies that worked in such well-established areas as transportation, construction, agriculture, and manufacturing. Investments, research and start-ups in the IoT area produced significant progress in embedded sensors, sensor data management systems, and applications, which used this data.

Probably, the first successful example of that era was the appearance of navigation systems, which were able to gather information about the traffic from their own consumers. Such systems are a perfect example of how sharing data between devices belonging to unrelated people can bring advantages in efficiency optimisation for each of them. However, the dissemination of all the data was controlled by companies which provided these services. No one else could access the data; that kind of systems represented a typical silo, i.e. a vertically integrated system with no horizontal connections.

Then, the operators of the transportation infrastructure started online publishing of data about planned road closures, planned changes in timetables, and similar plans. Developers of navigation systems began to integrate this information to enhance the quality of the navigation [11]. We can see these data as *contextual*, as it is coming from related sources, which are not controlled by the developers of the navigation system. Such integrations formed the first horizontal connections and could be seen as a paradigm shift, bringing us to the next IoT phase.

The IoT ecosystems phase (Phase 3) refers to the stage, when smart devices can communicate with each other spontaneously, without being locked into a silo of a company, which produced the device or the software. This stage is characterised by a massive number of horizontal connections; devices and vertical silos form an ecosystem. However, the progress towards the "true" interoperability-enabled IoT, where devices can seamlessly exchange data with devices belonging to another IoT silo, is still in the early stage [12]. We are just entering the ecosystem phase.

There are several active directions of research, which include the creation of semantic standards and query languages for data annotation and retrieval, development of marketplaces for finding the right context sources and services, and development of gateways for storing and sharing sensor data. On the networking side, the IoT research focuses on such technologies as low-power wide-area networks (e.g. LoRaWAN [13]), Dedicated Short Range Communication (DSRC [14]), and 5G mobile networks [15].

In general, the horizontal communication between IoT devices can be organised in two ways: (i) peer-to-peer (P2P), and (ii) through middleware [1]. The P2P communication has its own benefits; we can compare it with visiting a known website via a browser. However, finding any unknown site requires using a search engine, which is, to some extent, a middleware system. In theory, development of a fully distributed P2P IoT network (e.g. blockchain-based) is possible; however, in practice, the physical limit in network bandwidth, data storage, and data processing capabilities of individual devices are not allowing the creation of such a system in the foreseeable future.

The '*middleware way*' removes the aforementioned physical limits. Moreover, it also has the potential to solve the problem of privacy concerns, which is one of the most widely discussed issues in the IoT area [16]. With P2P communication, each device will have to request data from individual devices to build an aggregation (e.g. traffic map); consequently, every individual can be tracked by unknown requesters. On the contrary, middleware can take

the role of building the aggregations, (e.g. navigation system optimises routes based on GPS tracks of users), thus protecting individuals from direct access to sensitive data. At the same time, the middleware should not forbid the P2P communication between devices in cases when the provider allows the direct discovery and querying. The modern state of IoT and CMP middleware was inspired by and emerged from the earlier research is such areas as pervasive computing, mobile computing and context awareness. A survey of pervasive computing middleware, which illustrates the details, models and ideas that were widely adopted in the earlier works can be found in [17], [1].In Chapter 1, we pointed out how a class of middleware platforms called Context Management Platforms (CMPs) can help to facilitate the horizontal context exchange in IoT ecosystems. We also highlighted that while having many similarities with IoT platforms, CMPs have their specific requirements, as the business model of such a platform differs from a platform which is an enabler for operation an IoT silo. Typically, IoT platforms are offered as software for on-premises installation or Platform-as-a-Service (PaaS). However, for the enabler of horizontal integrations in IoT ecosystems, the term *'Context-as-a-Service'* is widely used nowadays.

Wagner et al. [18] analysed requirements for the Context-as-a-Service middleware platform. The defined requirements related to the storage part of the platform are (i) possibility to exchange context information that is heterogeneous and (ii) consumption of resources used by context services should be minimised. Hong et al. [19] advocated advantages of an infrastructure approach to context aware computing which include (i) system interoperability, (ii) loose coupling and independence of systems and (iii) simpler mobile devices with less power consumption. Authors also declared five high-level challenges for context-aware infrastructure which are (i) simple but expressive data formats for context data representation, (ii) building discovery services, (iii) finding balance between smart infrastructure and smart devices, dividing their responsibilities, (iv) defining scopes for dealing with security and privacy of data and (v) building scalable infrastructures for dealing with a large number of sensors and devices.

2.2.2 CONTEXT: DEFINITIONS AND DISCUSSION

The term "context" is well studied in literature and has a number of definitions. In this paper we will use the definition provided by Dey in [20]: "Context is any information that can be used to characterise the situation of an entity. An entity is a person, place, or object that is considered relevant to the interaction between a user and an application, including the user and

applications themselves." Dey also provides a definition for context-aware computing: "A system is context-aware if it uses context to provide relevant information and/or services to the user, where relevancy depends on the user's task".

Bazire et al. stated that "it was not possible to develop in isolation a model of context because context, knowledge and reasoning are strongly intertwined" [21]. Brezillon et al. also suggested that "the notion of context can take on different meanings, depending on, well. . . the context" [22]. Dourish stated that there is no need in deciding what is context and what is not in general. He defines contextuality as "a relational property that holds between objects or activities" [23].

While the concept of context is still debated, from the middleware perspective context is any data and metadata that can be queried for making decisions about an entity's situation.

There are several dimensions for characterizing context. First of all, context can be classified by, as sensed, (e.g. current GPS position), static (e.g. map of the location), derived (e.g. address of presence) and profiled (slowly changing) [24]. All these types can be used as context by different applications. Secondly, context can be current (e.g. GPS coordinates), historical (set of points representing users track), aggregated and compressed (most common tracks of user represented by critical points only) etc. Historical and aggregated context plays a significant role for any machine-learning algorithms, which are used for making reasoning and predictions. Thirdly, context can be used by different types of applications ranging from one person's needs in managing any smart space, to city authorities needs for making tactical or even strategic decisions about infrastructural management.

We also need to give some clarification here, as the aforementioned classical Dey's definition is not always entirely reflecting our perspective. In our vision, the border between *'data'* and *'context'* is not in the level of processing. While many researchers suggest 'raw data' is data, and 'high-level data' is context, such an approach does not provide total clarity. The often used term '*situation'*, which is commonly used for describing a higher level of context, only tangles the discussion.

To clarify our vision, we looked at the problem by putting the *access to data* in the spotlight, things change. The data which is always directly accessible by the application the own data of the application. It is precisely what happens in IoT silos, and the discussion of context is not really applicable. However, if the access to data is not entirely controlled by the application which wants to use it, then this data can be called contextual. For instance, the data

is requested from a horizontally integrated device, which can grant or not grant access based on its own decision.

In this research, we use examples and use cases, where IoT entities request context from external providers. Consequently, every data item is referred to as a context attribute. At the same time, the IoT entity is still able to put its data in the CMP and then request it back, thus using CMP as an IoT platform. It seems that the overlapping between the notions of context and data is unavoidable.

2.3 CONTEXT MODELLING APPROACHES

In this section, we analyse the existing context representation techniques. We included only those techniques in our scope, which are supported by relatively common storage and processing technologies.

2.3.1 POPULAR STORAGE APPROACHES AND CORRESPONDING CONTEXT MODELLING

Key-Value is a popular NoSQL storage technique that represents any information with a key association and retrieves it by the given key effortlessly and quickly. Key-value modelling is the fastest, easiest and noticeably scalable way of retrieving information from storage. However, standards, schema, verification and relations between entities are not offered. The most important point from our perspective is the absence of means for searching inside values, making it possible to request data only by key. The key-value is mentioned as a context modelling technique in many surveys (e.g. [25], [26]).

Document-oriented or Mark-up scheme tagged encoding is another NoSQL technique, and at the same time one of the most popular ways for representing context. Older proposals were based on XML (e.g. ContextML [27], SensorML [28]). One of the pioneering works in adopting the document-based markup to model the description of sensor data sources was made in the Global Sensor Networks (GSN) project, where XML was chosen as the base format [29]–[31]. Later, the extended version of GSN component was used as a part of the OpenIoT project [32].

Then, with the rising popularity of JSON-based storage, the attention also shifted. The document-based approach is still very flexible and scalable, but allows organisation of data in structures, which are grouped into collections. The important point is that there are ways to organise different types of indices over collections, making fast queries possible. Data

denormalisation is a strong and at the same time weak point of this approach. It is fast to retrieve and write, but the data can easily become inconsistent. Furthermore, document-oriented approaches consume more disk space in comparison with the relational approaches due to applying data denormalisation as a main data modelling technique. Maintaining relations between documents is possible, but not all the document storage engines support joins, as the NoSQL concept assumes that this work should be done by higher-level software components. The most widely used document-oriented stores are MongoDB [33] and CouchDB. JSON-LD fits naturally with MongoDB document model.

Relational database is another way of context storage. Relational database management systems (RDBMS) technology is one of the most well-established technologies and have been used as a main approach for data management for more than 40 years. Allowing an excellent level of stability, functional richness, knowledge base and other benefits, the relational model has a serious disadvantage for modelling context – it has a rigid schema that makes it hard to store any information that is not structured in the way that is defined by relational schema. Another problem is the expensiveness of joins between tables. The most well-known open-source relational databases are PostgreSQL and MySQL. One of the early context modelling approaches based on the relational model is ORM [24].

Ontology-Based Modelling is a way of organizing context into ontologies using semantic technologies like RDF or OWL. A large number of development tools, reasoners, standards and storage engines [34] are available. Ontologies give capabilities for defining entities and expressing relations between them. However, when dealing with Big Data, retrieval of context can be resource consuming and issues with scalability may arise. Besides, ontologies are not recommended for representing streams of sensor data. Examples of RDF storage engines are Jena2, Sesame, AllegroGraph, Virtuoso, etc. [34] Most popular serialization formats are Turtle, N-Triples, N-Quads, N3, RDF/XML and JSON-LD.

Graph-based modelling is a natural way of representing entities and interconnections between them. They are ideal for representing unstructured information and information that has ambiguity. Graphs are typeless and, schemaless, and there are no constraints on relations. This structure is ideal for representing social networks and is recommended for read-mostly requirements. Graph databases have a lot in common with RDF storages but use different languages for querying data. Some graph databases can be used as RDF storages with special plugins applied. Graph databases is a rapidly developing field; the popularity of graph databases has increased by 500% within 2014-2015 years period [35]. Most popular graph

Databases are Neo4J, Titan and OrientDB. However, the context modelling community is usually preferring the ontology-based approach over the graph-based approach, due to such reasons as existing standard semantic vocabularies and embedded reasoning features.

Object-Based Modelling. Numerous projects focus on common object-oriented programming languages techniques for modelling context [36],[37]. These projects deliver huge theoretic base and numerous advanced features for context processing, without focusing on the persistence problem that makes them hard to use in a large-scale environment. Though numerous attempts were taken to develop object storage, the industry standard is still mapping objects to a relational database schema. This is usually done manually, or with a special object/relational framework facilitating the automatic process of mapping entities and hiding the persistence level under ORM abstractions [38]. The main problem of this approach is called object-relational impedance mismatch [39], which represents a set of difficulties while transferring data from object model, with polymorphism, inheritance and encapsulation, to the denormalised table-based database approach.

Based on the discussion above, our research of context representation approaches is summarised by providing quantitative analysis in Table 2.1. In [40] we have identified main requirements to the CMP storage. The comprehensive discussion of the CSMS requirements is presented in Chapter 3. We use the following designations: Disk-based (D); Relations (R); Veracity (C); Geospatial data indexing (GSI); Storage of Sensory Data (SD); Schemaless/Structural data freedom (SL); Horizontal Scalability (HS); Fast Writes (FW); Strong/native support (++); Supported (+); Limited support (+/-); Not supported (-).

Table 2.1. Summary of context representation approaches and their intersections with CMP storage requirements

	D	R	V	GSI	SD	SL	HS	FW
Relational	+	+	-	+	+/-	-	-	+/-
Ontology	+	+	+	-	-	+	-	-
Key-Value	+	-	+	-	+/-	++	++	++
Document	+	+/-	+	+	++	++	++	+
Wide-Column	+	-	+	+	+	+	++	+
Graph	+	++	+	+	-	++	+/-	-
Object	-	+	-	-	-	++	-	+

As it can be seen from the table, the document-based approach has support or strong support for all the identified requirements, except the limited support for relations. All the other approaches only support one or several requirements and have limited capabilities.

According to the analysis of context representation and storage techniques, we identify the document-oriented approach as the most suitable for our purpose.

2.3.2 RECENT TRENDS IN CONTEXT QUERYING AND EXCHANGE FORMATS

The discussion of context exchange platforms had its rebirth with the wave of IoT. There exists a considerable body of knowledge in the area of context modelling, which was created before the IoT era. There also exists several Context Query Languages (CQLs). However, due to different reasons, these approaches did not grow into widely accepted standards [41]; moreover, not all of these proposals are applicable to IoT-generated context. [42]

Many of the existing works are based on Semantic Web concepts as the means for context modelling and querying. There exists a significant number of *semantic vocabularies*, designed to provide a model for context modelling in certain areas. For instance, in the field of sensor data exchange, the Semantic Sensor Network (SSN) ontology [43] is mentioned the most often in related literature. Probably, the most influential general-purpose semantic vocabulary is the schema.org [44]. This community was founded with the collaboration of such companies as Google, Yandex, Yahoo and Microsoft. Recently, the schema.org community started the development of an extension for the support of IoT data, which is called IoT.schema.org [45].

In the bIoTope project, several vocabularies were created to support the use cases. For instance, the MobiVoc semantic vocabulary [46] was developed for the parking and electric charging scenarios. Another example is the Waste Management vocabulary [47], designed to support the context exchange in solid waste collection and bottle bank management use cases.

At the same time, there exist attempts to create context management middleware based on the principles of document-based modelling. For example, the FIWARE Orion [5] context broker is using the JSON (Java Script Object Notation) as data exchange and storage format. The FIWARE's NGSI [48] language is built on top of JSON to enable data transfer and querying. The reasons for this choice are the simplicity of JSON integration with modern web and application development frameworks. Moreover, JSON is natively supported by several NoSQL databases. The latter opened possibilities for high-performance data ingestion and querying, which were hard to achieve with a semantic graph approach.

Another attempt for modelling IoT context was made by the The Open Group (TOG) [49], where XML (Extensible Markup Language) is used as the base. The TOG's OMI/ODF

standard [50],[51] defines the messaging and document format. In general, both the NGSI and ODF are examples of document-based markup modelling approaches.

As an approach to join the world of Semantic Web and JSON, the JSON-LD initiative was proposed [52]. Later, as part of ETSI efforts to standardise the context information management [53],[4], FIWARE proposed the modification of NGSI, which was based on JSON-LD approach. The result is called NGSI-LD [54]. Then, Fiware started the development of the NGSI-LD –based schemas [55] for smart city scenarios. Similar decisions were made by the BIG IoT [56] and simbIoTe [57] projects, which are also a part of the EU FP7 [58] family.

At the same time, the modelling and querying approaches have a direct impact on the design of data storage and processing systems. A mismatch between the concepts of modelling and storage may cause a significant overhead. Accordingly, the design of the CSMS is influenced by context modelling and querying decisions. However, existing data storage and processing technologies can also influence the design of context models.

2.3.3 SITUATION MODELLING AND REASONING

The notion of *situation reasoning* is often discussed together with the concepts of context modelling. Usually, by *situation,* we mean a context of higher level, that was derived from the raw context by applying specific algorithms. The important point is that often situation reasoning involves the handling of uncertainty. There exists a number of works in the area [1]; however the analysis of these works lies beyond the scope of current research.

We would like to highlight the Context Spaces Theory (CST) [59], as it was chosen as the first step for embedding the situation reasoning in CoaaS platform, as well as the CST concepts were used for situation modelling in CDQL [60]. CST uses geometric metaphors for representing context attributes and building multidimensional spaces. Special context situations algebra is used for situation detection and prediction.

The visualisation of a situation subspace and context-situation pyramid [61] in CST is presented in Figure 2.1. CST proposes steps to a generic framework for context-aware applications, and provides a model and concepts for context description and operations over context. This theory is implemented in two frameworks ECORA [37] and ECSTRA [36] and has been extended in Fuzzy Situation Inference (FSI) [62] for situation modelling and reasoning under uncertainty, and other advanced reasoning capabilities.

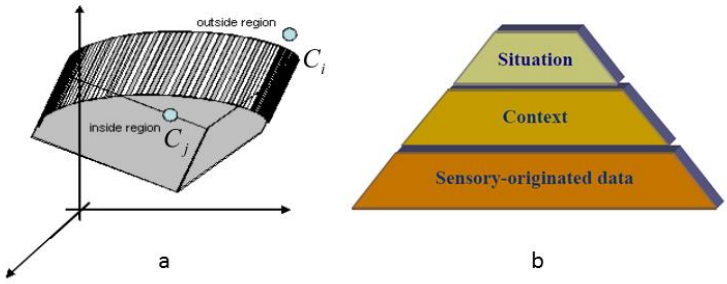


Figure 2.1 - Visualization of situation subspace in Context Spaces theory (a) and Context-Situation Pyramid (b) [59]

The main notion in CST is the concept of situations. The CST model represents situations as geometrical objects in multidimensional space [63] . Such a geometrical object is called a s*ituation space*. A *situation space* is a tuple of regions of attribute values related to a situation. Each region is a set of accepted values for an attribute based on a pre-defined predicate. For instance, in [60] we used a 'Good for Walking' situation as an example of a CST-based function. This situation indicates if the walking path from the suggested carpark location to driver's destination is good for walking, or not. This situation space can be characterized using several context attributes such as temperature, rain intensity, snow intensity, time of the day, safety of the area, health status of a driver, age, etc. Further, the acceptable regions of values for each context attribute should be defined, e.g., the lower and upper bounds of temperature. This forms a basic layer of situation modelling, which can be seen as rule-based reasoning.

In addition to basic concepts and techniques for situation modelling and reasoning, the CST model provides heuristics developed specifically for addressing context-awareness under uncertainty. These heuristics are integrated into reasoning techniques to compute the confidence level of the occurrence of a situation [64]. One of the main heuristics of the CST model is considering individual significance (*weight*) of each attribute. Weights are values from 0 to 1 assigned to every context attribute, and they represent the importance of each attribute in a situation, with a total sum of 1 per situation. In a simplified version of the example, only considering temperature, rain intensity, and safety of the area, the values 0.2, 0.2, and 0.6 can be assigned to these attributes respectively.

Moreover, CST assigns a *belief* value to each region that indicates its level of confidence in the occurrence of the situation. The confidence in the occurrence of a whole situation is defined as $Confidence = \sum_{i=1}^{n} w_i * b_i$, *w*here w_i represents the weight of a particular context attribute and b_i stands for the belief of the range to which the attribute's i value belongs to.

2.4 CONTEXT STORAGE AND PROCESSING ARCHITECTURES IN IOT AND CMP PLATFORMS

In this section, we focus on discussing the organizational structure of the ingestion and storage components of IoT and CMP platforms. We analyse how the architectural components and differences influence the overall functionality and performance of platforms.

Performance of a real IoT system's storage component depends not only on the underlying storage technology but also on the ingestion pipeline, messaging queue and other components of the platform which take part in data processing or storage. Decisions about how to store, cache and access different types of data also have significant influence on the overall performance.

In this section, we analyse how different IoT platforms deal with data management issues. We have grouped the analysed platforms into two categories: commercial IoT platforms and academy/research projects. This grouping was done due to seriously different approaches followed by these two worlds. Difference in targets and approaches makes it hard to compare projects from different groups, but we believe that best practices from one group can be applied in another for raising the functionality and performance to the next level.

A corner-stone of all commercial solutions is in avoiding technologies that were not seriously tested in real work, focusing on security, performance and cost reduction. These solutions are often cloud-based and use SaaS or PaaS model. Their main target is to allow customers straightforward and rapid development of applications that will connect companies' "things" together. These solutions provide interoperability in terms of various sensors and protocols, but the problem of horizontal data exchange between companies, state organizations, and individuals is not brought into question. Research projects, on the other hand, provide open-source software that can be hosted and maintained by organizations themselves. These projects often focus on semantic and horizontal interoperability, and while providing cutting edge functionality in some aspects can lack functionally or performance in others.

Some cloud solutions do not fully disclose their underlying architecture, limiting the scope of our research in some aspects.

2.4.1 COMMERCIAL IOT PLATFORMS

Predix [65] is a PaaS IoT platform developed by General Electric and aiming to provide services in data collection and processing in the area of Industrial Internet of Things. Predix

has a catalogue of provided services that includes a set of tools for data management. Predix does not produce a one-fits-all solution and propose only the use of one or several services that are best suited for the current task. These services are (i) Asset Data, (ii) Time Series, (iii) SQL database, (iv) Blobstore, and (v) Key-Value store. Communication of components is organized by Message Queue based on AMQP. Asset data is a set of models that are used to describe machines and instances that are created based on these models. Time Series service provides means for efficient ingestion, distribution and storage of sensory data, including indexing for enabling fast querying. The SQL database service is built on top of well-known open-source PostgreSQL database. Blobstore service provides means for storing and retrieving any amounts of binary data and ensures high availability and horizontal scalability. Key-Value store service is built on top of open-source Redis project and serves as an advanced cache store. Predix uses a hybrid (or polyglot persistence) storage solution, but all the responsibilities for choosing the right options are left to the application developers.

Data service is a promising feature of the platform, which is in the beta stage; only two services are available: (i) Places data services and (i) Seismic data services. This is a remarkable step to horizontal IoT solutions. The platform provides easily accessible data from external data sources or sensors, to application developers, making it possible to adapt industrial automation solutions to detected earthquakes or other circumstances.

Predix uses a graph database for its asset service to store data as RDF triples. A special Graph Expression Language (GEL) is used for data retrieval [66].

Tibbo Aggregate [67] is an example of a commercial non-cloud IoT platform. All data is logically separated into two groups: (i) configuration and (ii) events. This approach helps in providing flexibility of data storage in the case when new business objects are added.

Configuration data can be stored in almost any enterprise-grade relational database that supports JDBC connectivity, key-value database or in a file-based storage. In case of a relational database, the AggreGate platform includes an embedded database or a preconfigured version of MySQL. AggreGate provides the means for database clustering for achieving high availability.

Key-value integrated storage is recommended for scenarios which need clustering together with high update rate. *File-based storage* can be used in environments with limited resources. *Event data* can be stored in relational database, NoSQL database or in-memory storage. RDBMS puts some limits on insertion performance. NoSQL database provides

horizontal scalability, high insertion rates and failover functionality. Approximate estimated insertion rates for a relational database are about 500-2000 events per second and 10-20 thousand events per second for a NoSQL database [68]. Aggregate also provides functionality for building a failover cluster and achieving high availability.

ThingWorx [69] is a cloud-based platform enabling developers to build solutions for IoT. It provides three main ways for data storage: (i) Data Tables, (ii) Streams and (iii) Value streams. ThingWorx also uses concepts of an InfoTable and a DataShape. The InfoTable is a JSON document in which all the objects share the same properties. InfoTables are fast in-memory objects and are recommended for storing temporary data. The DataShape specifies what property names are required in an object and what types they have. This means a DataShape represents a schema for defining a "thing". The concept of a DataTable in ThingWorx is similar to a table in relational databases, but columns are defined by a DataShape. A DataTable supports the creation of indexes on its properties. It is recommended to build an index for each common request for achieving high performance, and to use DataTables when it is expected to have not more than 100000 rows in it. Storage of time series data is facilitated by streams. A stream consists of a timestamp and additional properties defined by a DataShape. For dealing with things-driven models, it is recommended to use Value Streams, which have some differences with ordinary Streams. Value Streams provide persistence for associated property and return only property values on request. On the contrary, an ordinary stream returns a whole row when querying a single column.

Amazon AWS IoT platform is a cloud based platform that makes use of all the impressive technological stack provided by Amazon. Communication between devices and cloud is organized by a Device Gateway which supports the publish/subscribe approach. Configuring rules for filtering and transforming incoming events is done by a Rule Engine. This configuration includes routing of data to various supported databases, messaging queues, AWS Lambda and other services. Registration and monitoring connected devices is done in a Device Registry. Configuration of processing rules for the device is done in a JSON document consisting of an SQL statement and an action list.

Amazon's IoT solution uses storage solutions provided by Amazon Storage Services. Full description of its capabilities is not possible in this paper due to space limitations. Amazon Storage Services focus on providing scalability, availability and elasticity for mostly well-known storage technologies and promote a so-called NoDBA approach which reduces

operational costs for customers. The variety of provided storage services includes Amazon DynamoDB, Amazon RDS, ElastiCache and ElasticSearch.

Amazon DynamoDB is a cloud managed NoSQL key-value store, but a version for self-hosted installation is also available. Amazon Relational Database Service (Amazon RDS) can use any of the six most popular relational databases. This set of databases incudes a cloud database Amazon Aurora, open-source databases like PostgreSQL and MariaDB, and commercial solutions like Oracle DB and MS SQL server.

Amazon's in-memory data store cloud service is represented by so-called "ElastiCache". This service can significantly improve system performance by reducing the number of slow disk reads. ElastiCache is based on two popular open-source in-memory engines: Redis as an in-memory data store and Memcached as a system for object caching. For such use cases as device-log analysis and real-time monitoring of applications, Amazon recommends the ElasticSearch service that is based on a famous cognominal search engine. Amazon IoT platform uses a messaging system based on Kafka-based named "AWS Kinesis" for event-broadcasting. Capturing and loading streaming data is performed by Kinesis Firehose and analytical processing of streaming data is performed by Kinesis Analytics [70]. Amazon AWS IoT introduces the "thing shadow" or "device shadow" concept. A special "Thing Shadows" service is responsible for managing fast and easy access to a JSON document, with a current state of device that was last reported to the platform.

IBM Watson IoT solution relies on the IBM Cloudant database. It is a cloud fully managed document-oriented database sharing many common features with Apache CouchDB. IBM recognizes the need for flexible storage solutions, but currenly their solution is mostly document-oriented. Describing plans for the future, IBM's specialists state that variety of tasks causes different requirements to latency, scalability, cost and performance, causing the need for different storage solutions [71]. By now, data from devices can be stored in two formats. If the API receives a valid JSON, it is stored in the same way. In the other case, the data is saved as a base64 encoded string inside the payload field of a JSON document.

2.4.2 OPEN SOURCE IOT AND CMP SOLUTIONS

The **FIWARE** community's aim is to create an open ecosystem that will enable development of Smart Applications. This ecosystem is based on royalty-free standards and covers a wide range of tasks. Software for different category of tasks is grouped into modules, which are called "*generic enabler*s" [72].

FIWARE provides several generic enablers for dealing with various types of storage. The central module of FIWARE ecosystem is the Orion Context Broker. This component uses a connector called “Cygnus” that is responsible for persisting or retrieving data from a specific storage. Current release of Cygnus can communicate with HDFS, MySQL, PostgreSQL, CKAN, MongoDB, Comet, Kafka, DynamoDB and CartoDB.

Time series data in FIWARE ecosystem is managed by a component called Comet or Short Term History (STH). This component deals with storage, retrieval and removal of raw time series data as well as aggregated context information. This component relies on MongoDB as the datastore.

Semantic Application Support (SAS) GE provides a possibility for developing applications based on Semantic-web technological stack. In [73], Ramparany et. al. suggest that OWL and other Semantic technologies can help in solving such problems as (i) Semantic data interoperability, (ii) data integration and abstraction, (iii) data discovery, and (iv) reasoning. FIWARE developers admit that despite massive investments and development of mark-up and query languages, the progress with penetration of Semantic web technologies into the market is still too slow. They identify several reasons which include both technical, engineering and commercial problems. SAS GE tries to solve technical and engineering problems which are (i) scalability, (ii) performance, (iii) distribution (iv) security, (v) lack of methodologies and best practices, and (iv) lack of development instruments. The GE consists of a GUI client and server-side components which are responsible for storing and managing ontologies. Server-side components provide scalable and secure ways to publish and retrieve metadata, as well as instruments for managing the infrastructure and data.

The data layer of SAS GE consists of a relational database which stores information about ontology documents, and a Knowledge Base that supports OWL-2RL. At the moment there is no knowledge base-independent solution and it is implemented as a combination of Sesame and OWLIM [74] .

Object storage generic enabler is based on OpenStack Swift. It provides REST API for storing objects taking care about scalability, high availability, and robustness. Swift is a recommended technology for an efficient, inexpensive and safe way for storing a large amount of data. [75] The Object storage generic enabler extends standard Swift Object Store with special scripts that are executed inside the Object storage when some data is uploaded or

downloaded. These scripts are called "*storlets*". This technique is useful for transforming objects, extracting additional information or analysing the object is some way [76].

OpenIoT [32] is an open source IoT platform, which includes a set of novel functionalities, as it is based on semantic web concepts and uses a triple store.

In OpenIoT the registration, data acquisition and deployment of sensors is managed by X-GSN. X-GSN is an extension of the GSN [29] which is responsible for semantically annotating both sensor data and metadata. Virtual sensor is the main fundamental concept in X-GSN, which is capable of representing any abstract entity (e.g. physical devices) that collects any features. In order to make a virtual sensor accessible from the rest of the OpenIoT platform, each virtual sensor needs to register within the Linked Sensor Middleware (LSM). LSM is another core component in OpenIoT which is responsible for handling the sensor data delivery chain. In this regard, LSM transforms and annotates, (based on the supported ontologies), the data coming from virtual sensors, (through X-GSN), into a Linked Data representation i.e., RDF, and stores it in the database. The OpenIoT platform relies on Openlink Virtuoso (it is also known as Virtuoso Universal Server) as the main database. OpenLink Virtuoso is a hybrid database engine that combines the functionality of a traditional RDBMS, ORDBMS, virtual database, RDF, XML, free-text, web application server and file server functionality in a single system [77]. According to information on the website, Virtuoso can handle the insert rate of 36K triples per second on a single 4-core machine.

2.4.3 MODES OF CMP OPERATION

During the initial research stage and our consultations with the bIoTope project partners, it became apparent that there exists several ways for platform development, based on the approach to managing data flows.

Database Mode - The first way is to put all the data from all the connected devices into the storage and always answer the queries from the platforms storage. As the IoT data is transient, the platforms will have to refresh the data at a high pace, causing high load on the data sources, network, and computation resources. At the same time, serving queries based on the stored data is a more straightforward task, which guarantees lower latencies for the consumer. We call this approach a *database mode*, as all the required data is contained in the internal storage. The difference with the regular database is that the IoT data can potentially be different at the source side when the consumer requests it from the platform. The main

limitation of a CMP working in a database mode is its inability to reach the highest freshness of context.

Redirector Mode - The second way is to retrieve all the data from external sources when the request arrives. This approach guarantees the delivery of the freshest context to the consumer. However, it can also cause significant network and computation load. Moreover, retrieving data from external sources causes delays, which result in a long overall latency of serving the context query for the consumer. We call this approach a *redirector mode,* as all the context queries are always redirected by the platform to providers.

NoD-NoR Mode - Trying to tackle the issues possessed by the database and redirector modes, we concluded that the best balance of performance and cost-efficiency could be obtained only by a platform, which combines the features of both approaches. We called this approach the *NoD-NoR mode (Not only a Database, Not only a Redirector)*.

Most of the CMP and IoT platforms are designed around a "database" or a "redirector" approach. In CoaaS, we have chosen the NOD-NOR strategy, as we believe it is the most cost-efficient way to enable an ecosystem-scale context acquisition.

The effective realisation of a CMP which supports the NoD-NoR mode is a challenging task, requiring significant research efforts. First, the software development requires significantly more efforts due to an increasing complexity of the dataflow. Secondly, the NoD-NoR approach requires efficient self-adaptation mechanisms.

2.4.4 A GAP BETWEEN THE STATE OF THE ART IN DATA STORAGE AND PROCESSING WITH CMP PROTOTYPES

The data storage and processing technologies made a huge step during the last decade. Among the relatively novel achievements, we can mention the rise of NoSQL databases (e.g. MongoDB [33], Apache Cassandra [78], Redis [79]), distributed data storage and analytics frameworks (e.g. Apache Hadoop [80], Apache Spark [81]), message queues (e.g. Apache Kafka [82], RabbitMQ [83], ZeroMQ [84]), full text search engines (e.g. ElasticSearch [85], Sphinx [86]), stream processing frameworks and complex event processing frameworks (e.g. WSO2 CEP [87], EsperTech Esper [88]).

Stream processing systems, continuous querying and complex event processing (CEP) could be seen as separate fields of research. However, we noticed that in recent time most of the frameworks and systems developed in each of these fields are trying to extend their

functionality by borrowing functionalities from each other. For instance, CEP engines are extending the scalability support, while stream processors are adding more advanced CEP functionalities. We can expect that these fields will be further merged in the near future. For that reason in this thesis, we consider use the terms that refer to these fields (i.e. the stream processing and CEP) interchangeably in this thesis.

Moreover, specialised storage solutions aiming at sensory data were created, such as OpenTSDB [89], InfluxDB [90], Logstash [91], and Graphite [92]. On top of that, the traditional relational databases have also evolved and offer new functionalities and scaling options (e.g. TimescaleDB [93] for PostgreSQL [94]). Another area of growth was the graph databases (e.g. Neo4J [95]). We do not consider the proprietary cloud-based solutions (e.g. Amazon IoT platform [96], Google IoT core [97]), as these technologies cannot be deployed on-premises. All these relatively novel solutions have proved themselves in real-world deployments. At the same time, the growth in the area of RDF databases was not inspiring, due to the lack of open, stable and industry-tested solutions.

The first (pre-IoT) generations of context exchange middleware were mostly custom solutions, which were not trying to achieve large scale of data storage or processing. Some projects were based on RDF-based databases. The recent CMP projects are usually based on one or a combination of several mentioned modern technologies.

However, the most crucial problem is still not solved. This problem lies in the lack a way to harness the underlying data storage and processing solutions with a single API, which will be convenient and safe to use by context providers and consumers. This API should take into account such aspects as near real-time querying, situation monitoring and event processing, access control, the need for aggregation and reasoning functions, and, potentially, other functionality. The access control and convenience requirements mean that providing access to underlying storage components using their native query languages is not an option. Such an advanced API and the Context Definition and Query Language (CDQL) [98] are, in general, interchangeable notions. Designing components, which can enable CDQL to use modern data storage and processing technologies is an important research direction in the design of CSMS.

2.4.5 DISCUSSION

After analysing several IoT platform approaches to data storage we identified that mostly such platforms prefer not to limit developers in choice of the data storage format. Some IoT platforms introduce their own storage technologies, others offer well-known open source or

commercial solutions. Mostly these platforms offer the following storage types: (i) in-memory, (ii) document-oriented, (iii) column-oriented, (iv) relational, (v) RDF. Organization of blob storage is done using OpenStack Swift. The scalability and high performance of message queueing is achieved by using technologies like Apache Kafka, RabbitMQ, or ZeroMQ. For Big Data processing IoT platforms usually rely on Apache Hadoop or Apache Spark.

Research prototypes often use RDF or OWL, but this trend is still mostly avoided by commercial companies due to issues with scalability and low industry penetration of Semantic Web technologies. All the discussed platforms leave the decision about how to store incoming data to developers and do not provide means for automated efficient resource allocation.

It is also worth noting that some of the discussed platforms are developing and introducing new features at a very fast pace so that we can expect major changes in the market in the near future.

The discussion of context modelling techniques and architectures of IoT and CMP platforms always lead to a question of comparing the efficiency of existing platforms. We highlight current works in the area of IoT and CMP platforms benchmarking in the next section.

2.5 MEASURING THE PERFORMANCE OF IOT AND CMP PLATFORMS

Comparing the performance of different platforms is a complex task. The complexity grows exponentially with the number of features and possible use-cases. However, performance benchmarks are needed both for product consumers and developers. Product consumers can rely on the benchmarking results for making a better choice and developers can analyse weaknesses of their product, improve and demonstrate the results.

2.5.1 BENCHMARKING

The lack of accepted and appropriate benchmarks is still an open challenge for IoT middleware. To tackle the issue, a renowned non-profit organization Transaction Performance Council (TPC) [99], which has been developing benchmarks for data-centric systems since 1989, started the development of a benchmark for the IoT. In 2018, TPC released a new benchmark called TPCx-IoT [100] with the aim to enable a fair comparison of IoT gateways performance. Development of the benchmark involved contributors from such companies as Red Hat, Intel, Cisco, Huawei, Dell, Microsoft, IBM, Oracle, HPE, and VMWare.

As a base for the dataset generation, TPC used a "realistic dataset based on data from sensors from modern electric power substations" [100]. However, it is arguable that a particular scenario will be representational of other types of load. The main metric which is used in TPCx-IoT benchmark is the IoTps *(performance metric*). The IoTps represents the throughput of a system and is calculated as the number of ingestions performed during a time period (e.g. ingestion/second). The *Price Performance Metric* is also defined as $/IoTps = P/IoTps, where P is the total cost of ownership of the system.

Authors of TPCx-IoT highlight that it is not allowed to tune the system under test to be able to pass the benchmark with better results, especially if such modifications negatively impact other parts of functionality. However, it is not easy to prove if such modifications were introduced, especially in non-open source systems. This issue is one of the most problematic obstacles to overcome in scenario-based benchmarks. Another issue is the inability to compare systems working under load, where queries are significantly different to queries that are used in the TPCx-IoT benchmark. Moreover, the querying part of the benchmark is not covered by the IoTps metric, while for a CMP this part has at least the same importance as the ingestion.

TCPx-IoT is a good starting point for the discussion of IoT Gateways benchmarking. However, it is not very useful for evaluation of CMPs. In other words, the CMPs require more sophisticated benchmarks, which will cover such aspects as high-level queries, analytical queries, subscriptions, and situation monitoring over continuous data streams.

In recent years, several academic papers in the area of IoT platforms evaluation were published [101]–[105]. These works are mainly focused on measuring the performance of IoT platforms in terms of ingestion and not paying enough attention to data retrieval performance. There are only a few papers [100], [105] that took data retrieval performance into account. However, in our opinion, the metrics that are used in these papers are too basic and unlikely to represent the actual performance of IoT platforms.

In the world of transactional databases this effort was started and supported since 1988 by TPC) [99]. Actual benchmarks are TPC-C, TPC-H, TPC-E, TPC-DS, TPC-DI and TPCx-HS, which cover such areas as OLTP, ad-hoc DSS, complex OLTP, complex DSS, data integration and Big Data.

In the NoSQL movement, which has significant differences in approaches with the classical transactional world, the most popular benchmarking approach is the Yahoo Cloud Serving Benchmark (YCSB) [106], which is supported by a number of open-source tools [107].

Semantic web community also introduces a number of benchmarking strategies for RDF Stores (e.g. Berlin SPARQL Benchmark [108]).

Discussion of benchmarking strategies for the IoT platforms has already started [109],[110] and some attempts have been made [111],[112]. For example, in [113] an approach of benchmarking the results of IoT platform deployment in a Smart City is discussed; the EEMBC IoT benchmark focuses on devices and connectivity [114]; HP develops the IoTAbench [115] with initial focus on use-cases like smart metering. The problem is in the variety of vendors understanding of the IoT platforms principles, tasks, main features and system complexity in general.

We have also proposed our approach to CMP benchmarking in [16]. The core of the approach is based on the notion of '*query richness*'. In general, we introduced a set of variables and a set of metrics. The variables are: (i) *hardware capacity, (ii) ingestion rate, (iii) incoming query rate, (iv) pull query richness, (v) push query richness, and (vi) push query number*. The metrics are: *(i) query execution time, (ii) event handling time, (iii) network used, and (iv) transition time between modes* [16]. We also demonstrated the importance of taking high-level queries into account when testing IoT systems, as such queries can provide very significant improvement to the performance.

2.5.2 SERVICE LEVEL AGREEMENTS

To link the discussion of performance measurement with the CMP self-adaptation techniques, we need an instrument for defining what performance can be considered satisfactory for the consumer, and what price the consumer agrees to pay for the proposed performance. These tasks are usually implemented by establishing a Service Level Agreement (SLA) between a context consumer and the middleware platform. In [116], a systematic study of IoT SLA management is presented.

The commercial cloud-based IoT platforms have already established their own SLAs. For instance, we analysed SLAs of Amazon IoT platform [117], EVRYTHNG Platform [118], Microsoft Azure SLA for IoT Hub [119], and Google Cloud IoT Core Service Level Agreement (SLA) [120].

To start, most Platform as a Service (PaaS) providers define the price for the uptime per month (e.g. more than 99% uptime). If this condition is not satisfied, the provider returns a certain percent of the service fee (e.g. 30%) back to the consumer's account.

At the time of analysis, the most well-defined SLA was proposed by Amazon IoT. This platform defines SLAs for connectivity, messaging, device shadow and registry, and the rules engine. An example of an SLA for messaging is presented in Table 2.2: the pricing is per million messages and the pricing depends on usage (plan), and the region where the platform is located.

Table 2.2 - Amazon IoT SLA for messaging [117]

Monthly Message Volume	US East (N.Virginia)	EU (London)	APAC (Sydney)
Up to 1 billion messages	$1.00	$1.20	$1.65
Next 4 billion messages	$0.80	$0.96	$1.32
Over 5 billion messages	$0.70	$0.70	$0.84

A similar approach is applied to other services, every action is counted and a price is charged for a million of operations. Amazon does not define the performance of execution (latency).

Although during recent years commercial IoT platforms achieved a certain level of SLA definition, in the field of CMPs the discussion has just started. The reason for that is, first of all, the lack of real CMPs operating at large scale. Secondly, the complexity of business rules for a CMP is higher, as providers and consumers do not belong to the same owner. Jayaraman et al. [121] discussed the problem of orchestrating QoS in IoT ecosystems. They came to a conclusion that the core of the QoS definition should be built around the quality of IoT data (freshness, coverage) and timeliness of its delivery (latency); and so, these parameters should be linked to the cost of IoT services.

Along with the architecture and realisation of software modules, there is another aspect which can significantly influence the performance of a CMP. As the load comes in a form of queries composed by context consumers, the variety of queries can be massive, and as these queries can change over time, it is impossible to correctly tune the CMP manually. Consequently, self-adaptation becomes a crucial factor. We discuss the techniques which can be employed for workload adaptation in the next section.

2.6 ADAPTIVE WORKLOAD OPTIMIZATION TECHNIQUES

The self-adaptation process can pursue several, often mutually conflicting, objectives: fast data ingestion, fast query response, low consumption of main memory and disk space, low consumption of processing power, network bandwidth and external services. We can roughly divide these aims in two categories: (i) Quality of Service (QoS) optimisation and (ii) Resource optimization. In the QoS category, the latency of query serving plays a critical role. For instance, the link between latency and sales has been studied in the fields of e-commerce and search engines. In 2006, Amazon reported every 100ms of delay resulted in 1% sales decrease. Google showed how a 0.5 sec latency in delivering search results decreased the traffic by 20% [122]. Since then, the user expectations and demands have only grown; latencies in serving user requests are not tolerated anymore.

There is a broad range of research efforts in the area of self-adaptation mechanisms, covering such topics as: (i) saving and reusing results of completed computations (caching), (ii) in-memory caching, (iii) reducing big data, (iv) proactive retrieval of raw data, (v) pre-computation of results, (vi) adaptive indexing, (vii) elastic cluster sizing, (viii) dynamic choice of storage format and data placement.

- A survey on *big data reduction* methods was performed by Rehman [123]. The main research directions in this area are the network theory, big data compression, redundancy elimination, data pre-processing and dimension reduction.

-A significant research was done in the area of *elastic cluster sizing* and resource allocation for reducing the cost of cloud services while satisfying the SLA [124], [125]. Dutreilh et al. propose to use reinforcement learning for efficient automated resource allocation in cloud systems [126].

- *Data and task placement* is a term referring to the adaptive choice of the storage and processing facility. For instance, the decision can be made between factors like using an in-memory or disc-based storage, number of the server nodes containing the data and facilitating parallel access, geographic location of server nodes. Dynamic resource reallocation between HDD and SSD was studied in [127],[128], [129] and [130].

-Approaches for *lowering the load on network* infrastructure and decreasing the network latencies include (i) bringing the computations to the data [131], (ii) data co-location [132], (iii) caching and reusing data from remote providers in the middleware.

-Implementing the query-based data *proactive data retrieval* adaptation is the promising approach. For instance, PRESTO, a feedback-driven data management system for sensor networks, is described in [133]. The main aim of PRESTO is to reduce the number of queries to sensors by using a shared by sensors and central node prediction model. With this approach, the sensor has to push data to the system only in the case when the measurement is different from the prediction and the central node can serve queries without querying remote sensors. Simultaneously, the system automatically adjusts the models based on query dynamics, especially to such parameters as error and precision tolerance.

- In the area of predictive queries based on objects location, a substantial work was performed by Hendawi et. al. In [134] a system called Panda is described, which is able to scale up and support a large number of moving devices, as well as large number of queries.

-*Indexing* involves building an accessory data structure and maintaining data in this structure, to facilitate fast access to the place, where the full document (or row in case of a relational database) is located. Maintaining the right indexing strategy is a hard task that is typically performed by the database administrator [135]. Lack of indices causes full scanning of available data, which consumes lot of time and resources. At the same time, maintaining indices also has a cost, as every newly added, deleted or modified piece of data needs to be added to, (or deleted from), all the corresponding index structures. An over indexed datastore will be performing unacceptably in many cases, and the ingestion process will suffer, in particular.

With semi-structured contextual information, with not predefined and constantly changing queries and workloads, maintaining the right balance manually becomes impractical. There is a variety of works discussing and proposing techniques that will allow indexes to be built based on background analysis of workload. First of all, there is a need to understand the characteristics of the workload. The research in automated workload characterization that was conducted as part of the Cloud-TM project is presented in [136]. Authors stress that effective mechanisms for self-tuning are essential for a cloud system that has to deal with the changing workload. They present a component called Workload Analyser that is responsible for gathering statistical data from cluster nodes, producing workload profiles and generating characteristics of present and predicted needs of the deployed applications.

In [137] authors divide approaches to indexing into four classes, which are: (i) no indexes, with full table scanning, (ii) the traditional offline approach, (iii) online tuning and (iv) adaptive indexing, when the index is built as a side effect of the query execution.

Moreover, authors propose a methodology for benchmarking the effect of adaptive indexing on the performance of the system. Authors examined other approaches to the problem, being soft indexes [138] and online tuning [139], which involve a monitoring phase and an index building phase.

-*Caching* the results of query execution in memory (or even on disk) can sufficiently increase the system performance by reusing these results instead of performing the whole query once again. Brin and Page in their paper [140] name the caching of results as one of the most efficient approaches for increasing the performance of a web search engine. *Caching strategies* are an active research direction in many areas, including the IoT middleware. Caching can be applied to raw data and to the results of query execution, which have the potential to be reused. The research of caching strategies includes such traditional techniques as Least recently Used (LRU) and Least Frequently Used (LFU), cost-function based approaches, where cached items are assigned a score based on the value of an item, and intelligent techniques, where advanced machine learning techniques are applied for decision making.

-*Prefetching (or proactive retrieval)* is a technique often combined with caching to achieve higher efficiency. Prefetching refers to a situation when the middleware retrieves the data from the source before the arrival of the query, which might need the data. Prefetching requires analysis of access patterns (queries), as well as the analysis of the behaviour of data items.

-*Pre-computation of results* is a technique when results are computed before being queried. In [141] the basic principles of SensorDB are discussed. Authors state that while having thousands of data points coming from sensors, the individual results are usually not as important as patterns and correlations. For this reason SensorDB pre-calculates certain features of the data stream using a set of pre-defined aggregation windows. This technique helps to increase scalability, query response time and overall performance of the system.

The same approach is presented in [142]. Authors of SmartFarmNet platform use the micro summarisation approach to calculate statistical features of a data stream for aggregation windows. SmartFarmNet also makes decision about the placement of results based on data

access patterns. For example, frequently accessed data is stored in in-memory database. Related work in the area on location predictions can be also found in [143].

-Discussion of *in-memory databases and In-memory Data Grids (IMDGs)* is very active nowadays. It is a more technological, rather than methodological way to reach efficiency and high performance of a system. IMDGs receive a lot of attention in the IoT community [105]. In a survey of Big Data management and processing technologies [144] Zhang et al. state we are facing a revolution in system design. Access to memory is hundreds times faster in comparison with spinning disk or SSD technologies. This makes this kind of data management system a leader in performance, in comparison with all other competitors. At the same time cost of a system that works with Big Data can become unreasonably high. Di Sanzo et. al. [145] propose machine learning techniques for tuning the performance of IMDGs.

Most of the techniques discussed above rely on sophisticated statistical, probabilistic and machine learning techniques. Many of the developed approaches can be useful for the CoaaS middleware. We have chosen the cache management and prefetching as the central direction of our research. We discuss the relevant work in detail in the next sections.

2.7 TRADITIONAL CACHING APPROACHES

In this section, we analyse the existing traditional approaches to data caching used across various internet middleware projects. While the IoT, and especially CMP middleware is still passing through the infancy stage, the area of web servers has passed this stage decades ago, and the techniques used there can already be seen as traditional. Nevertheless, reviewing these techniques is essential for the understanding of the field, as the area of web caching is related to IoT caching. In web caching related literature, the caching approaches are usually classified into client caching, proxy caching, and server caching. A CMP acts as a proxy and as a server at the same time, as it redirects the contextual data from the provider to the consumer, as well as handles the execution of a query, which can be complex [16].

We can mark the border between the traditional approaches and modern approaches in the following manner: if the approach can be efficiently applied in a system, which can be *'infinitely'* elastically scaled, then we call the approach *modern*. The reason for that is the recent migration of majority of systems deployment towards the cloud, where any software that has the capability to scale horizontally can get almost any amount of resources. The limit with this approach is only the price, which the owners of the system are required to pay to the operator of the cloud.

However, if the system is working in a non-cloud mode, for example on a dedicated server or on an edge device, the computational and storage resources are strictly limited, and the system (or its designers) has to make a choice of a caching strategy based on the fixed size of these resources. These systems were the majority in a recent past, and we call the caching techniques used for these types of systems - *traditional caching approaches*. In the literature, these approaches are also referred to as *cache replacement strategies*, as the decision is usually about deleting (evicting) an object from the cache, when the available resources are close to being exhausted. We can also separate the traditional caching approaches to basic and intelligent. Basic approaches rely on simple algorithms, which can be efficient in certain cases, but cannot adapt to changes in load. Intelligent approaches, on the contrary, are designed to analyse continuously the performance and load in order to tune the cache decisions. We review basic and intelligent approaches in the following sections.

2.7.1 BASIC POLICIES

The most common example of the traditional cache replacement policies are LRU, LFU, SIZE, GD-size, GDSF, and their variations [146]. The main factors, which influence the decision about the eviction of an object from the cache are: (i) the *recency* of access to an object, (ii) the *frequency* of access to an object, (iii) the *size* (in bytes) of an object, (iv) the *cost* of retrieving the object, (v) the *latency* of access to an object, when retrieved from an external source. The LRU (Least Recently Used) algorithm chooses objects for eviction, which were requested least recently. However, while being simple to implement, not considering other influential factors causes a low performance of LRU, when used in web caching [147]. There exists a number of extensions for the LRU algorithm, e.g. LRU-threshold, SB-LRU, HLRU, LRU-hot, and Pitkow/Recker.

LFU (Least Frequently Used) is a frequency-based algorithm, which evicts the objects that were accessed less often. However, LFU suffers from the cache pollution problem; the objects, which were accessed many times long ago, are still kept in the cache and occupying the space. The variations of LFU are such policies as LFU-aging, Window-LFU, HYPER-G, LFU-DA.

The policy called SIZE evicts the objects, which have the largest size to free the storage resources. Obviously, not taking into account the latency and cost of re-obtaining the large objects causes cache pollution and poor performance. The extension of the SIZE policy is the Greedy-Dual-Size (GDS) algorithm [148]. This algorithm assigns a value to each object, by

computing a function over several parameters of the object. These parameters are the cost of retrieving the object *C(p)*, size of the object *S(p)*, and the aging factor *L*. Later, the GDS algorithm was upgraded to take the frequency F(p) of access into account, and, accordingly was called GDSF [149]. An expression for calculating the key is presented below:

$$K(p) = L + F(p) \times \frac{C(p)}{S(p)} \qquad \textbf{(Eq. 2.1)}$$

Technically, the GD-Size and GDSF policies are the basic examples of a bigger group, which is called Function-based policies. Algorithms belonging to this group are available to overcome the disadvantages of LRU, LFU and SIZE –based policies, however, finding the right parameters and weights for these parameters is a complex task.

There also exists a group of policies called Randomized policies, where the choice of an object for eviction is randomised. The examples of such policies are RAND, LRU-C, and HARMONIC. These policies can provide a simple way to clear the space in the cache for new items, however, the performance of such policies is questionable and it is hard to figure out clearly the advantages and disadvantages.

While the above-described group of algorithms can bring benefits in certain scenarios, they cannot guarantee any kind of optimality. In that sense, we can call them heuristic-based methods. These types of algorithms have no self-adaptation and the designer or the administrator of a system has to evaluate and compare the performance of these algorithms in case he/she wants to adopt these policies in the system under consideration.

2.7.2 INTELLIGENT POLICIES

The emergence of machine learning (ML) technologies lead to the appearance of intelligent caching policies, which are using the access logs as training data. Some existing works show the higher efficiency of intelligent approaches compared to basic techniques. For instance, artificial neural networks (ANN) were used for making cache decisions in proxy caches [150], [151]. In these works, objects are assigned a rating, and the objects with the lowest rating are removed. Another approach based on logistic regression technique (LR) was proposed in [152]. The aim of the algorithm is to predict if the particular object will or will not be requested in a defined time frame. Similar aim was pursued in [153], but the back-propagation neural network was used instead of the LR.

There also exist works showing the applicability of genetic algorithms to cache replacement [154].

The main criticism of the usage of ML-based approaches to large caches is that the learning process can take a significant amount of time and computational resources. Moreover, quick adaptation to changes in load is also challenging for such methods.

2.7.3 PERFORMANCE CHARACTERISTICS OF TRADITIONAL CACHING APPROACHES

According to [155],[146],[156] there exist several main measures (metrics) that can be used to analyse the performance of a chosen caching strategy in a certain environment. These metrics are the Hit Rate (HR) and its inverse, the Miss Rate (MR). These metrics are often also referred to as the Hit Ratio and Miss Ratio. Other important metrics are the Byte Hit Rate (BHR) and the Latency Saving rate (LSR).

HR represents the percentage of requests which a system can answer from the cache. Correspondingly, the MR represents the percentage of requests that a system cannot answer from the cache and has to retrieve the requested data from the external source on the fly. However, HR cannot fully represent the performance, as it only shows the percentage of requests, but not the amount of data transfer reduced, or the reduction in wait time for the client. The expressions for calculating the HR and MR are presented below. In these expressions, N is the total number of requests, δ_i equals to 1 if the request was served out of the cache and equals to 0 if the request caused a cache miss [146].

$$HR = \frac{\sum_{i=1}^{N} \delta_i}{N} \quad \textbf{(Eq. 2.2)}$$

$$MR = 1 - HR \quad \textbf{(Eq. 2.3)}$$

BHR represents the percentage of bytes retrieved from the cache to serve requests. LSR represents a ratio of sum of latencies of request serving, based on cached objects, over the sum of all request serving times. Expressions for finding the BHR and the LSR are presented below. In these expressions, b_i represents the size of requested object, and t_i represents the time of serving the request to this object [146].

$$BHR = \frac{\sum_{i=1}^{N} b_i \delta_i}{\sum_{i=1}^{N} b_i} \quad \textbf{(Eq. 2.4)}$$

$$LSR = \frac{\sum_{i=1}^{N} t_i \delta_i}{\sum_{i=1}^{N} t_i} \quad \textbf{(Eq. 2.5)}$$

2.7.4 PREFETCHING

The discussion of caching techniques is often followed by the discussion of prefetching. Caching is always a reactive technique, and the decision is made on which objects that have already been obtained by the server should be kept and reused, and which objects should be evicted. When the system predicts the future requests, the needed data can be retrieved from the sources and pre-processed, and an incoming query can be served with less latency. Mostly, the prefetching approaches are based on analysing the content of objects or history of access to objects. In [157], a double dependency graph (DDG) was used to manage the prefetching decisions. Another popular approach to predict the access to objects is the Markov Model (MM) -based approach. For instance, such an approach was used in [158] and [159]. In order to reach a decent level of prediction precision, higher levels of MM are used. However, such models cause significant complexities and computation load. There are also challenges in taking recent changes in access patterns into account.

Another group of prefetching algorithms is based on the computation of the cost function. There exist works where the prefetching decision is based on the popularity of objects [160], lifetime of objects [161], or a balance of popularity and update rate [162]. There also exists a group of algorithms called Objective-Greedy prefetching [163], which aim to improve a chosen metric, by prefetching such objects, which will maximise the chosen metric.

There are also works employing data mining [164] and clustering-based prefetching [165] for improving the performance. One of these works, for instance, employs the page-rank algorithm for clustering web objects [166].

Surveys of performance efficiency criteria for web prefetching are available in [167], [168] and [146]. Main metrics are: (i) precision, (ii) byte precision, (iii) recall, (iv) byte recall, (v) traffic increase, (vi) latency ratio. *Precision* represents the ratio of hits to the number of objects that were prefetched. *Byte precision* shows the ratio of sizes of objects that were hit and sizes of objects that were prefetched. *Recall* represents the ratio of cache hits to the total number of requests, while *byte recall* represents the same notion measured in bytes. The *traffic increase*

measures the ratio of the network traffic in the case when prefetching is enabled to the case when it is not enabled. The *latency ratio* represents the difference in the time of request serving for a user.

2.7.5 CACHING OF IOT DATA

As it was already mentioned, the IoT data differs from many other data types, as it is transient. In other words, the data is changing over time. Depending on the type of source, changes can happen more or less often. Moreover, the consumer can also be interested in receiving data with a certain level of precision. For instance, the readings of a high-precision sensor can change every millisecond. However, these changes for a fraction of unit of measurement are not always of interest for a consumer. A consumer might be interested in significant changes, for instance, when the value changes for 10%. There might be classes of consumers, which may want to receive the contextual data with low latency and high precision, and classes of consumers who may settle for less precise results and longer latencies, but for a lower price. The most important thing is that by monitoring the behaviour of a data item, we can compute the TTL for a certain level of precision.

There exists a number of works on caching IoT data in Information Centric Networks (ICN) or Named Data Networks (NDN). Originally, NDN was designed to support fast access to immutable objects (e.g. video streaming). Meddeb et al. [169], [170] showed that the caching nodes of ICN can also be used to store the IoT data. They used the classification of IoT traffic proposed by Liu [171], who proposed four main categories, which are (i) continuous, (ii) periodic, (iii) On/Off, and (iv) request-response. In general, these four modes can be represented as only two – periodic and On/Off. By the On/Off mode, authors mean event-driven updates of a sensor value. Then, they propose an algorithm to predict T_{fresh}, the period when the data item is considered fresh. The algorithm is based on the Autoregressive Moving Average (ARMA) model [172], which is a version of the Kalman filter algorithm. Thus, after estimating the T_{fresh} based on a time series, Meddeb et al., proposed the Least Fresh First (LFF) caching strategy for IoT data in ICN. The LFF strategy is based on evicting those values from the caching node, which have the worst freshness.

Al-Turjman et al. [173] proposed the Least Value First (LVF) policy for ICN caching nodes. It is a function-based caching approach, which takes into account the delay of data fetching, popularity and age parameters for making a decision about the cache eviction. The proposed utility function assigns a value to each object. Al-Turjman et al. in general defined a

Delay Model, a Popularity Model, and an Age model. A formula for computing the value of a data item is presented below (Eq. 2.6).

$$Value_{NDO\,i} = \alpha \times D_i' + \beta \times Pop_{NDO\,i} + \gamma \times Drop_{NDO\,i} \quad [173] \quad \textbf{(Eq. 2.6)}$$

In the formula above, *D* represents the delay model, which is calculated based on the latency of access to the data source providing this data item; *Pop* represents the frequency of queries to the data item; *Drop* represents the age of the data item, which is proportional to the TTL. Parameters α, β, and γ are introduced for manual tuning purposes. Authors showed the superiority of the LFF strategy compared to LFU and LRU. However, as the costs are not taken into account in this model, it is not possible to apply it for multiple SLAs, and its application to non-fixed size systems is also questionable.

In this section, we have summarised the main cache management policies for fixed-size systems, where the aim is to clear the limited cache for new objects, but at the same time to provide the best service for the client. In the next section, we are presenting the analysis of publications related to the field of cache management for cloud-based non-fixed size systems.

2.8 CACHING STRATEGIES FOR ELASTICALLY SCALABLE CLOUD-BASED SYSTEMS

As it was mentioned in Section 2.7, the traditional caching techniques aim to find the best decision on which objects to evict from the cache, in order to free the space for other objects. However, this approach is questionable in the age of cloud systems, when the size of space that can be allocated for the cache is potentially infinite. For instance, the paradigm shift for cache management when moving from fixed-size systems to the cloud systems was highlighted in [174], [175] and [176]. Cloud systems usually rely on the pay-per-use model. Eventually, the cache management problem is not limited by the size of storage or processing resources anymore. It is limited by monetary costs, which allow use of cloud resources. Consequently, minimising the cost of using the cloud-based system while keeping a defined level of QoS becomes the main objective in the cloud paradigm.

Le Scouarnec et al. [177] developed a cloud-oriented caching policy for video streaming services. In their model, the frequency of access to a cached data item and the cost of cloud services were the determining factors for cache management decisions. As an example of cloud services, authors used Amazon S3[3] and Amazon EC2[4] for storage and computing

[3] https://aws.amazon.com/s3
[4] https://aws.amazon.com/ec2

correspondingly. In the proposed model, the choice was made between storing the object for the price of storage (S3) or recompute every time it is needed for a price of computing resources (EC2). The authors call the model 'time-based', as it returns the amount of time, during which the object should be kept in the cache.

Three analytical models are proposed. The first is based on the complete knowledge of the future, (time of requests to each item), to provide the lower bound of the possible cost. The second model is a global policy without distinguishing items, and the third model is based on the knowledge of demand for each object individually and independently from other objects. Then the authors show that the second model is able to approach the performance of the ideal cache policy. The distribution of arrivals to an individual object is modelled as a homogeneous Poisson process; the measurements of the arrival rate are performed over a sliding temporal window.

While the proposed models in [177] are valuable for video streaming services, they are not directly suitable for IoT data, as such an essential parameter like freshness is not taken into account. The latencies of access and their influence on the final cost are not taken into account as well.

During several last decades, we got used to the fast growth in the performance of processors, memory and storage. Unfortunately, now we see the signs of reaching the limits of current silicone-based chips. Many recent articles are questioning the Moore's law validity. [178]. As the requirements from the consumers will only increase, we cannot expect to cover the demand by throwing more and more processing power on tasks. Thus, the research in the field of smart, adaptive, energy-efficient solutions will most likely gain more attention in the IoT community.

2.9 PROBABILISTIC APPROACHES TO CACHING AND PREFETCHING

In Section 2.7.5, we concluded that the freshness of cached objects is an essential parameter, which has a direct influence on the efficiency of cache decisions. We also reviewed how SLAs can define the level of freshness (Section 2.5.2) and how the freshness can be predicted (Section 2.7.5). In this section, we review the existing works on the modelling of caches, looking from a perspective of a single object (data item), which is subject to freshness loss over time. Once the freshness level acceptable for servicing queries is defined in the SLA, and the prediction of how long the particular data item is maintaining the defined level of

freshness, for any cached data item, an associated *expiry time* is recorded. In the literature, expiry time is also referred to as the *time-to-live (TTL)* or the expiry period.

In [179], Jung et al. investigated the following problem: *how to predict the hit rate of a single data item, if the statistics of requests to this item and its TTL are known?*

The initial models used by Jung et al. were developed by Cohen et al. in [180] and [181]. They have developed a simple model to predict cache hits in distributed caches for several specific different distributions of inter-request time.

Jung et al. [179] focused on a single cache but provided a model for predicting the hit rate for any arbitrary inter-request time distribution. The important assumption is that a sequence of requests to a particular data item can be represented as a sequence of random variables, which are independently and identically distributed (i.i.d.). This means, that the process can be viewed as a renewal process [182]. Despite the approach being conservative and simple, Jung et al. reported good prediction accuracy.

The strength of the model is that it considers evicting the data item from the cache only when the TTL expires. This makes the model suitable for elastically scalable systems, as it does not limit the number of cached objects in the system. Moreover, objects are treated independently. The modelled environment is presented in Figure 2.2. The process starts at time t = 0, when a cache miss happens. It is assumed that a cache miss causes a retrieval of data from the external system. Until the time t = T, when the retrieved data expires, the requests are served from the cache. *T* represents the TTL. The time between requests is marked as X1, X2, X3 and X4. At time S4, a miss happens again, and the process starts from the beginning.

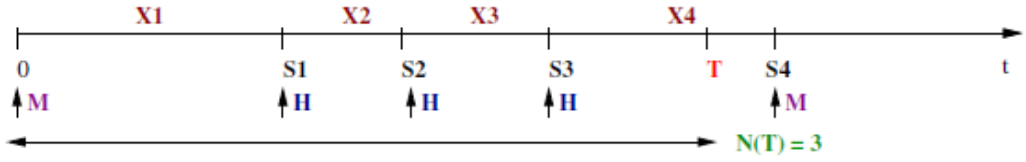


Figure 2.2 - Visualization of request arrivals, cache hits and misses [179]

The main finding of Jung et al. in [179] were the formulas for finding the hit and miss rates based on the expiry period (T) and the expectation of requests arriving during this period E[N(T)]. The formula for hit rate is presented below [179]:

$$HR(T) = \frac{E[N(T)]}{E[N(T)] + 1} \qquad \textbf{(Eq. 2.7)}$$

Based on the fact that MR(T) = 1 – HR(T), the miss rate can be found as follows [179]:

$$MR(T) = \frac{1}{E[N(T)] + 1} \qquad \textbf{(Eq. 2.8)}$$

The formulas above provide a way to predict it and miss rates for reactive caching. These formulas can also be helpful in the case when optimization of the expiry period is needed. However, they cannot be used when prefetching is involved.

Scwefel et al. analysed the caching strategies and ways to establish the adaptive strategy for context middleware in [183]. They have looked at the problem from a different angle. In the proposed scenario, the mismatch probability between the value 'contained' in the source system, and the value 'contained' in the cache of the middleware was in the scope. In general the described scenario is a reactive Quality of Service (QoS) -based strategy. Another detail which is taken into consideration is the latency of transferring the request from the middleware to the source, and the latency of getting the answer back. The scenario is graphically represented in Figure 2.3.

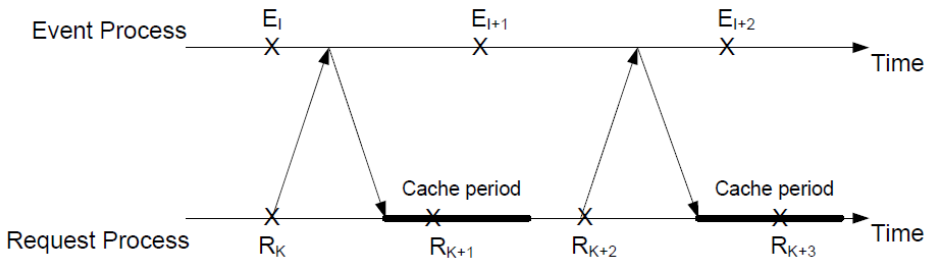


Figure 2.3 – Reactive caching strategy with QoS as the main focus [183]

In the diagram, two processes are presented. The *event process* shows how a data item is changing its value at certain moments in the data source. The *request process* represents arrivals of requests to the middleware system that might contain the item in the cache. In the provided diagram, the request R_k is arriving at the same time as event E_I changed the value in the source. As there is no value in the cache, the middleware retrieves the freshest value from the source, services the query and caches the data item. The query R_{k+1} arrives during the period when the item is considered fresh (bold line). As the request happened before the event E_{k+1}, there is no mismatch in the query result returned to the consumer, and the value contained in the source. Request R_{k+2} is a miss again, and the request R_{k+3} is a cache hit. However, R_{k+3} arrived later than the event E_{I+2} changed the value of the item in the source system. This situation is called a mismatch, which causes the level of QoS. Authors defined three main performance metrics, which are (i) the *mismatch probability* (mmPr), (ii) the *access delay*, and

(iii) the *network overhead*. The mismatch probability is the probability of request to obtain the data item value from the middleware, which already has changed in the source. The access delay is the average latency of request servicing, and the network overhead is the bandwidth consumption to serve the request process. With this approach, the only parameter which the middleware can change is the expiry period (T), as the network delay, (upstream and downstream), as well as the event process, cannot be influenced by the middleware.

In [184], Olsen et al. proved that a mismatch probability for a scenario without caching (T = 0) can be found as follows:

$$mmPr = \frac{\lambda}{\lambda + \upsilon_D} \qquad \textbf{(Eq. 2.9)}$$

In the expression above, the event process, as well as the delay process, are Poisson processes. The event process has the rate λ, and the delay process has the rate υ_D.

In [183], Schwefel et al. upgraded the model to reflect the presence of caching. They have developed models for (i) *calculating the mismatch probability*, (ii) *calculating the mean access delay*, and (iii) *calculating the network overhead*. Based on the provided mathematical models, authors formulated two scenarios for optimisation of the cache period: (i) minimisation of the average access latency and network usage, while keeping the probability of mismatch under a certain threshold, and (ii) minimisation of mismatch probability, while keeping the latency of access and network usage under a defined threshold.

In [185], Bogsted et al. further developed probabilistic models for optimisation of access to data items, which are subject to change. In this paper authors defined models for reactive and proactive strategies. Technically, proactive strategy represents prefetching. Authors define models for two main classes of the proactive strategy, which are (i) *event driven* and (ii) *periodic*.

In the reviewed papers ([186], [184], [185] and [183]), the mismatch is always viewed as absolute. However, in many IoT scenarios, aging of data items can be treated as a decay function. For example, we can consider a linear function of a data item aging. If, for example, a consumer agrees on the particular level of freshness (e.g. 80%), we can straightforwardly calculate the TTL and use the cached data for serving queries of a corresponding SLA.

2.10 OPTIMIZATION OF ALLOCATION OF TASKS AND RESULTS

In the previous section, we looked at the problem of caching from the perspective of making a decision about the eviction or prefetching of an individual data object. At the same time, we can look at the problem from another angle – the allocation of computational tasks and allocation of storage facilities for intermediate results and large chunks of data. For instance, cloud compute nodes have different performance and corresponding costs. Cloud storages also have different time of data access (HDD, SDD, in-memory) and price. If there exists several types of cached data, it is possible to look at the problem of searching for cached results as a bag of tasks and intermediate result allocation.

Malawski et al. [187]–[189] proposed a methodology for cost-optimal task and data allocation for distributed scientific applications. They defined the goals for optimisation, variables, parameters, and constraints. They took into account the cost of cloud compute, storage, and networking services. The delays of transferring the data from one cloud service provider to another was also taken into account. Moreover, the possibility of using limited private resources and possible limits in cloud resources was taken into account. Eventually, based on the defined goals, a Mixed Integer Non-Linear Problem was formulated. This formulation was converted to an AMPL program, and a CBC solver was used to find the global optimum.

The model is called linear program if objectives and constraints are a linear combination of systems' variables. In the usual case, when the linearity assumption is too far from reality, non-linear functions are used and, consequently, the model is called a "non-linear program". Moreover, if a variable can have only integer value, the problem is called "integer programming". From the computational perspective, these type of problems are much harder to solve. The types of possible problems are classified into the following categories: Linear programming (LP), Quadratic programming (QP), Non-linear programming (NLP), Mixed-integer programming (MIP), Mixed-integer non-linear programming (MINLP), Constraint programming (CP).

There exists a significant body of knowledge in the area of applying linear/non-linear programming approach to the problem of finding an optimal solution for balancing complex systems. In [190], Turinsky and Grossman applied convex optimisation techniques to optimise data mining strategies, with the possibility of choosing between centralised and in-place

strategies. Menache and Singh in [191] described applying convex optimisation to online caching to reduce infrastructure cost in cloud environments.

In general, there is a substantial amount of work that use a linear programming approach to address the problem of allocation of intermediate results, including caching, in scientific computing and other related areas.

The linear programming approach applied to costs of various types of cloud compute instances, costs of cloud storage, latencies of access to data, and corresponding SLA conditions can be used by CSMS self-adaptation framework to optimise the amount of purchased cloud resources. This approach can also be applied to the optimisation of allocation of tasks and big slices of data [187], [189].

2.11 CONCLUSION

In this chapter, we reviewed the publications, which are relevant to the field of IoT and CMPs. We concentrated on the context modelling techniques, storage components of IoT middleware, and self-adaptation techniques used for efficiency optimisation. Based on the analysis above, we can state that while the market of IoT systems is quickly growing and evolving, the CMP area is still in its infancy.

We can see that most of the IoT middleware solutions reuse and harness available data storage and processing technologies. Mostly, the used technologies are open source-based or require no license. Another common feature of the used base-level technologies is the orientation for horizontal scalability. For these reasons, NoSQL solutions are often used as the core storage and processing components. However, we have identified the lack of CMP projects with a common, unified and flexible interface, which allows defining, injecting, querying, monitoring, and managing the context. While such an interface was developed as a part of the CoaaS project, the storage system which can facilitate the work of this interface still needs to be designed.

Another common area of research in CMP storage and processing is the self-adaptation mechanisms, strategies, and algorithms. The core of these mechanisms is adaptive cache management, as it is one of the factors, which have the highest impact on the performance of the whole platform. While some basic benchmarking frameworks for IoT platforms are already introduced, there is still no commonly accepted benchmarking framework for the CMP area. At the same time, there is an urgent need for such benchmarks, and the influence of self-adaptation should be taken into account.

An important factor for performance analysis is the ability to define SLAs between CMPs and context consumers. While cloud IoT platforms already use simple SLAs to bill their services, in the area of CMPs, the discussion of SLA definition only begins.

Traditional caching techniques are not applicable to elastically scalable cloud systems. For these reasons, optimization of caching strategy for CMPs required defining costs through cloud resources and SLAs. There exists a number of works in the area of defining task and data allocation, as well as works aiming to predict the hit and miss rates in CMP-originated environments to solve the cache optimisation task. However, there is a research gap for cases where several SLAs are defined, and the arrival of queries is a stochastic process. This problem requires research and development to establish such techniques.

Based on the literature review and the analysis of research gaps, in the next chapters, we proceed to the development of CSMS architecture, CSMS modules design, and investigation of cache management techniques.

Chapter 3: Context Storage Management System requirements and architecture

3.1 INTRODUCTION

After the initial research, analysis of the literature (Chapter 2) and use cases, it became evident that the CoaaS middleware platform cannot be only used to retrieve information from providers, fuse it and send back to customers on request, playing just a role of a redirector or a reasoner. It also need to maintain the quality of performance and meet the agreed quality of service.

First, the need to query numerous sources of information sequentially, and potential unavailability of these sources as well as network latencies can lead to unexpected delays in serving queries. However, consumers expect the platform to meet the agreed quality of service and not being able to conform with this level will lead the platform's operator to financial losses. Therefore, keeping the most critical data at the platform's side could significantly improve the quality of service.

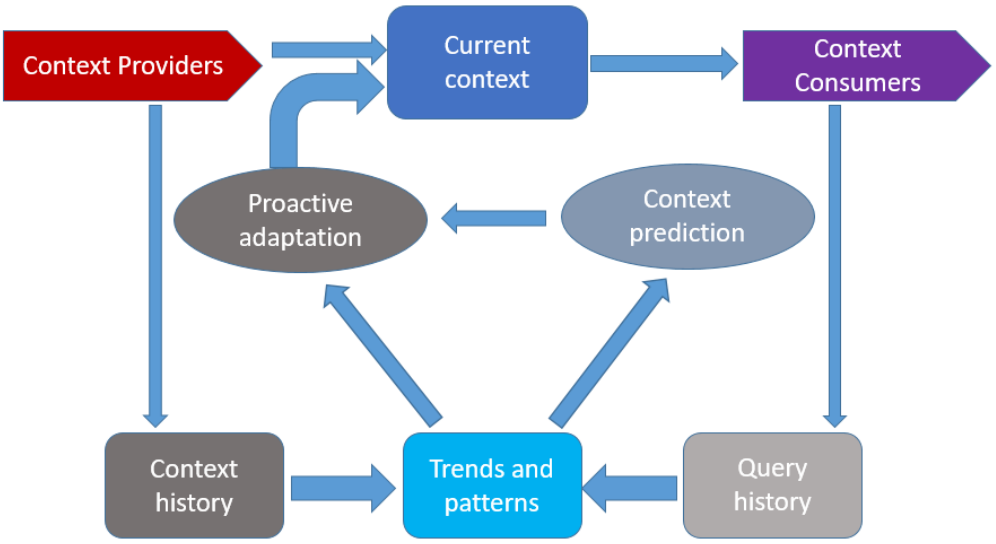


Figure 3.1 - Acquiring current context requires support form deeper levels

Secondly, the process of deriving and managing context depends on the knowledge of the history, trends and patterns of context changes. Moreover, enabling proactive adaptation and caching in CSMS, which we consider as a potential solution to achieve cost efficiency, requires the information about the patterns of incoming queries. Proactive adaptation also depends on context prediction, which in turn depends on the context history. The only way to

acquire the mentioned patterns and trends is by storing the incoming data, (both from the providers and consumers sides), at the platform's side and continuously analysing it.

Schematically this concept is presented in Figure 3.1, which depicts all types of data which are passing through the CSMS or are stored in CSMS. To clarify the figure, we can consider a simple scenario, when a context consumer (the purple arrow) requests the data about available carparks in a certain area. Contextual information about the number of parking places is retrieved from the context providers (the red arrow). At the moment of retrieval this context is fresh (not expired) and can be returned to the consumer. The number of available parking spots for each parking facility at a current time can be saved as a history. As the history dataset keep growing in size, this data can be aggregated and converted into compressed knowledge (trends and patterns). The incoming query is saved in the query history, and the aggregated patterns are derived from this dataset. As a result, the platform has the data, which describes how often a particular service provider has been queried, as well as how fast the value of the measurement (number of available parking spots) has been changing. This data is used for computing the optimal behaviour of the platform (proactive caching). Moreover, this data is also used for enabling the prediction functionality, for instance, to service a query which requests the number of available parking spots in an hour from the current time.

It is also worth noticing, that while the two reasons above (speed of query execution and adaptation) are essential for fast query servicing and advanced functionality, there is also a requirement to store the metadata about context providers. Without this data, CoaaS will not be able to find the sources of context to serve a query. Consequently, the storage of context providers' descriptions is a basic and essential part, without which a CMP cannot function.

The above-mentioned reasons highlight the need to store metadata and context as a core part of the CoaaS middleware platform. This chapter is organised in the following way: in Section 3.2, we discuss the vision of CoaaS, its overall architecture, and its main interface, the CDQL language, as an input for the CSMS project. In Section 3.3, we discuss the requirements which are essential for CSMS to effectively serve as a part of the CoaaS platform. Next, in Section 3.4, we describe the proposed CSMS architecture and its main modules. Section 3.5 concludes the chapter.

3.2 COAAS AND CDQL BACKGROUND[5]

In this section, we introduce the vision of the Context-as-a-Service (CoaaS) platform, including such aspects as its blueprint architecture, principal components and the CDQL query language, which is the primary interface for communicating with the platform.

This section explains the alignment of the CoaaS vision, and especially the unique features, of the platform to the requirements, which the storage system of the platform (i.e. CSMS) must satisfy.

3.2.1 COAAS VISION

The CoaaS platform is designed to enable the context exchange in IoT ecosystems. We have defined the concept of IoT ecosystem in Chapter 2. The big picture of an IoT ecosystem and the role of the CoaaS platform is presented in Figure 3.2. The main aim of designing the platform was in enabling the developers of context-aware applications to query context from external providers seamlessly, flexibly, and cost-efficiently. Thus, the CoaaS platform belongs to the class of Context Management Platforms.

The top layer in Figure 3.2 represents the context-aware IoT applications, which request contextual information from the IoT ecosystem. These applications are called *Context Consumers (CC's).* In Figure 3.2, the bottom layer represents IoT entities, which can provide the contextual information about entities. These sources are referred to as *Context Providers (CP).*

As is defined in Chapter 2, "context is the information that can be used to characterise the situation of an entity" [20]. For instance, we can consider such objects as cars, parking facilities, locations, and persons as *entities*. Each entity can be defined by specific parameters, which are called *context attributes*. For instance, a parking facility can have such attributes as geolocation, opening hours specification, and occupancy of the facility. In general, a context provider can be any system, ranging from a single sensor connected to a wireless network, to a complex backend enterprise-level system. For instance, a simple connected temperature sensor can be considered a context provider. On the other hand, web services which are

[5] Section 3.2 is not presented as a personal contribution of the author. This discussion is needed as a background. At the same time, as the author has been an active member of the CoaaS research group, placing this section in the literature review was considered unfitting.

providing information about weather or traffic can also be considered context providers. However, in the latter case, a service can provide context about multiple locations, which are different *entities* of interest.

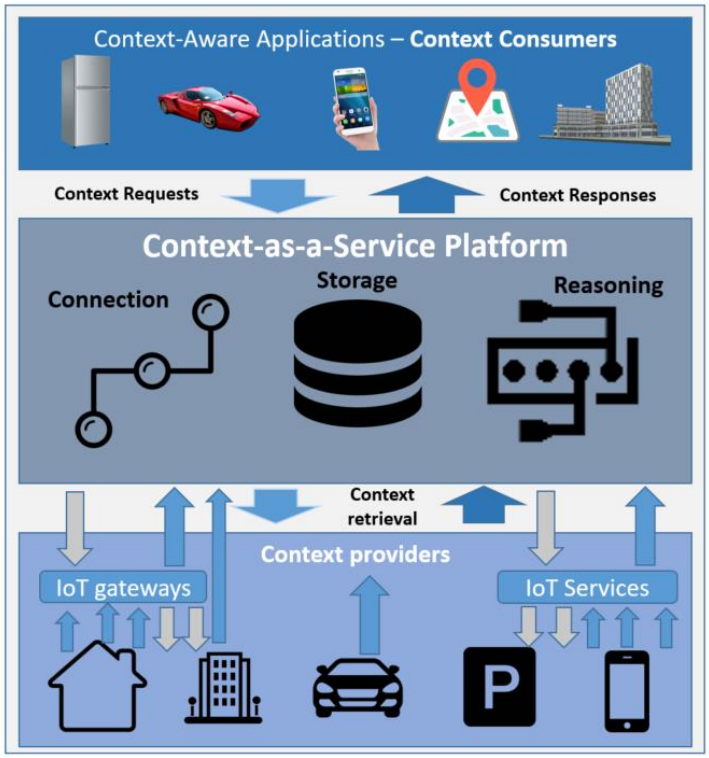


Figure 3.2 - CoaaS platform in IoT ecosystem

Each CP can provide information about one or more entities. For this reason, the notion of *a context service* was introduced. A context service makes available the information about an exact entity. A context service makes available the information about an exact entity. A context service can also be seen as an API to a particular entity. The important point is that the access to this API is controlled by the CP, so the access control restrictions, fees and other features of the as-a-service model can be applied.

It is also essential to notice, that the same device or application can be a context consumer and a context provider; the difference is only in the direction of the dataflow.

In Figure 3.2, the CoaaS platform is represented as a middle layer. CoaaS receives *context queries* (CQ) from context consumers. A context query is a description of information, which should be requested from one or multiple entities. This information can be low-level, (context attributes), or high-level when additional functions are applied to the low-level context.

A context query consists of one or several joined *context requests (CR)*. A context request is a retrieval of information about a certain entity type. CR consists of one entity and zero to many predicates.

There are two ways of communication between the platform and context providers. The platform can retrieve data from the provider when this data is needed, or the provider can push the stream of *context updates* into the platform. A context update is a message, which contains the state of a particular entity (context attributes with values), and the time of measurement. To enable the subscription functionality, which is also called situation monitoring, CoaaS infiltrates the stream of context updates though the registered subscriptions.

The main interface of communication with CoaaS is the Context Definition and Query Language (CDQL), which was developed specifically to comprehend the features of the CoaaS platform. More details about the theoretical foundations of CDQL can be found in [192].

We have discussed the high-level view on the ecosystem and highlighted the main definitions and concepts. In the next subsection, we discuss the blueprint architecture of the CoaaS platform.

3.2.2 COAAS ARCHITECTURE

In this section, we concisely describe the high-level architecture of the CoaaS platform, as well as the principal functionality of the main components. The designed architecture is presented in Figure 3.3.

As it is shown in the figure, the platform comprises four principal components, which are depicted as blue rectangles. These components are (i) Query Engine (QE), (ii) Context Storage Management System (CSMS), (iii) Communication and Security Manager (CSM), and (iv) Context Reasoning Engine (CRE).

The Context Query Engine (QE) is a component responsible for handling the execution of incoming context queries, which are formulated in CDQL language. QE controls the parsing stage, generation of the query execution plan and the assembling of the final result. An important subcomponent of QE is the Invoker, which is responsible for retrieving context information from context providers when there is a demand from internal components of CoaaS.

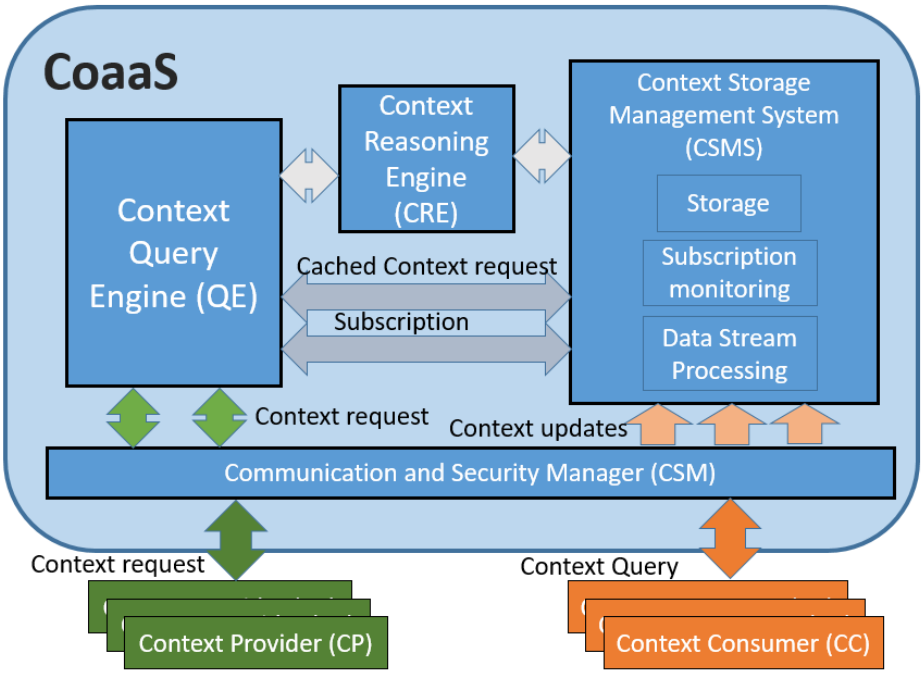


Figure 3.3 - CoaaS Blueprint Architecture

Context Storage Management System (CSMS) manages the process of caching the context information and facilitating the efficient access to the cached data. Besides caching, CSMS provides the service discovery functionality based on the stored information about the known context services. The third important feature of CSMS is subscription module, which is designed to support the continuous monitoring of incoming context, infer situations from available context, detect changes in situations and provide notification of detected changes. This component monitors the real-time context of the IoT entities by percolating the incoming events through all the registered PUSH-based queries. The last objective of CSMS is storing and analysing the historical context to facilitate self-adaptation and efficiency optimisation.

The Communication and Security Manager is a mechanism designed to handle the incoming and outgoing traffic. The important features of CSM are the transferring of messages to the right components, checking the validity of messages and initial authorisation of context requests.

The Context Reasoning Engine is responsible for inferring higher level context from the raw contextual information. In general, CRE can contain a set of reasoning enablers, which are based on different techniques. At the moment, the CRE is represented by a software component called ECSTRA [36], which is capable of estimating the likelihood of the situation based on the Context Spaces Theory.

Table 3.1 contains a brief list of functions for each of the components.

Table 3.1. CoaaS components and their functions

Component	**Functions**
Context Query Engine	(i) Context query parsing (ii) Execution plan generation (iii) Context result assembling (iv) Sensor data acquisition (via Invoker) (v) Context service selection
Context Storage Management System	(i) Current and historical context storage (ii) Context service registration and discovery (iii) Event processing (iv) Subscription monitoring (v) Caching and prefetching strategies (vi) Triggering sensor data acquisition
Communication and Security Manager	(i) Access control (ii) Authorisation (iii) Messaging (iv) Security monitoring (e.g. DDoS)
Context Reasoning Engine	(i) Inferring higher-level based on advanced reasoning techniques (e.g. CST)

Next, we discuss the critical characteristics of the CDQL language.

3.2.3 COAAS INTERFACES – CDQL

Context Definition and Query Language (CDQL) [10], [60], [98] is a flexible and generic context query language that allows IoT applications to publish and query context. Figure 3.4 illustrates the production rule of the CDQL language, which consists of three mandatory and two optional clauses.

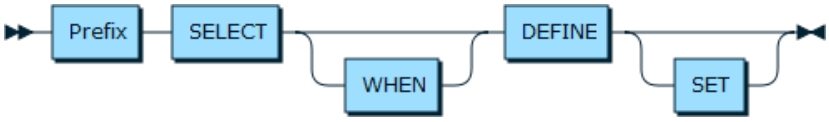


Figure 3.4 - CDQL Production rule [98]

The mandatory clauses are PREFIX, SELECT, and DEFINE.

The PREFIX clause is used for listing the semantic vocabularies, which are adopted to describe the entities in the query.

The SELECT clause is used for identifying the output of the query which can be either low-level context (a set of context attributes), or high-level context.

The DEFINE clause is needed to compose complex queries that include various entities and constraints. This clause defines the entities that are involved in a context query. Figure 3.5 provides an example of a basic CDQL query. This query expresses a request to find available car parks and can be issued by a smart car or the navigation system's backend server.

```
1  prefix schema:http://schema.org, mv:http://schema.mobivoc.org
2  select (parking.*)
3  define
4  entity parking is from mv:ParkingFacility
5  where distance(parking.location, event.location , "walking")
6  < {"value":500, "type": "distance", "unit":"m"} and
7  parking.cost < {"value":5,"unit":"aud"},
8  entity event is from schema:event where
9  events.attendee.email = "person1@test.com"
```

Figure 3.5 - PULL-based CDQL for finding parking

One of the distinguishing features of CDQL is the integrated ability to represent situations in a probabilistic form, is based on the Context Spaces Theory (CST); more details can be found in [63] and [60]. This feature enables the uncertainty handling in context queries. To enable this feature, a specific syntax was designed.

In Figure 3.6, an example of situation representation is provided. This example defines a probabilistic 'goodForWalking' situation-function, which computes the probability of the comfortable walking condition for a particular location. The representation of the situation contains definitions of ranges of values for every attribute. Each attribute is assigned with a weight and, each range is assigned with a belief.

```
prefix schema:http://schema.org
create function weatherSituation is on
schema:weather as r1 {
   "goodForWalking" : {
      r1.airTemperature : {
         ranges : [
            { value:(0;6], belief : 20 } ,
            { value:(6;13], belief : 50 },
            { value:(13;28], belief : 100 } ,
            { value:(28;38], belief : 20 }
         ],
         weight : 10
    } ,
      r1.windSpeed : {
         ranges : [
            { value:(0;8], belief : 100 } ,
            { value:(8;20], belief : 50 },
            { value:(30;40], belief : 10 }
         ] ,
         weight : 5
      }
   }
```

Figure 3.6 - Example of CST-based situation function definition

The syntax for describing situations in CDQL also contains means to represent window-based functionality, trends and temporal relations between events [60].

The second type of a CDQL query is a PUSH-based query, which is designed to enable the subscription for situation monitoring functionality of the CoaaS platform.

```
prefix schema:http://schema.org, mv:http://schema.mobivoc.org
select (altParking.*)
when isFull(selectedParking, car, event) > 0.80
define
entity selectedParking is from mv:ParkingFacility where
selectedParking.id = "parking 1" ,
entity event is from schema:event where
events.attendee.email = "person1@test.com",
entity car is from mv:car where
car.vin = "sample vin",
entity altParking is from mv:ParkingFacility where
(
distance(altParking.location, event.location , "walking")
    < {"value":500, "type": "distance", "unit":"m"}
or (
distance(altParking.location, event.location , "walking")
    < {"value":2, "type": "distance", "unit":"km"}
and goodForWalking(altParking.location.weather) >= 0.80
    )
)
and altParking.cost < {"value":5,"unit":"aud"}
and isFull(altParking, car, event) < 0.80
```

Figure 3.7 - PUSH-based CDQL for finding parking

The code snippet in Figure 3.7 shows an example of a PUSH-based CDQL query. This query will instruct the CoaaS platform to monitor specific parking that a car is driving to. If CoaaS detects that the carpark will be full by the time a car will arrive there, the platform suggests alternative carparks. It is worth mentioning, in order to select a list of alternative car parks, CoaaS takes the distance and walking conditions between the destination and parking into consideration by using the *'goodForWalking'* function.

To support the PUSH-based query functionality, CDQL contains the WHEN and CALLBACK clauses. The WHEN clause allows us to describe the situation, which will be monitored based on the incoming events. When the situation is detected, the subscription is triggered, and the SELECT clause of the corresponding query is executed. The CALLBACK clause defines the format and address of the endpoint, where the result of the query execution will be sent.

On the contrary, a PULL-based query does not contain a WHEN clause, as it is executed only once immediately after the query has been received.

In this section, we provided an overview of CoaaS platform and presented its blueprint architecture. Moreover, we identified the main components of CoaaS and explained their roles. Based on the provided discussion, we can proceed to identifying the requirements to the context storage management subsystem of the CoaaS platform that is the main focus of this dissertation.

3.3 CSMS REQUIREMENTS

In this section, we identify and discuss requirements to CSMS. We look at the requirements from two perspectives. The first perspective is the functionality that should be supported by CSMS to effectively serve as a part of the CoaaS platform. Analysis of functional requirements will lead to the development of the CSMS architecture.

The second perspective is the technological side of the problem. The proposed analysis of technological requirements leads to the mapping of the architecture on the existing basis of data storage and processing technologies. Then, we proceed to the physical architecture and its implementation, which are described in Chapter 4.

3.3.1 CSMS FUNCTIONAL REQUIREMENTS

In this subsection, we briefly identify the main functional requirements. The method for elicitation of functional requirements was based on thorough and critical literature review

(Chapter 2), comparative analysis of related work, and analysis of the bIoTope project use cases.

We have separated the functional requirements into three groups: the fundamental requirements, the advanced requirements and the CoaaS-specific requirements.

The basic CMP can work in the '*all or nothing*' fashion where the '*all*' stands for the database mode and '*nothing*' stands for the redirector mode (refer to Chapter 2). While the redirector mode cannot provide a decent performance in wide scale scenarios, it can still be useful in certain cases; besides, it has minimal requirements of the storage subsystem.

The fundamental functional requirements group contains minimal requirements, which are essential for any CMP to enable the ability to serve the use cases. These requirements are as follows:

Context provider selection. The CMP storage system should be able to return a list of context providers, which can potentially provide the contextual information requested by the consumer. For that, a repository of possible providers should be organised.

Subscription/Situation monitoring. In order to support the PUSH-based queries (subscriptions), the storage subsystem should support the following two main functionalities: (i) the subscriptions derived from PUSH-based queries should be stored in a repository, and (ii) every event, which arrives to the CMP, should be checked against each related CDQL WHEN clause, to estimate the occurrence of a situation, which triggers the subscription.

Having these minimal requirements satisfied, the storage system can support a CMP operating in the redirector mode. However, to provide advanced functionality or better performance, the requirements which form the *advanced functional requirements* group should be satisfied.

The advanced functional requirements group contains functionalities, which add the possibility of a CMP to work in the database mode. It includes the following:

Current context storage and retrieval. If the database or NoD-NoR mode is chosen, the storage system should contain the repository and interfaces in order to enable the ingestion of current context data and its retrieval for serving context queries.

Historical context retrieval. The storage subsystem should support storage and retrieval of historical context data to support the historical queries as well as the predictive functionality of the CMP.

As it was mentioned in Chapter 2, the CoaaS platform is designed to operate in the NoD-NoR mode. Moreover, CDQL offers a novel approach to context querying with several unique features embedded into the language. That adds another group of requirements - the CoaaS-specific requirements.

The CoaaS-specific functional requirements group contains requirements that need to be satisfied to serve the use cases, which rely on the novel features provided by the CoaaS platform.

CDQL request processing. An essential requirement is to be able to connect the CoaaS Query Engine (QE) with the underlying data storage facilities. Consequently, the storage subsystem should contain a mechanism for transparently converting a CDQL request to the required format.

On the fly retrieval of external data when the cached data is expired at the query arrival time. To support the NoD-NoR mode, the storage subsystem should be able to combine the retrieved from storage cached context with the data that was retrieved during the query serving (on the fly), as part of the cached data was expired.

Post-retrieval computations. Another side of the request processing is the support for such CDQL features as CST-based functions, window-based functions, aggregation functions, built-in and computation functions. The storage subsystem should support the storage of definitions of functions (applicable to CST functions) as well as processing the data by these functions after retrieving the raw context from the cache or external providers.

Analysis of query patterns and context sources behaviour for cache management purposes. The NoR-NoD mode of operation requires an intelligent approach for cost-efficiency adaptation. Enabling this approach requires storing the patterns of queries to context attributes, as well as storing the patterns of changes in the attribute's value. Acquiring these patterns requires an analytical module embedded in the cache management system of the storage.

Higher-level cache storage and retrieval. As described in Section 3.2.3, CDQL supports complex queries, which consist of requests to different entities. The storage subsystem should support storing cached context at levels higher than simple context attributes, as it can significantly improve the cost-efficiency of system operation, and reduce the latency of query serving.

Optimization of the external context retrieval strategies based on the data extracted from the pattern analysis. The storage subsystem should contain a module responsible for

calculating the optimal strategy of computational resource allocation, as well as the optimal strategies for the retrieval of context from the external sources.

We discuss the realisation and more detailed justification of these requirements in Chapters 4 and 5.

3.3.2 TECHNOLOGICAL AND NON-FUNCTIONAL REQUIREMENTS

The group of technological requirements includes such areas as modelling, scalability, reliability, performance, and querying capabilities. During the development of CSMS, we aimed to make use of existing technologies and build the CSMS modules on top of them, instead of trying to create another base-level technology. By 'technological requirements' we mean the requirements to technologies, which are used as the base-level, in other words, as enablers for CSMS modules. In cases where one technology could not cover the requirements, we united several pieces through the higher-level components; thus, following the Polyglot Persistence approach [193]. The decisions we made during the implementation phase were according to the defined requirements.

We have identified the following technological requirements of a context storage middleware:

Disk-based core storage. The modules of CSMS which provide the core CSMS functionality should be based on technologies which support disk-based storage. In-memory systems (e.g. in-memory databases) are gaining more and more attention nowadays. However, the reliability of this class of systems is significantly lower than the reliability of disc-based systems, as in-memory systems are heavily dependent on the infrastructure and hardware. In case of a hardware fault, the loss of stored data is inevitable.

The loss of some types of information is tolerable for the CoaaS use cases. For instance, the loss of cache will cause a decrease of efficiency during a certain period, but will not completely disrupt the operation. However, the failure of such a component as the repository of context providers will cause a complete halt of the system until the recovery of the repository is completed. Losing the historical data will cause the inability to answer requests about the past, but the main CoaaS functionality will be still available. At the same time, the potentially tremendous amount of historical data makes it hard and very expensive to store all of it in an in-memory storage system. Moreover, the possibility and all the benefits of the proactive adaptation will be lost until the time when the history will be filled again with fresh data. The analytical metadata, on the contrary, requires much less space to store. However, acquiring this

metadata required significant time and processing power, making it too expensive to discard. Eventually, at this point of the state of technology, we made a decision to focus on reliable disc-based systems, and use expensive and less reliable in-memory systems only for boosting the cache performance in a situation where it is cost-efficient.

Scalability. It is hard to predict the amount of stored information, but in case of smart city scenarios, it would not be possible to provide the storage service by one server node. This means that the proposed solution must be horizontally scalable. By horizontal scalability, we mean the ability of the system to use many cloud instances (server nodes) in parallel. Another important factor is the ability to add more server nodes to the cluster during the growth of the system, and, also, to reduce the number of servers used in cases when there is no need in that amount of processing or storage resources. These scaling processes should not require major manual efforts.

High Availability – the storage components should not have a single point of failure. In case of one or several server nodes failure, the queries should be served from the nodes which are still online, and the failover should be managed automatically.

Various approaches to CAP theorem. Traditionally, one of the main principles of database management systems is ACID – Atomicity, Consistency, Isolation and Durability. According to the CAP theorem, we cannot have consistency, availability and partitioning tolerance in one system at the same time. As it is mentioned in Chapter 2, the high demand for horizontal scalability and high availability (partitioning) was one of the factors that gave momentum to the NoSQL movement, where these factors outweighted the requirement for ACID compliance.

As it was discussed in Chapters 1 and 2, the context retrieved from different sources can already be uncertain and, potentially, conflicting. Moreover, while being stored, it loses the freshness. Consequently, it makes it hard to speak about the complete consistency of IoT data in open ecosystems. That means the middleware solution in some cases can afford lack of transactional support and consistency in favour of high availability and partitioning, as the requirements for horizontal scalability and availability have higher priority. At the same time, some parts of a middleware system, such as consumer management, can have strong requirements for consistency, and these requirements should be satisfied also.

Structural freedom. The contextual information is pushed into CoaaS by independent context providers. It is hard to force these external entities to supply the data structured

precisely the same as expected by the storage side. The data description format can change or be extended over time, while the context providers are not updated synchronously with the platform. The rigid nature of relational databases makes it hard to deal with the challenges above. During recent years the JSON mark-up has become a de-facto standard in the web document exchange. The document-based storage allows some level of structural freedom, at the same time still keeping documents semi-structured and available for indexing.

Eventually, we came to the conclusion that the CSMS storage should be able to store structured or semi-structured data without applying severe restrictions on its structure.

Interconnected entities – in some cases storage must facilitate the means for storing highly interconnected data, (e.g. relations of people, organisations, transport, and infrastructure), and effectively running queries over such data.

Veracity – different sources can supply information that can be conflicting or uncertain and there should be a way to store all variants of incoming data with annotations about the identity and trust level of the originator and rank of the suggestion.

Large amounts of sensory data – sensors and other Internet-enabled devices generate a large number of time series events of similar, but not the same structure.

Ontology (semantic data) support – many research projects model data using ontological principles as it is a common way for modelling the domain knowledge. However, this approach has performance issues when used for ingesting and storing large amounts of raw data and low-level context. At the same time, the ability to store semantic data is useful, at least for the validation of the incoming data.

Fast information retrieval and rich indexing capabilities – performance is the critical requirement for context delivery in smart cities applications. The base-level technologies should provide efficient indexing features to enable the fast retrieval of stored context.

Fast writes – streams of sensor readings should be ingested into the storage system without long delays, queues or loss.

Geospatial data – many IoT scenarios and applications are highly dependable on geospatial context, so the middleware storage must be able to provide effective indexing capability for this type of context.

On-premises or cloud deployment – to avoid the vendor lock-in, the used base-level solutions should be possible to deploy on-premises or in an IaaS cloud; proprietary solutions which are available only as PaaS carry risks for the sustainable development of the CSMS.

Available open-source solutions – the used base-level technologies should be open-source, or at least not heavily licensed.

In this section, we identified the core set of requirements to CSMS, which are enough to cover the CoaaS -related scenarios that we consider in the bIoTope project, or the scenarios that are often discussed in modern IoT ecosystems literature. As the business, legislative, or technological changes are being introduced, there will be an increasing need for new features of the platform, and consequently, the list of requirements to CSMS can be extended to reflect these changes.

After analysing the requirements for the middleware storage system (i.e. CSMS), it becomes evident that fulfilling all the requirements with one existing base-level solution is not feasible. In cases where one technology cannot cover the requirements, we can unite several pieces through the higher-level components; thus, following the Polyglot Persistence approach [193].

In the next section, we introduce the principal architecture of CSMS and its main modules.

3.4 CSMS ARCHITECTURE

In this section, we present the architecture of CSMS, which is designed based on the discussion provided in the first part of this chapter, as well as on the findings from the literature review (Chapter 2).

The principal components of the context storage management system are: (i) Storage Query Execution Manager (SQEM), (ii) Context Service Description Repository, (iii) Context Repository, (iv) Performance Repository, (v) Subscription module, and (vi) a set of recommenders that enable the optimisation of the system performance.

Schematically the CSMS architecture is presented in Figure 3.8.

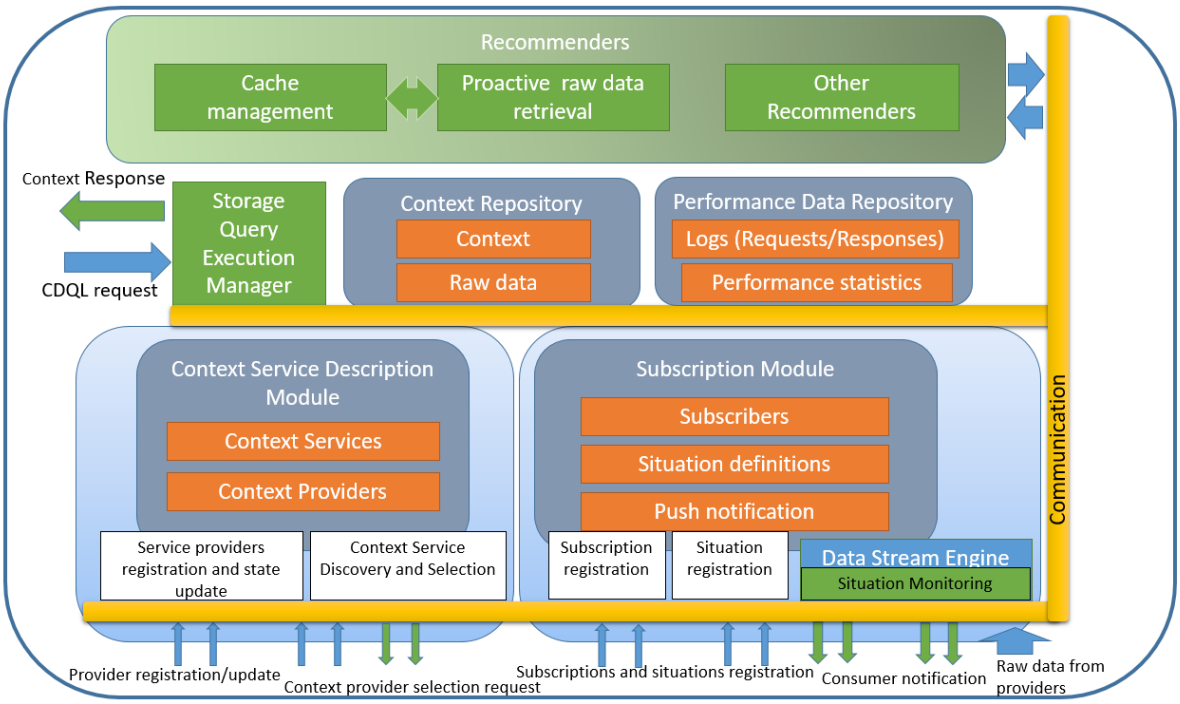


Figure 3.8 - CSMS architecture

Below, we provide a description of the aforementioned components.

Storage Query Execution Manager (SQEM)

The SQEM is the entry point for context queries. We should consider two different scenarios: (i) PULL-based query execution and (ii) PUSH-based query execution. During the PULL-based query execution, the CoaaS query engine (QE) transmits the parsed context query coming from a consumer. First, the query execution engine needs to obtain a list of context providers, which can serve as data sources. For that reason, the Context Service Description Repository (CSDR) is used.

Accessing the repository, which is a persistent storage, requires the ability of SQEM to construct a query to a corresponding datastore. For that, SQEM contains a translator, which generates queries in native query languages to corresponding datastores.

Once the list of candidate providers is obtained, SQEM accesses the Context Repository (CR) to fetch the stored (cached) context. We also apply the term '*cached'* to the stored context data, as it is mostly based on transient IoT data. In the case when the cached data is missing, or it is considered to be expired, SQEM requests the missing data through the QE, based on the context service description. In a case when the CDQL query contains functions (CST,

aggregation, computation functions), which require post-processing, SQEM manages the process of applying these functions to the retrieved context.

When the query is serviced, and the context is returned to the consumer, SQEM saves the query together with the result into a higher level of cache to enable the reuse of already computed results. These higher levels of cache are also stored in the CR. SQEM also saves the statistics about the access to entities, instances of entities, and context attributes into the Performance Repository (PR).

In case of a PUSH-based query, SQEM converts the *'WHEN'* clause of a query into a subscription, which is registered by a Subscription Module (SM). If a subscription contains a time window-based function, SQEM registers this function in an event-processing engine. Then all the incoming events are '*percolated'* through all the registered subscriptions.

Context Service Description Repository (CSDR)

CSDR, which is graphically represented in Figure 3.9, is the primary source of information for the query engine to determine a list of sources, from which the needed contextual information can be retrieved. The result of accessing the CSDR is a list of *context services* together with their properties. These properties should include, but not be limited to the following: (i) entity type, about which the service is provided (e.g. carpark, weather station), (ii) address of the endpoint (e.g. IP address, FCM), (iii) format of communication (e.g. HTTP post request), (iv) credentials to access the server (e.g. login/password), (v) cost of service request, and (vi) service schema (MobiVoc). The service schema contains a reference to the semantic vocabulary, according to which a new document with information about a service can be validated. The identifier of the service links the service description with the cached value of the last retrieved sensor measurement (raw context), and also the historical data if it is collected.

Another part of the CSDR is the *Context providers store*. Context providers are linked to context services so that one context provider can provide one or more services. The context providers store accumulates the information about billing and service level agreements between the provider and the Context Management Platform (CMP).

The third part of CSDR is the utility information collection. While the first two parts are created and initiated by the provider, the utility information is created by the platform and contains aggregated information about the provider, based on the analysis of statistics. These

parameters include but are not limited to the latency or access and the quality of supplied context.

Incoming requests to CSDM are (i) context queries, (ii) registration of context services, (iii) registration of context providers, and (iii) update of context service or providers state.

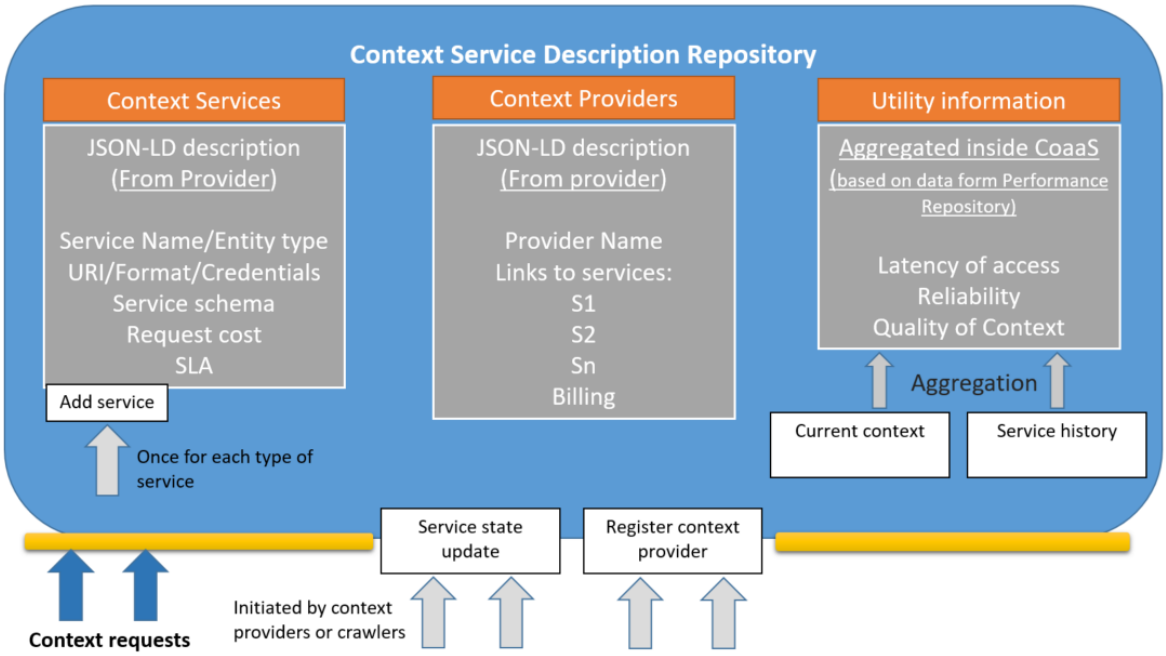


Figure 3.9 - Context Service Description Repository

Context Repository

Context repository (CR), which is graphically represented in Figure 3.10, contains the data, which can be used for directly answering context requests without accessing external providers. In general, CR can be seen as a cache, as the stored IoT data is subject to aging. CR contains the following parts: (i) current context, (ii) historical context, (iii) cache of higher levels.

The current context is the representation of the state of the context service. The current context store contains the context attributes that were retrieved last. The information is structured according to the semantic vocabulary, which is corresponding to the service entity type and linked to the service (CSDR). For each context attribute, the expiry time is attached. The expiry time can be provided by the source, or it can be obtained by analysing the historical information.

Historical context is time-series data, which is obtained by moving the current context to a separate store when the last cached context value is updated. Historical context is used for answering historical queries, aggregation queries and queries which require context predictions. It is also used for estimating the expiry period of context attributes.

The cache of higher level data contains the answers for context queries or parts of these queries, which can be reused in the nearest future; thus reducing the load on the CSMS.

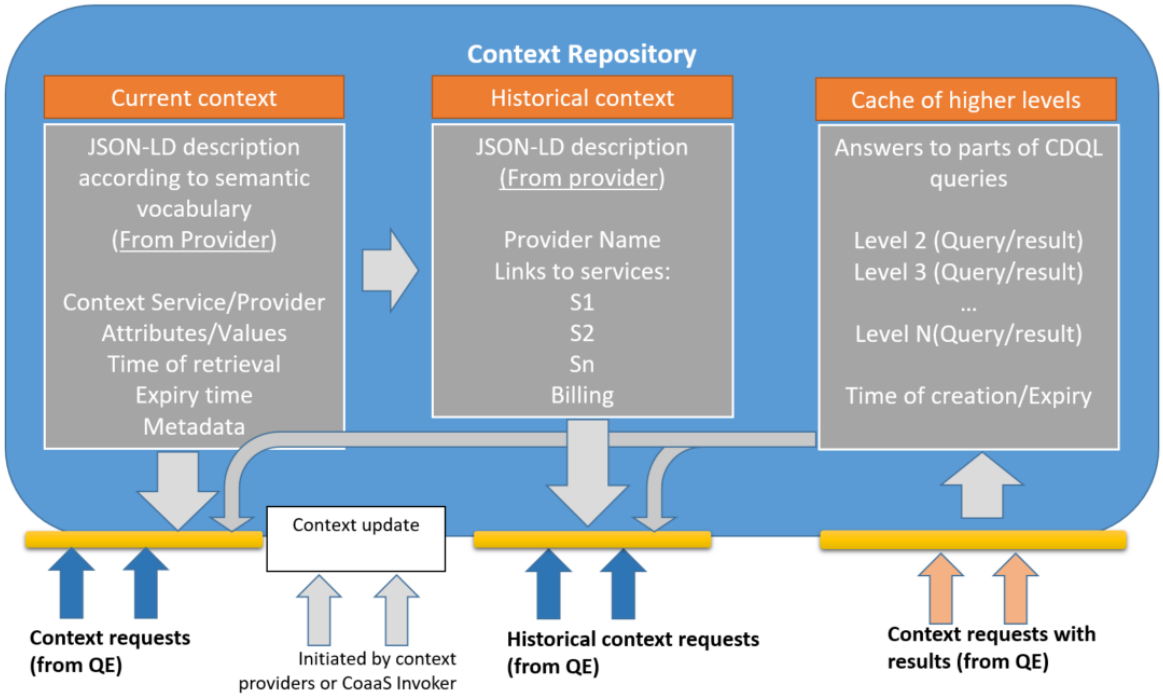


Figure 3.10 - Context Repository

Subscription Module (SM)

Subscription module, which is presented in Figure 3.11, facilitates the whole process of notifying the subscribers about a detected situation. By 'situation' here we mean the WHEN clause of the CDQL PUSH-based query. Each consumer can register an unlimited number of PUSH-based queries, and all the incoming events will be 'percolated' through these queries. If needed, CoaaS will initiate periodical requests to data sources for keeping the state of the involved context attributes up to date. Moreover, incoming events are used to change the condition of the service provider in CSDM if this state changes.

Main parts of the SM contain the data about (i) Subscribers, (ii) Subscription Definitions (CDQL WHEN), (iii) Actions (CDQL SELECT), and (iv) CST-based situation definitions.

The S*ubscribers* part contains the properties of subscribers (e.g. endpoint where to send the notification when the subscription is triggered).

The *Subscription Definitions* part contains the WHEN clauses of all the registered subscriptions, which are stored in a way which is convenient for accessing when the context update needs to be percolated.

The *Actions* part contains the SELECT clause of all PUSH-based queries when the subscription is triggered; the SELECT clause is executed in the same way, as if it would be issued as a PULL-based query.

The *CST-based situation definitions* part contains the definitions of all the registered situations, which are modelled based on the Context Spaces Theory [60].

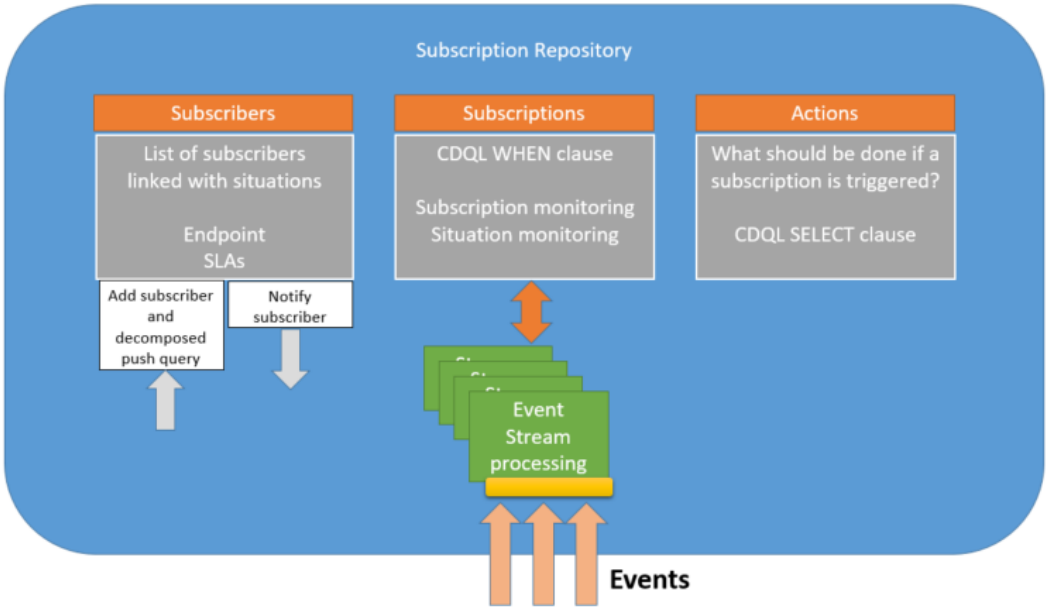


Figure 3.11 - Subscription module

Performance Repository

The performance repository (PR) contains the data about the performance of the external providers, consumers and internal components of the system. *For the external providers*, the latency of access is measured for each context retrieval. Based on this information, the expected latency and reliability is computed, which are then stored as part of CSDR.

For the consumers, the latencies of servicing the queries is measured for the purpose of fair billing. Moreover, each access to context attributes is registered in order to later estimate the popularity of each attribute. Whole CDQL queries with the attached execution plans and

details about how fast each part of the execution plan was serviced are also stored in PR for later being used as datasets for performance tuning.

For the internal components, the cache management decisions, storage and compute resource allocation decisions, latencies of access to cached data, and costs of used resources are recorded.

Recommenders and planners

For the purpose of managing the storage query process and adjusting the performance and cost-efficiency of the whole system, several recommenders and planners are needed. These software components rely on query execution statistics, logs of requests, responses, and plans. By analysing this data in background batches, the system adjusts its performance according to the load.

Proactive Raw Data Retrieval (PRDR) planner is responsible for initiating the data retrieval from remote context source and caching this data item before a context query requested this data. This strategy helps to reduce the time of serving the query. At the same time, retrieving data that will not be used is not cost-efficient. For that reason, the PRDR relies on predictive algorithms to keep the optimal level of performance and cost.

In-memory caching and co-location (IMCC) recommender is responsible for offering the following recommendation: what piece of information (e.g. level of cache) should be kept inside the in-memory caching node, or in a cheaper disk-based storage, or nor kept at all. This task involves predicting the probability of reusing the same context, based on its parameters (especially context lifetime), former behaviour patterns of consumer and other relevant historical data. On the other hand, the cost of consumed resources and corresponding delays in query serving is taken into account.

The technological aspects of storage

CoaaS architecture depends on several types of data: structured, semi-structured and unstructured. Apart from storing data for later retrieval (database-like access), there exist other data storage-related technologies, which include storing incoming messaging as queue, and processing streams of incoming messages.

Document store (DS) is the core part of the CSMS. DS stores all the incoming data and derived context that can be reused for serving future context requests. This load imposes a strict requirement towards reliability and scalability of DS, as this part must always be available. As

it is considered that the size of DS will be enormous, proper indexing is required to keep the data retrieval time inside the SLA bounds.

In-memory cache (IMC). Operating in near real-time is one of the requirements for the CoaaS middleware. For this reason, an in-memory caching layer can be used for storing context requests and corresponding answers. This layer can significantly reduce the query processing time in the case when a query with a complete or partial similarity has already been processed recently.

Message queue (MQ). In the perfect situation, the incoming messages which contain context updates, as well as context queries can be processed immediately at the moment of arrival. However, the incoming traffic can be "bursty". In such a case, there is a need to allocate more resources to process these requests; otherwise, the server will not be able to handle the load and some of the requests would be rejected. To efficiently tackle the issue, distributed message queues are used as the buffer for incoming messages. Moreover, in case of a failure and corresponding delays in the processing side, MQ accumulates the incoming messages and after the failed service is fixed, the processing can be continued from the moment when the service has stopped.

Complex Event Processor (CEP). Serving the PUSH-based queries (subscriptions) requires not only checking if the conditions in any registered subscriptions are triggered, or not, by a certain incoming event. There is also a need to monitor trends and enable the processing of sliding window-based functionality. Another task is to provide predictions based on the incoming data streams. For these purposes, a special class of data processing software is needed. Such software is usually referred to as CEP or data stream processors.

3.5 CONCLUSION

In this chapter, we discussed our approach to architecting the CSMS for the CoaaS platform. First, we introduced the vision of the CoaaS platform. We recapitulated the blueprint architecture and the main concepts of the CDQL language. Based on the provided discussion as well as on the analysis of CMP requirements provided in Chapter 2, we formulated the main requirements to CSMS. We grouped these requirements into functional requirements (Section 3.3.1) and technological requirements (Section 3.3.2).

Based on the formulated requirements, we proposed the architecture of CSMS and its modules (Section 3.4). In the next chapter, we proceed to the detailed description of the design and implementation of CSMS main modules.

Chapter 4: Design and implementation

4.1 INTRODUCTION

In this chapter, we describe the design and implementation of the Context Storage Management System (CSMS) and its main modules, which were conceptually introduced as architectural elements in Chapter 3.

The chapter is structured in the following way: In Section 4.2, we provide an overview of the CSMS implementation and define the queries, which we use throughout the chapter to illustrate the dataflow. In Section 4.3, we describe the details of the Storage Query Execution Manager (SQEM) implementation, which contains the main logic, facilitating the communication between the CoaaS Query Engine (QE), storage repositories and event stream processor. Then, in Section 4.4 and Section 4.5, we discuss the implementation of context service description and context repositories. In Section 4.6, we present the implementation of the subscription module. Section 4.7 concludes the chapter.

4.2 AN OVERVIEW OF KEY MODULES OF CSMS IMPLEMENTATION

Figure 4.1 provides a high-level view of the main modules of CSMS implementation. The horizontal blue dotted line represents the border between the Query Engine (QE) and the CSMS. QE converts the initial CDQL query to a set of context requests, which are sent to SQEM.

The core logic of CSMS is realized as JavaEE[6] applications, which are running over a Payara Application Server 5[7]. The main components of CSMS are the Storage Query Execution Manager (SQEM) and the Cache Manager (CM). SQEM governs the process of handling the requests from the QE and connecting them to corresponding repositories. For this reason, SQEM contains wrappers, which automatically generate queries to underlying datastores.

In particular, the Mongo wrapper was created for generating MongoQL[8] queries. MongoDB was chosen as a main datastore for CSMS repositories, as it supports document-oriented storage, is widely used, is horizontally scalable, does not require licensing, and is cross platform. Moreover, it also provides an in-memory option, reducing the need for adopting another datastore for fast caching, which will cause more complexities in development.

[6] https://www.oracle.com/java/technologies/java-ee-glance.html
[7] https://www.payara.fish/
[8] https://docs.mongodb.com/manual/crud/

Further, SQEM contains a wrapper to handle the process of event stream monitoring. To this end, the Siddhi[9] wrapper was created to generate Siddhi applications, which define the event stream processing chain in WSO2 Siddhi CEP 4.0[10] component.

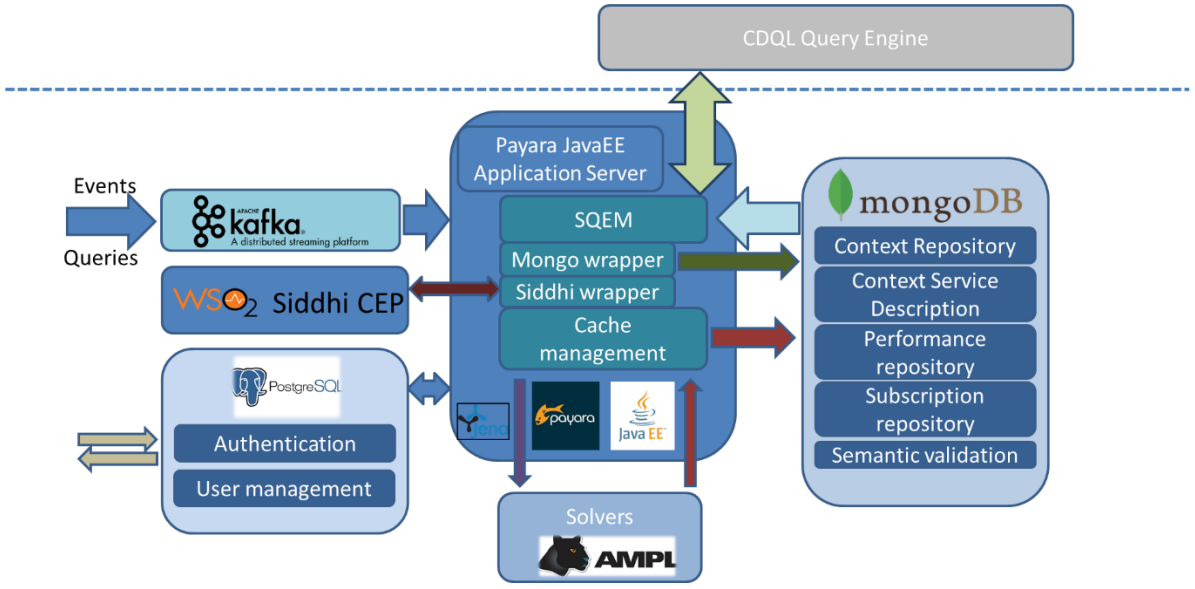


Figure 4.1 - CSMS implementation high-level view

The Cache Management (CM) component contains the algorithms for analyzing the IoT data (context attributes), which are stored in the context repository. The usage of these attributes (requests) is also taken into account to compute and apply the optimal caching strategy. The caching strategies and models are discussed in detail in Chapter 5 and 6, and the evaluation of the proposed models is provided in Chapter 7.

The main repositories, which are the Context Repository (CR), Context Service Description Repository (CSDR), Subscription Repository (SR), and Performance Repository (PR), are based on corresponding MongoDB 4.0[11] databases and collections. The incoming events are first routed into the Apache Kafka 2.3[12] queue, which is used as a persistent buffer for incoming messages. For platform administration purposes, such as user authentication and management, CSMS is using PostgreSQL 11[13].

The design and main workflows of SQEM, main repositories, and stream processing are described in the following sections of this chapter. For illustration purposes, two CDQL queries

[9] https://docs.wso2.com/display/CEP420
[10] https://docs.wso2.com/display/CEP420
[11] https://www.mongodb.com/
[12] https://kafka.apache.org/
[13] https://www.postgresql.org/

are used. In general, CSMS does not deal with full CDQL queries directly, except the retrieval of full query cache results. The QE breaks the CDQL query into requests, which are issued to CSMS in a particular order; the order is defined by the QE. A request is a part of the CDQL query, which is based on a single entity. In other words, requests are joined together to form a full CDQL query. A simple CDQL query can consist of one request, while a complex query can contain as many entities as the developer has defined. The term *'entity'* is used to refer to a class or a type of things (e.g. car, location, or event). When we refer to a single thing (e.g. car#1), we use the term *'instance of an entity'* or *'instance'*. In the current implementation, the instances are stored and returned in a JSON document format, which is shaped in accordance with a corresponding semantic vocabulary. For that reason, the term *'document'* is also used, referring to an instance of an entity during the data processing or filtering stage.

According to [98], a CDQL query can be PULL-based or PUSH-based, and both modes should be supported by CSMS. For the discussion of a PULL-based query handling by the CSMS, a CDQL query that is presented in Figure 4.2 is used. The query was designed during the bIoTope consortium meeting to facilitate the search of internet-connected waste containers for the developers of the Smart Waste Management (SWM)-related applications [194],[47]. There are three entities defined in the query: (i) Location (*targetLocation, line 5*), (ii) Weather (*targetWeather, line 7*), and (iii) Waste container (*targetBin, line 9*). Consequently, the query is executed as a sequence of three context requests from the QE to CSMS. These requests are (i) find a set of services, which can return locations based on the given criteria, or find cached locations, (ii) find services that can provide information about weather in locations, or find cached weather data in locations, (iii) find the waste container entities around given locations with a distance less than the given criteria.

The purpose of the query is to find all the waste containers (bins), which are located near the university, not filled over the top (*line 11*), and accept both plastic and glass waste (*line 12*). The subtle moment is how the proximity "*near the university*" is defined. The location of the university is defined by the address (*line 6*). Then the proximity is influenced by the weather. In case it is "good for walking", the 3km radius is used. In other case, it is not "good for walking", so only 100 m radius is applied (*lines 14-16*).

```
prefix swm:http://swm.schema.org,
prefix schema:http://schema.org
pull (targetBin.*,situWeather(targetWeather).goodForWalking,targetWeather.*)
define
entity targetLocation is from schema:place where targetLocation.address =
"Bol'shoy Prospekt Vasil'yevskogo Ostrova, 6, Sankt-Peterburg, Russia, 199034",
entity targetWeather is from schema:weather where targetWeather.location.latitude =
targetLocation.latitude and targetWeather.location.longitude = targetLocation.longitude,
entity targetBin is from swm:SmartWasteContainer where
targetBin.ownedBy = "ITMO University" and
targetBin.Capacity.realTimeCapacity < 100
and targetBin.AllowedWaste containsAny ["swm:Plastic","swm:Organic"]
and targetBin.AllowedWaste containsAll ["swm:Plastic","swm:Glass"] and
(distance(targetBin.geo,targetLocation.latitude,targetLocation.longitude,"WALKING") < 100 or
(distance(targetBin.geo,targetLocation.latitude,targetLocation.longitude, "WALKING") < 3000
and situWeather(targetWeather).goodForWalking > 40) )
```

Figure 4.2 – PULL-based CDQL query

While the approach allows a simple way to request context for a developer of a context-aware application (context consumer), supporting such queries requires significant development effort on the platform's side.

The novel feature of CDQL, compared to other context query languages, is its support for the Context Spaces Theory (CST) –based functions, which are used to describe 'situations'. An example of a CDQL definition of the CST-based situation function *'situWeather()'*, which is a part of a PULL-based CDQL query, is presented in Figure 4.3. The basics of CST are presented in Chapter 2 and [63], and the concept of CST-based situation functions is described in Chapter 3, [195] and [98]. In this example, the situation function defines three situations, which are *'cold'*, *'hot'*, and *'goodForWalking'*. The CDQL query uses the *'goodForWalking'* situation. This situation is defined by two parameters, which are the *airTemperature* and the *windSpeed*. Each parameter has a *weight*, and each parameter is also divided into ranges. *Ranges* define ranges of values, and when the input value is applied to the function, the *belief* of the corresponding range is applied. Then, the weights and the beliefs of each parameter are used to obtain a final value of the situation probability [63].

The input values for the situation function (air temperature and wind speed) are obtained from the entity *'Weather'*, which is retrieved based on the user's device location from a corresponding weather service. Eventually, the function returns the probability of situation 'Good for Walking', which is later compared with the criteria given in the query (40%), and the final Boolean value is obtained.

```
prefix schema:http://schema.org
create sFunction situWeather is on
schema:weather as r1
{ "hot" : {
r1.airTemperature : {
ranges : [
{value:(28;40], belief : 100 },
{ value:(23;28], belief : 60 } ] ,
weight : 12 } },
"cold" : {
r1.airTemperature : {
ranges : [
{value:(0;13], belief : 100 },
{ value:(13;22], belief : 60 } ] ,
weight : 12 } },
"goodForWalking" : {
r1.airTemperature :
{
ranges : [
{ value:(0;6], belief : 20 } ,
{value:(6;13], belief : 50 },
{ value:(13;28], belief : 100 } ,
{ value:(28;33], belief : 50 } ,
{ value:(33;38], belief : 20 } ] ,
weight : 10 } ,
r1.windSpeed : {
ranges : [
{ value:(0;8], belief : 100 } ,
{value:(8;20], belief : 50 },
{ value:(20;30], belief : 20 } ,
{ value:(30;40], belief : 10 }] ,
weight : 5 } }
} |
```

Figure 4.3 – Situation function definition

The processing of other operators used in the main ‘waste container search’ query is discussed in Section 4.3. For the illustration of how the second type of CDQL query (PUSH-based) is supported by CSMS, the smart vehicle preconditioning scenario is used. The PUSH-based CDQL query is presented in Figure 4.4.

```
prefix schema:http://schema.org
push (events.*,eventLocation.*,driver.*,car.*,temp.*)
when timeDifference(events.startDate,currentTime("Australia/Melbourne")) < 50
and distance (eventLocation.latitude,eventLocation.longitude,driver.geo.latitude,driver.geo.longitude).distance > 2000
and distance (car.geo.latitude,car.geo.longitude,driver.geo.latitude,driver.geo.longitude,"WALKING").distance < 500
and distance (eventLocation.latitude,eventLocation.longitude,driver.geo.latitude,driver.geo.longitude,"WALKING").distance >
distance (car.geo.latitude,car.geo.longitude,driver.geo.latitude,driver.geo.longitude,"WALKING").distance
and decrease(distance (car.geo.latitude,car.geo.longitude,driver.geo.latitude,driver.geo.longitude,"WALKING").distance,5m)
and (temp.airTemperature < 20 or temp.airTemperature >25 )
define
entity events is from schema:event where events.attendee.email="biotope2018.au@gmail.com",
entity eventLocation is from schema:Place where eventLocation.address=events.location.address,
entity driver is from schema:Person where driver.driverID="biotope",
entity car is from schema:Vehicle where car.vehicleIdentificationNumber = "9d791e4d-8181-4bca-a8e3-f83357e525ad",
```

Figure 4.4 – PUSH-based CDQL query

The query was developed as a part of the smart mobility use case of the bIoTope project in collaboration with a large vehicle manufacturer. The main idea behind the presented PUSH query is to facilitate the following scenario:

A driver has a meeting planned in his calendar. The meeting is far from the place where the driver is now. The temperature in the car is out of the preferred range, so it would be not comfortable for a driver to get into the car. When the driver starts walking towards the car, this event is detected and used by the car to start the A/C or the heater, in order to adjust the temperature to the preferred value. This is called preconditioning. To facilitate this scenario, CoaaS has to monitor the related context attributes continuously and, when the situation is detected, inform the context consumer, which is the backend system of the car manufacturer in this case.

There are four entities defined in the query: (i) *events*, (ii) *eventLocation*, (iii) *driver*, and (iv) *car (lines 10-14)*. The car is defined by its VIN number; the owner/driver is linked to the car in the context service definition. The driver is defined by the known identifier, the event is linked to the driver through joining the driver's e-mail with the list of event attendees, and the event location is extracted from the event definition.

The most important part of the PUSH-based query is the '*WHEN*' clause (*lines 3 - 9*), as it is the section where the 'subscription triggering' is defined. When all the conditions in the '*when'* clause are in the *'True'* state, the '*select'* part of the query is executed.

The feature which needs the most discussion is the *decrease()* function. The *decrease(distance(), 5min)* function returns '*True'* if the distance between the driver and the car is decreasing during the last five minutes (Figure 4.4, line 8). Supporting such functionality requires event stream processing. This feature is discussed in detail in Section 4.3.4. Processing of the other components of the '*when'* clause is discussed as the part of a subscription module implementation (Section 4.6).

We have introduced the main components of the implemented CSMS design, and also defined the queries which will be used for the demonstration of the dataflow throughout this chapter.

4.3 SQEM MODULE IMPLEMENTATION

In this section, the design, implementation and functionality evaluation of the Storage Query Execution Module (SQEM) is presented. According to its name, SQEM is a gate between the Query Engine, which deals with external context providers and consumers directly, and the components of CSMS. The main objective of SQEM is to serving the context requests from QE by routing these requests to a corresponding storage component and converting these requests to a proper format. There are several reasons for this design decision.

First, CDQL is designed to query real-time data from context providers together with the data that is cached in the storage. The query format is designed to help application developers to express their need in context. This includes high-level functions (refer to Chapter 3) and other CDQL-based functionality. Correspondingly, the format of a query is not aligned to the format of any existing datastore.

Since the beginning of the CoaaS project, it was decided to follow the principle of keeping the main interface of CoaaS (CDQL) and the Query Engine (QE) datastore agnostic. In case of future technological or licensing changes in underlying datastores, there would be a need to make changes only in SQEM part. Consequently, no changes would be needed in QE, and most importantly, no changes will be needed in queries, which are embedded in external software of context consumers.

Another reason for the need of the query translation in CSMS is the possibility to use the benefits of a hybrid system, comprising several underlying technologies such as document store, in-memory key-value (KV) store, data stream processors, and semantic graph storage. A CDQL query can be split into parts by SQEM, according to data placement strategy. Queries to each component are expressed with the means of different internal query languages.

One more reason for the need of the SQEM layer between QE and storage is the possibility to execute high-level queries without a need to investigate complexities of data organisation. These high-level functions need to be translated into queries, which match the context model. For example, if a CDQL query looks like *"find carparks available between 11.30 and 16.40 with cost less than $5",* the real data structure describing a parking facility will be complex, including information about the schedule during different days, price ranges, maximum length of staying, information about the user's permit and residence. To simplify the work of an application developer, CoaaS provides the option of using custom hosted entity-related functions such as "*availability*" and "*cost*". Handling of such high-level functions which are tightly connected to the context model is realised at the level of SQEM, as these functions are a part of the post-retrieval context processing flow. The details of post-retrieval data flow are discussed in Section 4.3.3.

4.3.1 CDQL TO MONGOQL QUERY TRANSLATOR

A CDQL request which arrives from the QE to SQEM must be translated for the possibility of being executed in underlying levels of CSMS. CDQL requests arrive through an API which provides the request in Reverse Polish notation (RPN). In this section, we illustrate

how a CDQL request is transformed and executed, to retrieve data from the Context Service Description Repository (CSDR) and the Context Repository (CR).

```
1  targetBin.ownedBy "ITMO University" = targetBin.Capacity.realTimeCapacity 100 <
2  targetBin.AllowedWaste ["swm:Plastic","swm:green"]  containsAny
3  targetBin.AllowedWaste  ["swm:Plastic","swm:Glass"] containsAll
4  distance(targetBin.geo,targetLocation.latitude,targetLocation.longitude,"WALKING") 100 <
5  distance(targetBin.geo,targetLocation.latitude,targetLocation.longitude, "WALKING") 3000 <
6  situWeather(targetWeather).goodForWalking 40 > and or and and and and
```

Figure 4.5 – RPN representation of a CDQL request

The original CDQL query is written in infix form. In order to compute the satisfiability of the WHERE clause, it is more convenient to transform the infix form to the Reverse Polish notation (RPN) form (postfix). In Figure 4.5, an example of an RPN representation of CDQL request is presented. SQEM accepts the RPN-CDQL input and generates an equivalent query to the underlying datastore. Such an approach significantly simplified the query translator algorithm, as it eliminated the dependence of infix expressions on brackets.

As it was stated in the previous section, the main functionality of SQEM is to facilitate the process of CDQL requests execution on the storage side. The CSDR and CR modules of CSMS are based on document-oriented storage principles. For the current implementation of CSMS, MongoDB 4.0 has been chosen. Thus, SQEM needs to be able to convert a CDQL request into a correct MongoQL query, which will be executed in MongoDB. In the following paragraphs, a description of transformation rules that are used for translation from CDQL to MongoQL is provided.

Simple and **complex** operators can be distinguished. Simple CDQL operators are executed directly in the database, as analogues for these operators exist in the database. The following list of **simple CDQL** operators is supported:

“=”, “<”, “>”, “<=”, “>=”, “containsAny”, “containsAll”, “and”, “or”, “not”,

Complex operators cannot be executed in the database and have to be partly or fully processed by SQEM. Some of the complex operators are dependent on the entity type.

For example, *“Distance”, “Cost”, “Availability”* are examples of **complex** operators. These operators require additional processing after being retrieved from the datastore. The complex operators are discussed in detail in Section 4.3.4.

The simple operators are translated according to the rules, which are presented and exemplified in Table 4.1.

Table 4.1. CDQL to MongoQL translation rules used by SQEM

Simple Operator	Notation	CDQL (SQL-like syntax) example	MongoQL (JSON-based syntax) example
Equality operator	*"="*	*"name" = "John"*	*"name": "John"*
Comparison operators	*"<", ">", "<=", ">="*	*"age" < 60*	*"age": { $lt: 60}*
Containment operator (Any)	*"containsAny"["arg1", "arg2", ..., "arg_n"]*	*"plugType" containsAny ["EU", "AU"]*	*{plugType: { $in: [EU, AU] } }*
Containment operator (All)	*"containsAll"["arg1", "arg2", ..., "arg_n"]*	*"plugType" containsAll ["EU", "AU"]*	*{plugType: { $all: [EU, AU] } }*
Conjunction operator	*AND*	*"height" > 2 and "width" > 1.7*	*{$and: [{ height: { $gt: 2 } }, { width: { $gt: 1.7} }] }*
Disjunction operator	*OR*	*"height" > 2 or "width" > 1.7*	*{ $and: [{ height: { $gt: 2 } }, { width: { $gt: 1.7} }] }*
Logical negation operator	*NOT*	*"colour" not "red"*	*{* colour*: { $ne: "red" } }*

In Table 4.1, a conjunction operator is used to select an entity only when both operands connected by the operator are *'True'*. In MongoQL there are two options for expressing AND operator: implicit *","* and explicit *"$AND"*. The implicit form is more convenient to read by human. However, from the implementation perspective, the explicit option is more convenient and robust. We have chosen to use the explicit operator in most cases, as it requires less complexity in the process of translation.

A disjunction operator is used to choose an entity when any of two operands the side of the operator are *'True'*. A logical negation operator is used to reverse the meaning of the operand. The described building blocks allow us to combine them to convert any incoming CDQL request based on simple attributes to a MongoDB request.

Next, we proceed towards the discussion of **complex operators**. Probably, the most important complex operator for smart city scenarios is the *distance* operator.

Distance – the distance operator is represented as **distance(entityA, entityB, commuteType)**. The *distance* function enables finding the most suitable context providers in case of a geospatial query. The distance function contains three attributes: (i) the location of Entity A, (ii) the location of Entity B, and (iii) a commute type.

The commute type describes the means of commuting between locations. Available options are: (i) linear, (ii) walking, (iii) car, (iv) cycling, and (v) public transport. Technically, the commute type parameter is making the distance operator complex. While it is possible to find a linear distance between objects in a datastore, finding a driving distance requires the usage of advanced external geo-information services. At the same time, calling external services for every registered context provider would cause significant latencies and expenses.

Consequently, to save these resources and reduce the search space, the initial filtering is performed in the datastore. All the entities which passed the initial filter are later checked for more restrictive conditions. The least restrictive condition is the linear distance.

All three attributes are needed when the QE has coordinates of both the context provider and destination and wants to check how far is one from the other using the means embedded in SQEM. In case, when it is needed to search for a list of providers, the location of a provider can be left empty.

The comparison operator "<" after the distance function shows that the distance should not be more than the expression on the right side. MongoDB uses a reversed order of geo points; i.e. [lon, lat]. However, CDQL query accepts a common order of [lat, lon].

For instance, coordinates of Monash University Clayton campus in CDQL are represented as [-37.910408, 145.1345673]

CDQL:

distance([144.49070753102933, -37.655660795303056], [144.49070753102922, -37.655660795303011], "walking") < {"value":1000,"unit":"m"}

MongoQL:

"entrance.location": { $geoWithin:

{ $centerSphere: [[144.49070753102933, -37.655660795303056], 1000 / 6371000] } }

In the expression above, *6371000* is an approximate equatorial radius of the earth, as it is needed to convert meters to radians.

After executing a MongoDB query, each of the results will be fed into a routing engine in order to obtain the walking distance, which is longer or equal to the linear distance. In Figure 4.6**,** an example of the resulting transformation of a CDQL request containing simple operators and a distance operator is presented.

```
prefix swm:http://swm.schema.org,
pull (targetBin.*)
define
entity targetBin is from swm:SmartWasteContainer where
(
    targetBin.Capacity.realTimeCapacity < 90 and
    targetBin.AllowedWaste containsAll ["swm:Plastic"]
    or
    targetBin.Capacity.realTimeCapacity < 80 and
    targetBin.AllowedWaste containsAll ["swm:Organic"]
)
and
distance(targetBin.location,"-37.655660795303056",
"144.49070753102933", "WALKING") < 1000
```

```
db.WasteContainers.find(
    {
        $or : [
            { $and: [ { "swm:Capacity.realTimeCapacity.coaas:contextValue": { $lt: 90.0 } },
              { "swm:Capacity.AllowedWaste.coaas:contextValue": { $in: ["swm:Plastic"] } } ] },
            { $and: [ { "swm:Capacity.realTimeCapacity.coaas:contextValue": { $lt: 80.0 } },
              { "swm:Capacity.AllowedWaste.coaas:contextValue": { $in: ["swm.Organic"] } } ] }
        ],
        "location":   { $geoWithin: { $centerSphere:
          [ [ 144.49070753102933, -37.655660795303056], 1000 / 6371000] } }
    }
)
```

Figure 4.6 – A MongoQL query illustrating the transformation of CDQL request with several simple operators and one distance operator, without taking the expiry into account

Freshness of context attributes - The transformation of a CDQL query is significantly complicated by the necessity to control the expiration of context attributes that are changing in real-time. In Figure 4.7, an example of a simple CDQL query, where only one context attribute taken into account, is provided. This query is designed to retrieve the information about all the waste containers that have the real-time capacity less than 100 per cent. While from the context

consumer point of view the query looks simple and straightforward, the NOD-NOR mode of operation adds complexity in the communication between the CSMS and the QE, which needs further discussion.

```
1  prefix swm:http://swm.schema.org,
2  prefix schema:http://schema.org
3  pull (targetBin.*)
4  define
5  entity targetBin is from swm:SmartWasteContainer where
6  targetBin.Capacity.realTimeCapacity < 100
```

Figure 4.7 - Simple CDQL query with one attribute

The full process of facilitating the NoD-NoR mode of operation is presented as a dataflow diagram in Figure 4.8 to show the complexity of freshness check. The process includes retrieving the available information from the datastore and enriching it with fresh values obtained through the Invoker component of the CoaaS platform. The Invoker is responsible for retrieving information from external context providers on request from internal components of CoaaS, or according to a schedule. The dataflow also includes fusing the retrieved data, returning it to the next stages of processing or a query engine. In the end, the context values that were retrieved by the Invoker are updated in the datastore.

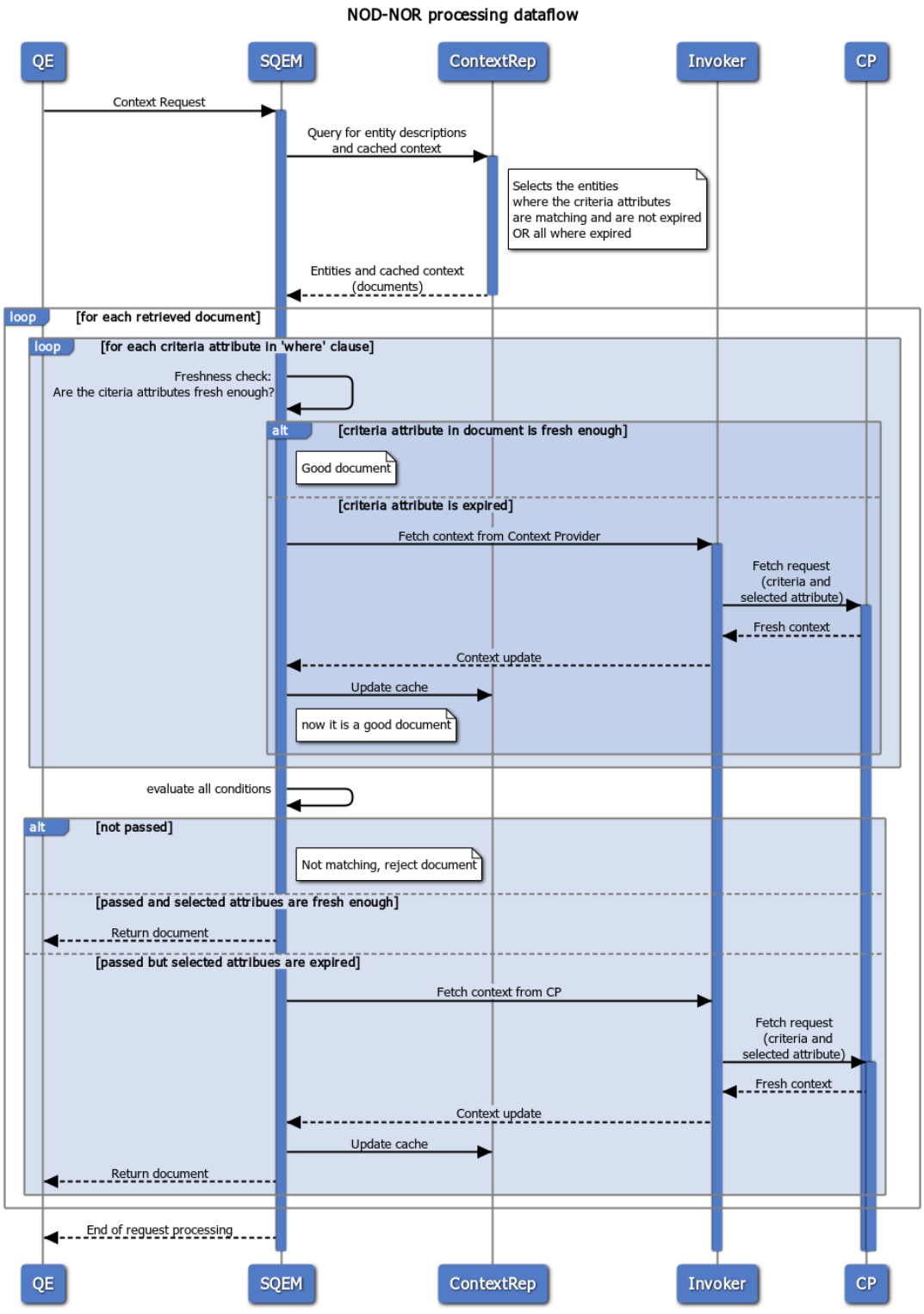


Figure 4.8 – Freshness check in NoD-NoR mode

After representing the whole process of the freshness check in Figure 4.8, we can concentrate on the query, which is generated by SQEM and then executed by the datastore. Due to the JSON-based MongoQL syntax, the generated query is quite verbose and bulky. As

it is hard to read long MongoQL queries, for the description purpose, a simple pseudo SQL representation of the query, which is presented in Figure 4.9, is provided.

```
1  Select * from WasteContainers
2        Where
3        (expiryTime <= NOW() )
4        OR
5        (RealTimeCapacity < 100 AND (expiryTime > NOW() OR expiryTime is null))
```

Figure 4.9 - Pseudo SQL query representing the logic behind the transformation

A pseudo SQL query above represents the logic behind the execution part, which is processed by the datastore. The datastore should return all the bins where the expiry time of the particular attribute has already passed (Line 3), or bins where the real-time capacity is less than 100, and the attribute is considered "fresh enough" (Line 5). The term "fresh enough" means that the expiry time is more than now, or expiry time is not defined. The datastore will also return all the bins which have no expiry time attribute (*expiryTime is null*), as such attributes are considered static (Line 5).

The entities with expired attributes should be retrieved from the datastore for the following reason: it is unknown if the entity matches the query or not. Consequently, SQEM retrieves entities with expired attributes from the datastore and passes them to the Invoker. The Invoker will request the data from the external provider in an ad-hoc manner and pass it to the QE. Then, the Invoker will also update the datastore.

A generated MongoQL query is presented in Figure 4.10. The outer implicit '*and*' clause contains two operands. The first operand states that the field *swm:Capacity.realTimeCapacity* contains an object (*$type: 3*). The second operand contains expressions that represent the logic defined with the pseudo SQL statement above. As can be seen, the generated query does not contain excessive clauses and is formulated effectively.

Due to the complexities added by the freshness check, even filtering by a single context attribute can generate a query, which is not very easy to read for a human. However, filtering can be done using many attributes, and each of these attributes adds complexity to the generated query, to deal with the possible expiry of a context attribute. Such generated MongoQL query is very hard to compose or to be read by a human; however, it is efficient for being read and processed by a machine.

```
db.WasteContainers.find({
 "swm:Capacity.realTimeCapacity": {
   $type: 3
 },
 $or: [
   {
     "swm:Capacity.realTimeCapacity.coaas:metaData.coaas:expieryTime": {
       "$lte": NumberLong(1570602710214)
     }
   },
   {
     $and: [
       {
         "swm:Capacity.realTimeCapacity.coaas:contextValue": {
           "$lt": 100
         }
       },
       {
         $or: [
           {
             "swm:Capacity.realTimeCapacity.coaas:metaData.coaas:expieryTime": {
               $gt: NumberLong(1570602710214)
             }
           },
           {
             "swm:Capacity.realTimeCapacity.coaas:metaData.coaas:expieryTime": {
               $exists: false
             }
           }
         ]
       }
     ]
   }
 ]
})

```

Figure 4.10 – Mongo query generated for the real-time capacity attribute taking expiry into account

With the example above, we have demonstrated how the transformation of a CDQL query in SQEM helps to avoid the tedious task of composing long queries, and having complex processing pipelines, for a developer of a context consumer application. Moreover, these transformations facilitate the possibility of CSMS to work in the NoD-NoR mode.

4.3.2 CDQL SITUATION FUNCTION TRANSFORMATION

In this section, we demonstrate how a CDQL situation function definition is translated into a stored format by SQEM. In Section 4.2, we introduced the '*situWeather'* function, which contained the definition of the *'goodForWalking'* situation. The call to a situation function is shown in Figure 4.2, and the definition of the situation function from the user (CDQL) perspective is presented in Figure 4.3.

Figure 4.11 - Stored CST function

In Figure 4.11, a stored definition of a situation function is presented. To reduce the size of the example, less important parts are collapsed. The main sections of a document that define a situation function are: (i) *ObjectID*, (ii) *functionTitle*, (iii) *nameSpaces*, (iv) *relatedEntities*, and (v) *Situations*.

The *ObjectID* is used as a unique identifier of the document in the datastore. The *functionTitle* is defining the name of the function, *nameSpaces* contains a list of semantic vocabularies used in the definition of the function, *relatedEntities* contains the mapping of the related entities, and the *Situations* section contains the list of situation definitions.

An expanded view of related entities is presented in Figure 4.12. In this case, the *relatedEntities* section contains only one mapping *r1*. This mapping is used in the definition of a situation to map the input values of an entity, (type "*Weather*" defined by the schema.org vocabulary), to the attributes, which are used in the definition of a situation.

```
8     "relatedEntities" : {
9         "r1" : {
10            "type" : "weather",
11            "vocabURI" : "http://schema.org"
12        }
13    },
```

Figure 4.12 – Related entities section of a stored situation function

The definition of the stored 'goodForWalking' situation contains a list of attributes, which are mapped to the input entities (e.g. *airTemperature*). Each attribute has a name, a list of ranges, and a *weight*. Each range contains a definition of a range and *Belief*. The definition of a range is a text field, the type of used braces (round or square) define if the point on the corresponding end of a range is included or excluded.

Figure 4.13 shows a schematic dataflow, which illustrates how a request with a situation function is processed.

When a situation function is called in a CDQL request, SQEM retrieves the definition of a function by name, parses the definition to find the relevant ranges for each attribute, puts the values of corresponding beliefs and weights into a call to a CST reasoner (ECSTRA), and finally, receives the likelihood of a situation occurrence, (i.e. a value between 0 and 1), based on the definition and input context attributes [36]. Then, the likelihood is compared to a value given in a query, and the final Boolean value for the RPN condition is obtained.

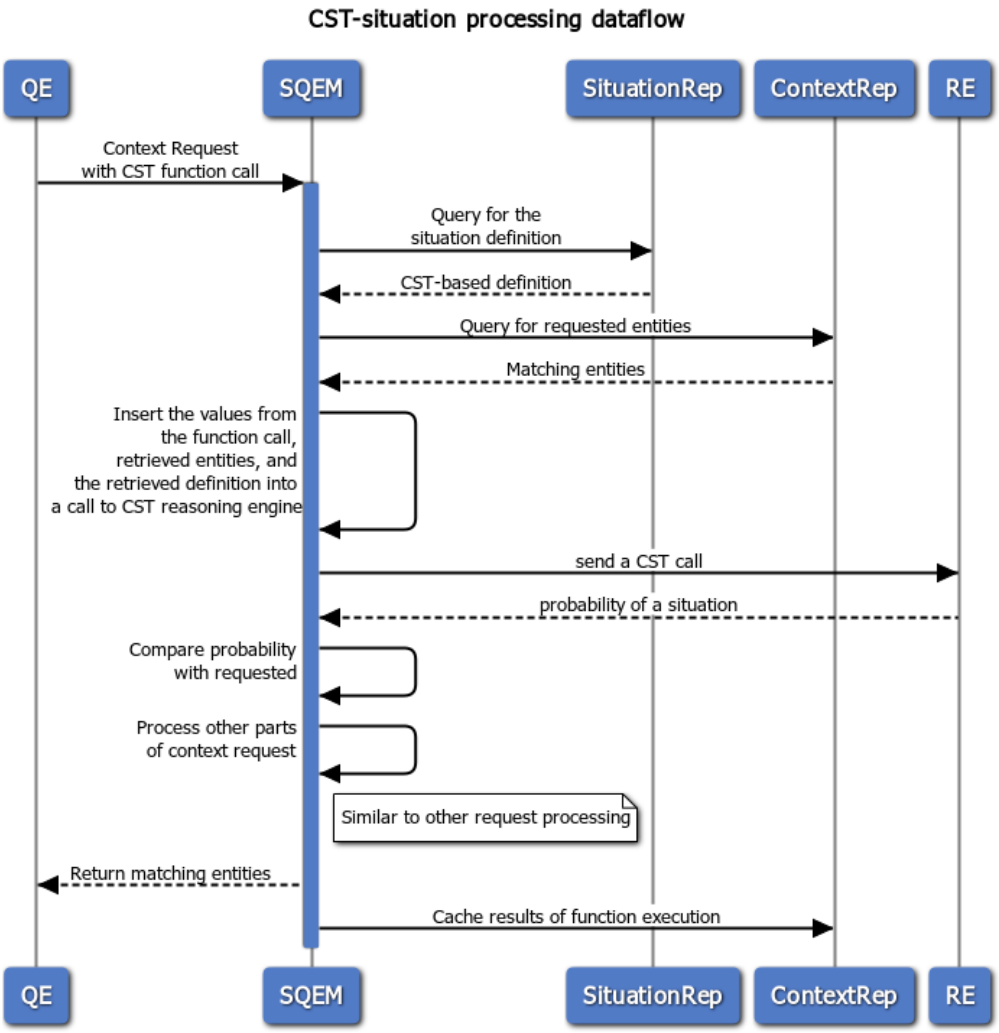


Figure 4.13 – Dataflow of situation function processing

4.3.3 SQEM POST-RETRIEVAL CONTEXT PROCESSING

In this section, we discuss the next stage of context processing, that is SQEM post-retrieval context processing. The term ‘post-retrieval’ in this section means ‘after context was retrieved from the Context Repository (CR)’.

In Section 4.3, we provided details about how contextual entities were processed in SQEM during the freshness check. The datastore returns the documents in which the attributes used as selection criteria are not expired and relevant, or known to be expired. The reason for this is that when the cached attribute is used for filtering, and the attribute is expired, it is impossible to understand if the document is relevant or not. Consequently, the document should be retrieved from the datastore and further processed for making the final decision: to determine if the document matches the selection criteria or not. In general, there are two types

of attributes from the query perspective: (i) attributes which are used for filtering, (defined by the *where* clause), and (ii) attributes that are used for returning to the consumer, (defined by the *select* CDQL clause). For example, documents can be filtered based on the location of an instance, but the returned attribute is the current velocity of the instance. Technically, both attributes can be expired, and the expiration period can be different. Further, if the attributes used for initial filtering are not expired and the document is considered worth being returned to a consumer, it may appear that the attributes that will be returned to the contained in the document are expired. Consequently, these attributes need to be refreshed before being returned to the consumer.

We also described the processing of CST-based situation functions (Section 4.3.2), which require retrieving documents from the context storage and injecting the retrieved data into the call to a CST reasoning engine.

There are several other types of post-retrieval processing, which need to be highlighted, namely (i) the **build-in functions** and (ii) **custom functions**. Both types of functions belong to a class, which is called **computation functions.**

The **built-in functions** are either embedded in the main functionality of the platform directly or make use of pre-defined external services. Examples of embedded build-in functions are *avg(), max(), min().*

An example of a built-in function which is relying on an external service is the *distance()* function with an argument other than '*linear'.* For instance, if the CDQL request contains a condition as follows: *distance(car.location, Carpark.location, 'waking') < 300m,* then the initial filtering of related instances is performed in the datastore by applying the distance function with a linear argument. Next an external router is used to obtain the walking distance to each carpark, and then the post-processing is applied to a set of documents, in order to filter out those where the walking distance is not matching the given criteria.

A **custom computation function** is a function defined by context consumers, or the DevOps team responsible for a particular CoaaS platform deployment. A custom function can be defined as an aggregation function over a set of documents, or it can be defined over one document. In any case, the aim of a custom function is to process specific fields of the retrieved documents and return the specific result, instead of returning the whole document.

The custom functions are deployed as a RESTful service. During the processing of a request that contains a custom function, CSMS sends the initially retrieved documents to the

corresponding RESTful service and uses the received results for further processing of the request. As an example of a custom function, consider the *parkingAvailability()* function, which can be used for simple, (from CDQL user perspective), filtering of parking facilities. The possibility of using a parking spot is often a complex issue, where the decision if a particular driver can use the spot, depends on the status of the driver, (e.g. has a permit, has limited mobility, etc.), the time of the day, the day of the week and other factors. These factors can be area-dependant. All these factors are represented in the document, which is describing the carpark, according to a corresponding ontology (semantic vocabulary). However, a developer of an external application might prefer to use a custom function like '*parkingAvailability(targetCarpark, 1:30PM, 3:40PM) = True'* in the query, instead of developing the code for processing of a complex structure in the application.

Another example of a custom function is the *costOfParking()* function. The parking pricing can vary based on the time of the day, day of the week, and the type of a vehicle (e.g. electric car). A CSDL-definition of a carpark entity contains a complex structure for defining the pricing. For instance, a CDQL request with a function which will significantly simplify the job of a developer will look like:

costOfParking(targetCarpark, targetCar, 1:15PM, 4:30PM).

Hosting custom functions at the platform's side brings not only the advantage of query simplicity. It also enables the option of caching at the level of functions, (discussed in Chapter 5). Moreover, it also enables the possibility for optimization, as it reduces the amount of data transferred to the client application and back [16].

In the next section, we present the discussion of another processing enabler, which is essential for PUSH-based queries and situation monitoring.

4.3.4 CDQL-SIDDHI TRANSLATOR

One of the main functionalities that is embedded in CDQL to serve the situation monitoring is the support of window functions. These functions are used to detect trends, which data sources exhibit during a certain period. The period of time is usually defined as a sliding window (e.g. last 30 min), where the beginning of a period is some point in the past defined as $t_{beginning} = t_{now} - Window_{size}$, and the end of the period is the current moment. In this case, as time goes by, some of the data points are falling out of the scope. At the same time, new data items may arrive. The difference between window functions and other types of

functions is that the result is influenced by both time and event flow. This means that SQEM cannot just call a function and obtain a resulting value. In such cases, it would be an analytical function over a recorded dataset. The window function, on the contrary, is always "loaded", which means it runs continuously and, consequently, consumes computational resources that need to be competently managed.

CDQL supports the *increase()*, *decrease()*, and *isValid()* functions. In Section 4.2 (Figure 4.4), we have used the *decrease()* function to detect the moment when the driver is heading towards the car. In this scenario, we want to capture when the distance between the driver's location and car's location is decreasing. The incoming data is a stream of events with the driver's location, which keeps changing. Without CDQL-specific details, the function can be defined as follows:

Decrease(distance(driver.geo, location.geo, "walking"), 5min)

The window function above returns '*True'* if the distance between two locations decreases during the last five minutes.

There can be numerous complexities in realising such type of functions, including issues like data fluctuations smoothing, memory and resource management, missing events and wrong order of events delivery handling. Moreover, trend prediction techniques can be applied to achieve advanced functionality. The type of software that provides the required functionality to support window function execution, is usually referred to as Data Stream (DS) processors or Complex Event Processors (CEP). The field is actively growing and offering more advanced functionality, as well as the complexity of defining the stream monitoring tasks.

As the task of trend monitoring is complex, a decision has been made not to re-implement the basic functionality, but to effectively reuse an existing open-source framework. The WSO2 Siddhi Complex Event Processing Engine[14] was chosen and integrated as a library into CSMS. This approach allows us to use available advanced functionality, as well as add new functionality once it becomes available in future.

According to CDQL aims discussed in Chapter 3, the main interface for the developer of a context consumer application should be kept (i) simple but flexible, (ii) unified, (iii) agnostic of underlying technologies. That means we had to find a way to connect the features available in CDQL to the capabilities of a streaming platform. We limited the consumer's direct access

[14] https://wso2.com/products/complex-event-processor/

to a streaming platform and configuration of the data stream monitoring, using the native syntax of the streaming platform in order to keep the consistency and simplicity of CDQL, as well as for the security and access control reasons.

In a similar way as it has been done with the storage component, a query translator was created, which connected the functionality supported in CDQL to the functionality embedded in the Siddhi CEP engine.

Siddhi CEP has an advanced but not very simple language called *Siddhi* for defining the stream monitoring tasks. Every monitoring task is formulated in the form of so-called Siddhi application, which is registered in the CEP engine. The aim of an SQEM Siddhi translator is to generate a Siddhi application or several applications based on the window functions defined in a CDQL query. A representation of the *decrease()* function, which was converted to a Siddhi application, is presented in Figure 4.14.

In this case, CSMS had to monitor the continual decrease of the distance between two objects during a defined period. The incoming events contained the geo-coordinates of a moving object and the timestamp of message sending moment. As the window function contained a *distance()* function inside, at first a pre-processor computed the distance between two coordinates, (coordinate of the second object were retrieved from the storage or external provider), and the distance was obtained. In the case when the non-linear distance parameter (e.g. Walking distance) was used in a CDQL query, an external router was applied to obtain the distance. Eventually, the stream of new events that contained a pre-computed distance was routed into a Siddhi application.

```
1  @app:name('sub_123')
2  define stream eventStream(name string, amount double, timestamp long);
3  define table eventResultTable (functionSignature string, initialAmount double,
4  finalAmount double,timestamp long);
5  partition with (name of eventStream)
6  begin
7  from every e1=eventStream,
8  e2=eventStream[e1.amount > amount and (timestamp - e1.timestamp) < 10 * 60000]*,
9  e3=eventStream[timestamp - e1.timestamp > 10 * 60000 and e1.amount > amount]
10 select 'functionSignature' as functionSignature,  e1.amount as initialAmount,
11 e3.amount as finalAmount, e3.timestamp insert into eventResultTable;
12 end;
```

Figure 4.14 – A generated Siddhi application for distance decrease monitoring over a sliding window

The *'@app:name'* field (Line 1) links the Siddhi application to the definition of a subscription, where the current windowing function is used. At Line 2, the format of the

expected stream is defined, where the '*amount'* field represents the '*distance'* from the incoming event. Line 3 defines an event result table, where the results of monitoring will be stored. Lines 5 to 11 define the logic, which allows us to compare the incoming parameter with all the previous distance parameters during the defined window size.

As it can be seen, defining simple monitoring tasks requires a great deal of effort in Siddhi. Moreover, event pre-processing, which requires interaction with CoaaS to retrieve missing parameters, also adds complexity.

In this section, we showed how translating a CDQL window function call into a Siddhi application for stream monitoring was achieved. Window functions can only be used in the *'WHERE'* clause of PUSH-based queries. The description of a dataflow for serving the whole PUSH query is presented in Section 4.6.

4.4 CSDR REPOSITORY IMPLEMENTATION

Context Service Description Repository (CSDR) is a place where the information about context services is stored. Context services are the external endpoints that provide context about a particular entity instance. Each context provider can have one to many context services. The incoming CDQL query can request for context from entities based on any information about the entity. Some parts of this information are static (e.g. type, ID), some are semi-static (e.g. IP address), and some are transient (e.g. location of a vehicle). The query can search for context based on a known ID of a vehicle. Then the search for a provider is based on its static information. However, if the query looks like "*return all the vehicles around location X*", then the task relies on the transient data. This means the entities, which were within the area a minute ago, might not be around at the moment of query execution.

The core of the context service definition consists of (i) the entity identifier, (ii) entity type, (iii) the endpoint address, (iv) the format of a request (from CoaaS to the provider), and (v) the format of a response (from the provider to CoaaS). In general, these components are essential to enable the communication between CoaaS and the service provider. However, the actual context (real-time data) also defines the entity during the search. For this reason, in the current implementation of CSMS, CSDR is merged with the repository of current context, which is described in the next section.

4.5 CONTEXT REPOSITORY IMPLEMENTATION

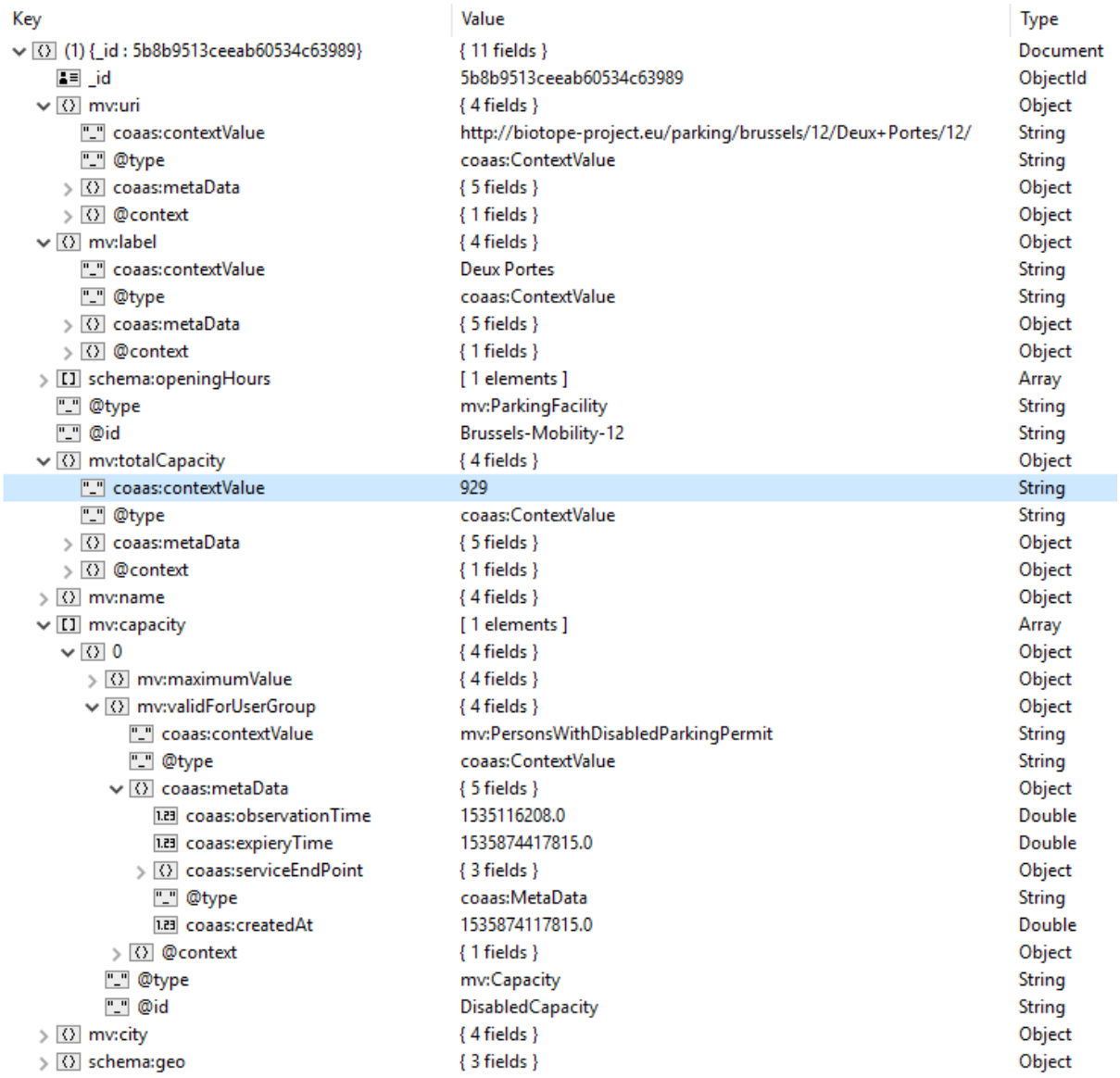
In this section, the implementation of the Context Repository module (CR) is presented. CR is the central part of CSMS and enables several features, which are crucial for the CoaaS platform. CR can be viewed as a datastore, which contains static data together with transient IoT data. Regarding the changing (transient) data, CR can be considered as a cache, which contains the last values of context attributes. These attributes were retrieved from external providers. The transient attributes are annotated with the timestamp of fetching from the external providers and the expiry period. The expiry period is used for making a decision if the value can be considered fresh enough at the moment of a CDQL request arrival, or the value needs to be refreshed before making a decision about returning it to a consumer.

Each entity instance is structured according to a chosen semantic vocabulary, and each field, in turn, is annotated with a CoaaS metadata. Figure 4.15 represents an instance of a *ParkingFacility* entity that is structured according to a MobiVoc semantic vocabulary. We have adopted JSON-LD structure for storing the description of the entity.

Each entity is represented by three main attributes, which are *@id* and *@type*, and *Attributes.* The *@id* field is a unique identifier of the entity instance. The *@type* fields links the entity with the corresponding semantic vocabulary structure. The *Attributes* field is an array of context attributes associated with the entity. Each context attribute has four sub-fields, which are *name*, *@type*, *value,* and *metaData.* The *name* field is the title of the attribute (e.g. capacity), the *@type* field is a type of the value of the attribute, and the *value* field contains an actual value. The *metaData* provides additional information about context attributes, containing such fields as expiry time, observation time, and accuracy of measurement.

The context repository is not only a cache of low-level context attributes. Once a query is served, the results can be also cached for later reuse. The initial query string is converted into a hash, and the result of a query execution is stored next to the query hash. The result of query execution is also associated with the expiry time of the context attribute, which was used for filtering or returning, and which has the shortest expiry period, compared to other used context attributes. In other words, the time of expiry of this attribute is the closest to the moment when the result was put into the cache. If the same query (same hash) arrives before the expiration, the cached result is reused instead of executing the whole process. Caching and reusing the whole query is the most efficient scenario; however, it is the least likely at the same time. For that reason, lower level parts of the query execution, which might be repeated in other

queries, are also cached, following the same principle of the shortest expiry period. These lower-level parts are CDQL requests (a single entity query), CST functions and computation functions. The lowest (base) level is the level of raw context attributes, which were already introduced in the first part of this section. The detailed description of CSMS cache levels is presented in Chapter 5.



Key	Value	Type
(1) {_id : 5b8b9513ceeab60534c63989}	{ 11 fields }	Document
_id	5b8b9513ceeab60534c63989	ObjectId
mv:uri	{ 4 fields }	Object
coaas:contextValue	http://biotope-project.eu/parking/brussels/12/Deux+Portes/12/	String
@type	coaas:ContextValue	String
coaas:metaData	{ 5 fields }	Object
@context	{ 1 fields }	Object
mv:label	{ 4 fields }	Object
coaas:contextValue	Deux Portes	String
@type	coaas:ContextValue	String
coaas:metaData	{ 5 fields }	Object
@context	{ 1 fields }	Object
schema:openingHours	[1 elements]	Array
@type	mv:ParkingFacility	String
@id	Brussels-Mobility-12	String
mv:totalCapacity	{ 4 fields }	Object
coaas:contextValue	929	String
@type	coaas:ContextValue	String
coaas:metaData	{ 5 fields }	Object
@context	{ 1 fields }	Object
mv:name	{ 4 fields }	Object
mv:capacity	[1 elements]	Array
0	{ 4 fields }	Object
mv:maximumValue	{ 4 fields }	Object
mv:validForUserGroup	{ 4 fields }	Object
coaas:contextValue	mv:PersonsWithDisabledParkingPermit	String
@type	coaas:ContextValue	String
coaas:metaData	{ 5 fields }	Object
coaas:observationTime	1535116208.0	Double
coaas:expieryTime	1535874417815.0	Double
coaas:serviceEndPoint	{ 3 fields }	Object
@type	coaas:MetaData	String
coaas:createdAt	1535874117815.0	Double
@context	{ 1 fields }	Object
@type	mv:Capacity	String
@id	DisabledCapacity	String
mv:city	{ 4 fields }	Object
schema:geo	{ 3 fields }	Object

Figure 4.15 - Context repository entity instance definition

The components presented above form the CSMS framework for serving the PULL-based queries. In the next section, we focus on the essential components for serving the second type of CDQL queries, which are the PUSH-based queries.

4.6 SUBSCRIPTION MODULE IMPLEMENTATION

In this section, the implementation of the Subscription Module (SM) is presented. SM is designed to facilitate the execution of PUSH-based queries in the CoaaS platform. An overall architecture of the Subscription Module (SM) is presented in Figure 4.16.

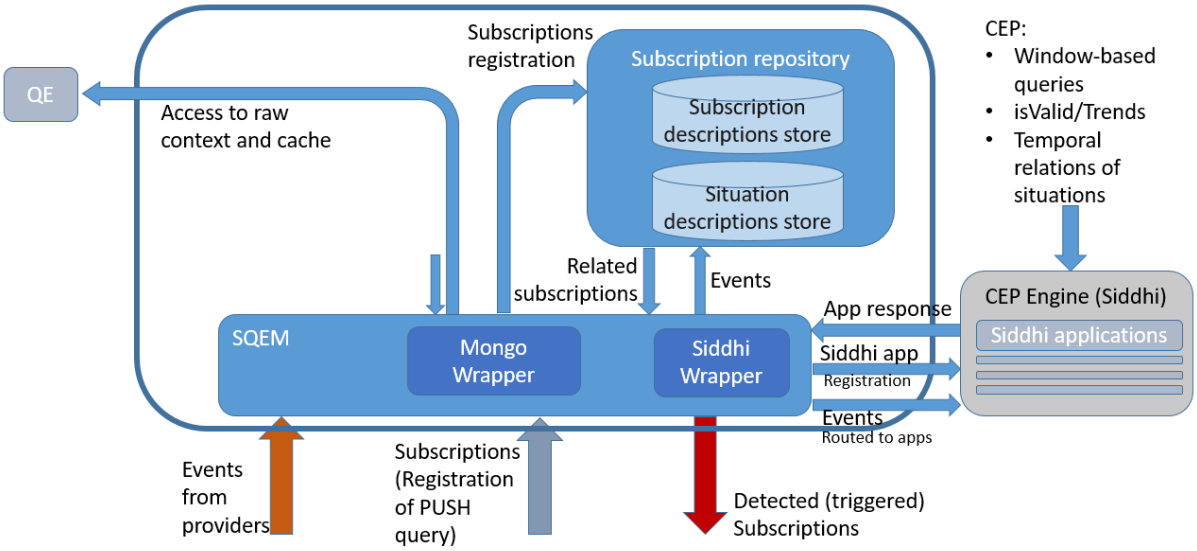


Figure 4.16 - Subscription module implementation

In Section 4.2, a preconditioning scenario is described, in which a connected car was informed by the CoaaS platform that a driver was walking towards the car, and he/she was likely to use it. When this situation was detected, the preconditioning procedure started to adjust the temperature and other internal parameters of the car for an immediate comfortable departure. A corresponding CDQL query is presented in Figure 4.4.

The Subscription Module, as well as other CSMS modules, is governed by SQEM. The main aim of the situation module is to inform the CoaaS Query Engine (QE) when the conditions defined in a *'WHEN'* clause of a PUSH-based query are met. After it happens, a corresponding CDQL query is executed in a similar way to a PULL-based query execution process. In the next paragraphs, the process of facilitating the monitoring of subscriptions and the main dataflow happening in CSMS is described.

The process starts with registration of a new subscription arriving from the QE side. In the use-case, several conditions should be met to trigger the subscription. In Section 4.3.4, we have already described the process of handling one of the conditions, which is a window function. The *decrease(distance(), 5min)* function returns *'True'* if the distance between the driver and the car is decreasing during the last five minutes (Figure 4.4, line 8). Other

conditions, which should be met to trigger the subscription, include the following circumstances. There should be an appointment for the meeting in the driver's calendar, and the time left until the start of the meeting is less than 50 minutes (line 3). The meeting should be located too far to walk from the driver's location (more than 2000 meters, line 4). The car should be parked within walking distance from a driver (less than 500 meters, line 5), and the conditions of the environment around the car are out of the preferred range (line 9). All these parameters should be continuously monitored.

In Figure 4.16, the flow of subscription registrations is depicted as a blue arrow in the mid-bottom part of the picture. During the situation registration procedure, a restructured subscription definition is placed into a subscription store. In Figure 4.17 the structure of a stored subscription definition is presented.

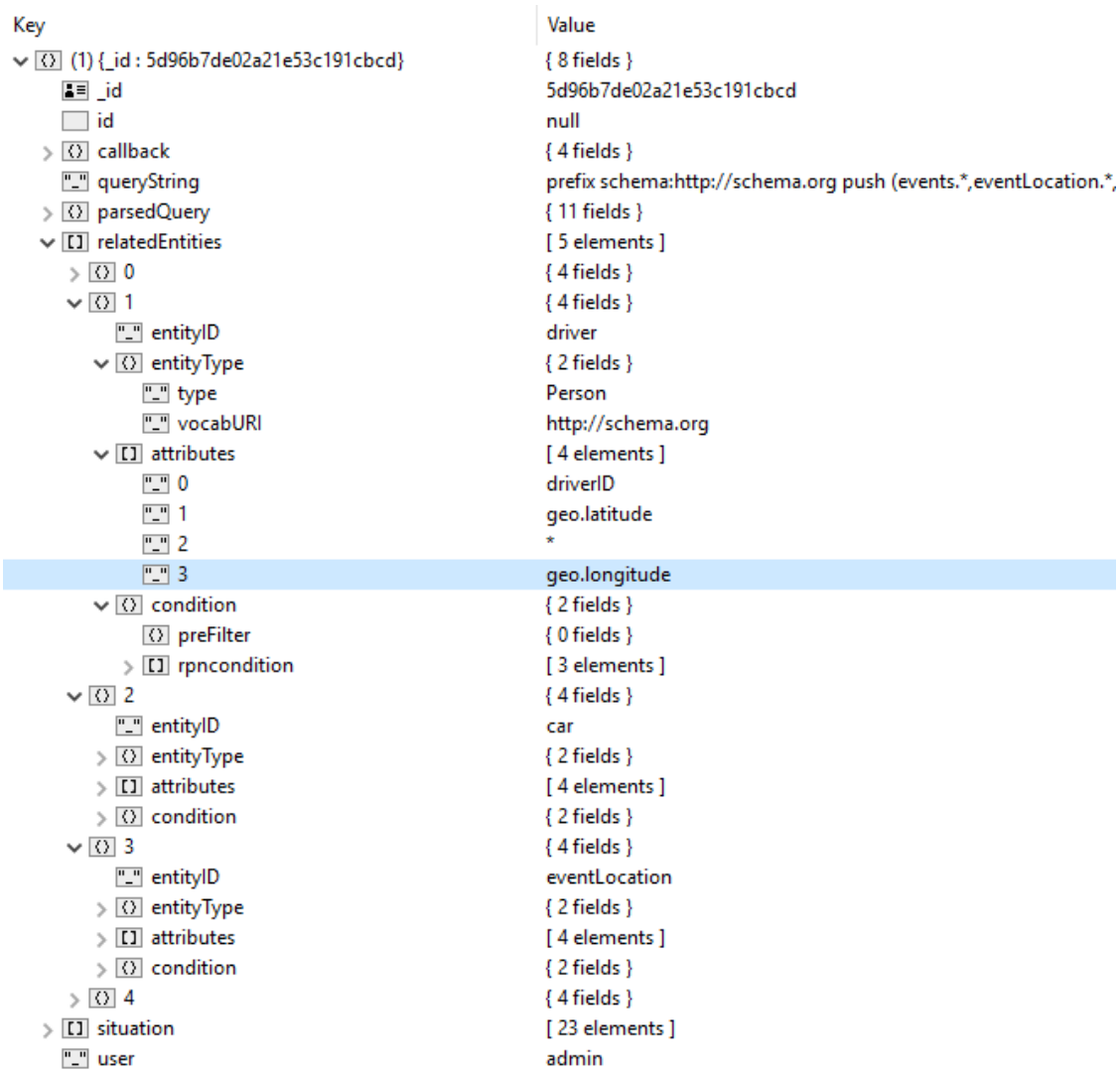


Figure 4.17 - Structure of a stored subscription definition

The full subscription description is stored in JSON format and contains 3205 lines for the defined scenario. Due to space considerations, in Figure 4.17, the structure is presented in a tree format, where most of the sections are compressed. However, it shows the most necessary parts. These parts are the callback, query string and parsed query, related entities, situation and the user. The most important part is the *relatedEntities* section. It contains the information that the entity *driver (Type:Person* defined in *schema.org)* is related to the subscription. Consequently, all the incoming messages, which change, for instance, the location, and are relevant to the entity *driver* with a certain *DriverID*, trigger the retrieval of this definition into SQEM and processing of the situation.

Other important notions for the subscriptions module are the *messages* and *events*. Here the assumption is made that all the involved entities send messages to CoaaS every time anything changes, for instance, the driver's smartphone sends the location when a driver starts walking, or the car sends the temperature measurements when the temperature changes. To achieve this, CoaaS retrieves needed contexts from external entities periodically, or according to a plan developed by the proactive cache management component. In any case, these context updates come in the form of messages that need to be processed and fused in order to understand if the conditions of the *'WHEN'* clause are met or not. These events are routed to SQEM and then processed in the situation module. In Figure 4.16, the stream of incoming events is depicted as dark red arrow in the bottom left corner of the picture.

When the event arrives at SQEM, a MongoQL query is generated to the subscription description store to check if there are any related subscriptions.

A sequence diagram of the event processing chain is presented in Figure 4.18. A generated query to find subscriptions related to an event is presented in Figure 4.19. The query is generated by the SQEM Mongo wrapper, similar to the process of query generation for the Context Repository (CR), which was described in Section 4.3.1.

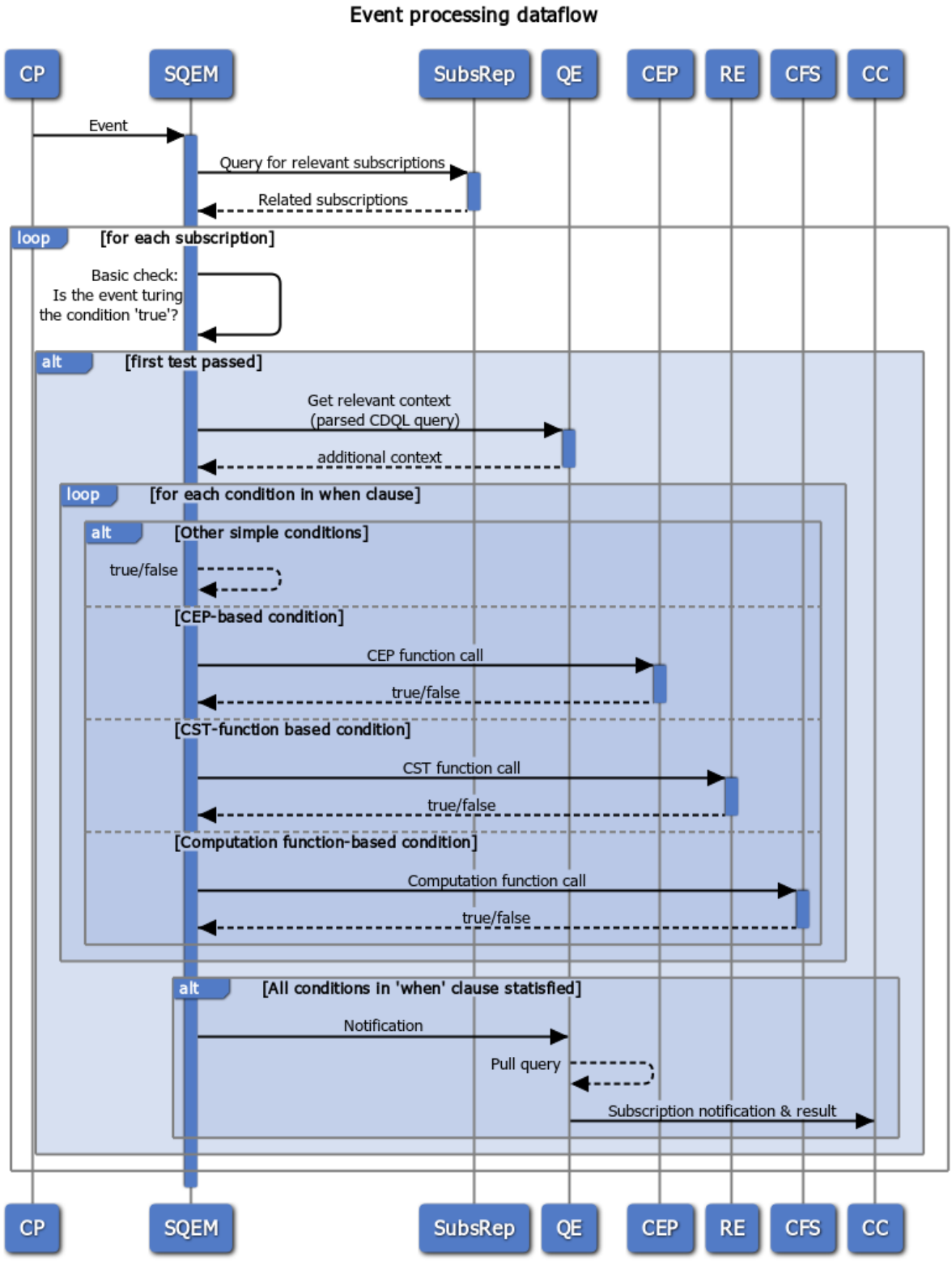


Figure 4.18 – Event processing sequence diagram

```
db.getCollection("subscriptionsts").find({
  "relatedEntities" : {
    $elemMatch:
    {
        "entityType.type" : "Person" ,
        "entityType.vocabURI" : "http://schema.org" ,
                "attributes" : {$in : ["geo.latitude","geo.longitude"]},
                "id":"biotope"
        }
   }
} )
```

Figure 4.19 – A generated query to find related subscriptions

In the case when a subscription registration related to the incoming event is found in the situations description store, the description is returned to SQEM.

The situation definition can be of any level of complexity, which is allowed by the CDQL 'WHERE' clause. The description can include a trend function (stream processing, windowing), a CST-function (context spaces situation definition), computational function, as well as other simple operators, which were described in Section 4.3.1.

If a situation description contains a stream processing function (or several functions), the event is routed to the CEP Engine's corresponding registered stream processing application (Siddhi app). In a case when the stream application returns '*True'*, the processing continues.

If a situation description contains a CST function, the request is sent to the ECSTRA-based reasoning engine, to estimate the result of the CST function. First, the definition of a CST function is retrieved. Then, this request has to be sent through the QE, (event and CST), as there may be other entities and context attributes involved in the definition of a CST function. If one of the conditions is not satisfied, the data from the event is used to update the current value of a corresponding context attribute in the storage.

Eventually, when an incoming message changes one of the conditions from the '*WHEN*' clause to '*True*', and all the other conditions are found to be met, SQEM sends a detected situation message to the QE. The message contains the parsed PUSH query from the situations description store. As the situation is already detected, the QE can act in the same way as with a PULL-based query. The QE uses the '*SELECT*' clause from the query to get the required context from the storage and sends the assembled final document to the context consumer.

4.7 CONCLUSION

In this chapter, we presented the design and implementation of the CSMS modules, which are necessary for the functioning of the CoaaS platform. We presented the architecture of each module and discussed the dataflow between components. Along with the discussion of the dataflow, we provided the evaluation of the transformation of CDQL queries into queries to the underlying data storage layers. We showed how existing available data storage and processing solutions that do not require licensing, can be effectively used to facilitate the process of large-scale context query serving and situation monitoring. While the CDQL query is designed with an aim to hide the complexity from the end-user, there are a lot of processes happening between the query engine and CSMS to serve the query. For instance, significant complexity is brought by the possibility to serve queries relying on the data, which is unavailable or expired in the external storage. While such functionality can bring significant flexibility and cost-efficiency in the process of CMP operation, the complexity of internal queries to data stores as well as the interaction between the modules of CoaaS grows, finding effective ways to harness the integrated data storage and processing solutions are required.

We showed the designed and implemented mechanisms, which facilitate such functionality as: (i) serving context requests from the query engine and retrieving current contextual information, (ii) discovery of context providers, (iii) registering and processing CST-situation functions, (iv) registering and processing window-based functions, and (v) processing the event streams for situation monitoring. Moreover, the workability of these mechanisms was illustrated on real smart city use cases and queries.

Chapter 5: CSMS caching approaches

5.1 INTRODUCTION

In this chapter, we focus on Context Storage Management System (CSMS) caching strategies and mathematical models, which support these strategies. The overall challenge faced by the Context as a Service (CoaaS) platform is processing and managing enormous amounts of context stemming from IoT context providers and other data sources. As it was mentioned in previous chapters, self-adaptation is the most critical factor for efficient CSMS operation.

In Chapter 2, we have described the Not only Database, Not only Redirector (NOD-NOR) principle of operation, which is used as the first fundamental principle of current research. For a platform operating in NOD-NOR mode, the core of self-adaptation is the decision about what data should be kept in the cache for serving queries and what data can be retrieved from external context providers in an ad-hoc fashion. Consequently, we can state that efficient proactive cache management is one of the impending challenges for CSMS.

The second fundamental principle of the current research is the cloud-based deployment of a Context Management Platform (CMP). In the past, the design of adaptation algorithms was aimed at optimising the performance of a fixed-size system or, alternatively, to find the right size for the needed system. Examples of this approach are LRU, LFU [196], and other methods described in Chapter 2. However, with the rise of IaaS and PaaS business models [197], it became possible for an operation team to quickly scale the system to suit the changing requirements. For instance, an administrator can promptly allocate several more cloud servers for a NoSQL database and reconfigure the load balancer for using these additional servers as shards. By this, an administrator can achieve faster query execution time or a higher number of parallel queries served per second. The described approach is called horizontal scalability.

In a cloud system, achieving higher performance will come at the cost of paying for these additional resources (e.g. servers, storage). However, when the need to serve an additional amount of queries disappears, an administrator can deallocate some of the cloud servers to reduce the cost of operation. If the pattern of incoming load (the number of queries in a particular time of a day) is known, the process of allocation and deallocation can be easily automated. Obviously, not all computational tasks can be easily scaled. For instance, operation of RDBMS is well known for issues with horizontal scaling. As it was described in Chapter 4,

we have designed the CoaaS modules in a way that is compatible with the concept of horizontal scalability to enable the platform to operate in cloud environments and be easily adaptable for varying loads. Adapting the approach in this way brings us to an understanding that the cost of caching (the use of cloud resources and external services), together with meeting the necessary constraints will be a crucial factor in deciding on the caching strategy.

We have already defined two main concepts used for the caching model: (i) the NOD-NOR principle and (ii) the non-fixed size nature of cloud deployment. Now we can move to the third principal component, which is (iii) the *transient nature of IoT data.*

IoT data consists of a large number of relatively small in size *data items*, which we also interchangeably call *context attributes*, as these raw attributes are used for deriving higher-level context. The smallest possible data item is a measurement with a timestamp coming from a remote sensor. An example of such minimalistic data item (context attribute) is presented below:

{

“sensorid”:”1068a0614803”,

“value”:”44.17”,

“timestamp”: “1524810064”

}

As the sensor identifier is known, we can obtain information about the type of data, which a particular source is producing, as well as other semi-static metadata (e.g., precision, owner). While managing semi-static data is not a big problem, the IoT data itself should be looked at from a closer perspective.

The main difference between IoT data and other types of data is that IoT data changes over time, or, at least, it can change. For instance, a car can change its location with time when the vehicle is used, or, the location will remain the same while the vehicle is parked. In this case, the location is an example of a continuously changing attribute. An attribute can also be binary; for instance, the occupancy of a particular parking spot. An attribute can also take a value from a defined set (e.g. red, yellow, green). The critical point is that IoT data can become obsolete at any time. A newspaper article or a video file, on the contrary, are examples of non-IoT data. Such data will not usually change over time. If we look at IoT data from another perspective, very strictly, we can say that any IoT data which is returned to a consumer can

become obsolete, when it takes at least several milliseconds to be transferred from a provider to a consumer. During these several milliseconds, a real value (in a sensor) could already be changed. In the best (and the most expensive) case, the speed of data transfer is limited by the speed of light.

However, a consumer is not interested in obtaining a value which absolutely matches reality, as, strictly speaking, absolute matching is impossible in general. Usually, a consumer is satisfied with some level of confidence. As we move from real-time critical systems towards smart city cross-domain scenarios, which inspire the development of CoaaS (Chapter 3), we can see that the level of needed confidence is reduced. For example, it is enough to know that a car is moving in a specific area and direction, without knowing the exact place, or it is enough to know that in a certain parking area there are about a hundred available parking spots, and there is no need to know when a value changes from 100 to 99 vacancies, or a specific spot is occupied. In time, a data item will lose its *freshness*. However, it will remain useful for many or some use cases, and every use case has to define its need individually. The discussion above brings about two more valuable questions: (i) how to assess the freshness of a cached context attribute and (ii) how to negotiate the needed level of freshness between a CMP and context consumers.

In the area of estimating the freshness of IoT data in Named Data Networks (NDN), a significant research effort was undertaken by Meddeb et al. [169][170]. In general, there is a considerable body of knowledge for time series analysis and prediction (refer to Chapter 2). These techniques, however, are not the focus of our work. The assumption made was that we could reuse existing techniques for each type of context attribute to estimate the speed of freshness loss. Instead of focusing on determining the freshness loss, we made the assumption that the speed of freshness loss is known and focused on the estimation of platform's operation cost under variable CDQL-query load, with respect to the varying requirements to context data freshness.

Negotiating the level of freshness and other Quality of Service (QoS) and Quality of Context (QoC) parameters between context consumers and a CMP is usually done by defining the Service Level Agreements (SLAs). While commercial cloud IoT platforms have defined the usage of resources in their own terms for their purposes (refer to Chapter 2), the definition of SLAs for CMPs should be based on different principles and is still in its infancy. However, some efforts have already been made [198]. Nevertheless, we can create a minimalistic SLA by defining just four main parameters: (i) maximum time of serving a request, (ii) minimum

level of freshness (iii) price of a request, and (iv) penalty paid in case when a request is not served within a defined time. We provide a more detailed description of each SLA parameter further in this chapter.

To sum up the discussion above and turn it into a practical CoaaS cache management framework, we need to fuse the objectives, pursued by a CMP, with the three main principles of CSMS operation: (i) NOR-NOD mode, (ii) non-fixed size of a system and (iii) SLA definition based on freshness, where multiple levels of SLAs are possible.

The main objective is to minimise the cost (or to maximise the profit) of a CMP operation. In the long run, the cost of serving incoming CDQL queries must be lower than the revenue acquired from the consumers. All other objectives are subordinate to the main objective and are reflected in the SLAs with consumers. These secondary objectives are: (i) reducing the time of serving queries and (ii) keeping the acceptable level of the QoC, which is, foremost, dependent on freshness.

The main aim we were pursuing while researching the cache management strategies was to find an answer to each of the following questions:

- How to define efficient usage of cached data in CoaaS? (Problem statement)
- What are the available strategies we can apply to caching at the level of context attributes? (Choice of strategy problem)
- When do we need to refresh a context attribute to achieve the peak efficiency? (Proactive caching problem)
- Can we apply caching at a higher level than at the level of raw data items? (Multilayered logical cache problem)
- Can the presence of different levels of cache influence the model? (Physical levels problem)

The rest of the chapter is organised in the following way: in Section 5.2. we define the primary approach to efficiency. In Section 5.3, we describe our approach to logical separation of cache levels, while in Section 5.4, we present another dimension of cache separation – the physical levels. Next, in Section 5.5 we describe the main parameters which influence cache decisions and define the main caching strategies which can be used. Section 5.6 describes the

model for cache management taking higher levels of cache into account, and Section 5.7 concludes the chapter.

5.2 CSMS EFFICIENCY DEFINITION

First, we need to define how to deal with the concept of efficiency in general. As a platform, CoaaS connects two types of entities: (i) context consumers, and (ii) context providers. Similar to the real world, consumers want the best possible service delivered free of charge and without any latency. On the other hand, providers have constraints in their physical abilities (latency, throughput) and, potentially, want to receive payments for providing the sensing services. In the middle of this IoT ecosystem, CoaaS (as well as any other IT solution) is operated by a private or public company. Consequently, the platform operation cost must be balanced with the received income, so that the overall service cost would be reasonable, predictable and clear.

We have identified the primary sources of potential income and loss. Further, we have to define service level agreements (SLAs) which are established between the context consumer and the context provider. Then, income is easily defined as a price paid by the consumer for having a query serviced correctly and in a specified amount of time. The loss is defined as the sum of several costs: (i) the price of remote services being called, (ii) the amount of cloud services (processing, storage, network) being used by the platform, (iii) the amount of penalties which are returned to the account of the consumer in the case when a query is not serviced in a specified time. Moreover, the administration of the platform can add any other constraints (e.g. maximum percentage of failed queries for a particular consumer). Eventually, according to the discussion above, we have defined efficiency as the optimal point of the platform's operation in terms of operational costs under certain circumstances (defined SLAs, estimated provider behaviour).

Defining efficiency in such a way allows us to proceed to the next step in the modelling process. The base level of context is a simple data item (also referred to as context attribute). A data item is an atomic value, for example, a sensor reading. The most essential characteristics of the data item which is used to serve a query are (i) the latency, required to access the data item, and (ii) the correctness of this data item. In general, according to the NOR-NOD concept, it does not matter to a consumer where the data item comes from to serve a query – is it retrieved

from a remote source or the cache. Consequently, all the storage of transient (non-static) IoT data can be viewed as a cache.

The "correctness" of a data item should also be considered. The assumption made here is that any data item which is retrieved from a data source can already be, strictly speaking, incorrect by the time it is delivered to a requesting site, as even with direct data transfer latencies of several milliseconds are unavoidable. Technically, the reading on the context producer side could change by that time. If the data item is cached for a specified period by the middleware platform, the probability that this data item holds an incorrect value increases.

We should also take into account the role of a CMP (facilitating horizontal integrations), and the difference to the role of real-time critical systems, which are designed for time and mission-critical applications. In CMP, a certain amount of inaccuracy and latency can be allowed if it is balanced by reasonable quality and cost. Consequently, we have adopted the notion of "freshness" [173],[169], which reflects how far a data item could deviate from the real value. The freshness can be estimated by monitoring the behaviour of a data item for a period of time. There is a vast body of knowledge applicable to predicting the behaviour of a value, based on time series analysis. However, these methods are uniquely defined for different types of data (i.e., binary, continuous) and will not be considered in this research. We assume that the expiration time is assigned to a data item so that that freshness will decrease linearly from 100% to 0% (or from 1 to 0), and it is always known to the platform in which state the data item is. In other words, it is always known in advance how many percent of freshness the data item will have at any point in time (refer to Chapter 2 for details). Hence, to define an SLA between a consumer and the platform, we just need to negotiate four main points: (i) the level of acceptable freshness, (ii) the price of serving a request, (iii) the acceptable delay, and (iv) the penalty in case the query is not served in a specified amount of time.

As the CoaaS platform is operating in the cloud environment, where the amount of resources is technically unlimited, we can potentially cache all the related data and refresh every data item at a very high rate. However, running a system with such a strategy will require an infinite amount of resources to afford paying for the infrastructure and the calls to remote services. Caching can undoubtedly help to reduce the number of expensive calls to remote providers as well as to reduce the query serving time. The disadvantage of caching is the loss of freshness and the increased cost of storage and processing resources, as well as the complexity added to the system. Finding an optimal balance between resource consumption and query execution time becomes an essential part of the project. Mathematical models,

which are used for decision making, are presented in Section 5.5. In the next section, we define the logical levels of cache.

5.3 CSMS LEVELS OF CACHE

In this section, we discuss the structure and levels of cache in CoaaS. We show how the cache structure and management techniques are related to the aforementioned NOR-NOD principle of platforms' operation, SLAs between CoaaS and context consumers, and CDQL language.

We start with discussing logical levels of cache, as this is more valuable, we will discuss physical layers in Section 5.4.

We distinguish four primary levels of cache, as shown in Figure 5.1. Level 1 (L1) cache is the lowest level, which is responsible for storing data items that describe the state of a particular entity. Higher levels of cache contain results of executing aggregation and situation functions (L2), CDQL request (single-entity query) results (L3), and full CDQL query results (L4). Each decision about caching or eviction must be based on an estimation of current and future validity of a particular piece of information. The details of these cache levels are described below. We use the bottom-up approach to introduce the cache levels from the most basic to the higher-levels, as it matches the nature of forming the cache.

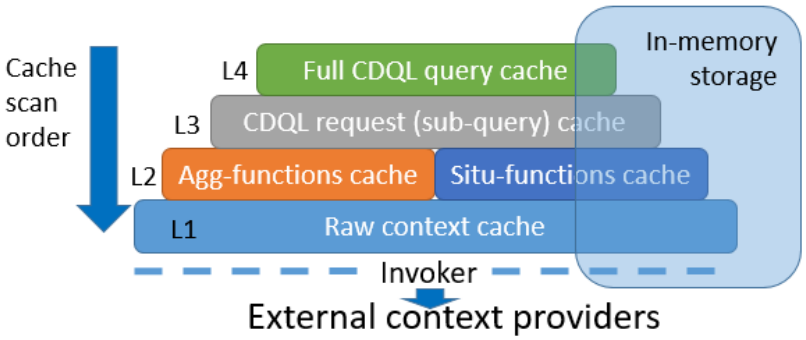


Figure 5.1 - Levels of cache in CSMS

We also use a CDQL query from the smart mobility use case for demonstrating how these levels are aligned to a CDQL representation. The query that aims to find parking facilities is presented in Figure 5.2.

```
1  prefix mv:http://schema.mobivoc.org , prefix schema:http://schema.org
2  pull (targetCarpark.*)
3  define
4  entity destinationLocation is from schema:place
5  where destinationLocation.address =
6  "Monash University Clayton Campus, 40 Exhibition Walk, Clayton VIC 3800",
7  entity targetCarpark is from mv:ParkingGarage where
8  (
9      (
10         distance(targetCarpark.geo,destinationLocation.latitude,destinationLocation.longitude, "WALKING") < 2000
11         and goodForWalking(destinationLocation) > 0.7
12     )
13     or distance(targetCarpark.geo,destinationLocation.latitude,destinationLocation.longitude, "WALKING") < 500
14 )
15 and distance(targetCarpark.geo.latitude,targetCarpark.geo.longitude,
16 destinationLocation.latitude,destinationLocation.longitude, "WALKING") < 2000
17 and targetCarpark.capacity.currentValue > 0
18 and costOfParking(targetCarpark,"10:10", "12:00") < 20
19 and availabilityParking(targetCarpark, "10:10","12:00") = true
```

Figure 5.2 - CDQL query used for pushing information about suitable carparks when a car is close to a destination.

5.3.1 LEVEL 1 (L1) – RAW CONTEXT (CONTEXT REPOSITORY)

We need to emphasise what we mean by caching in this particular study once again. When any transient data (context) is retrieved from the producer and saved in the CoaaS context repository, it is already considered cached. This data is no longer contained "in" the producer system and, potentially, can become invalid or obsolete at any moment. Consequently, the CoaaS context repository, which contains raw context retrieved from the producer, is considered an L1 cache.

It is not a database containing the freshest possible data. The producer system has the 'perfect' copy of the data. CoaaS has a copy that is 'fresh enough' for a certain SLA. If a cache miss happens, or the whole cache is empty, the fresh copy will be requested from the producer. For instance, consider the following expression (Line 17):

targetCarpark.capacity.currentValue > 0

The repository record *"capacity.currentValue = 14"* is an example of raw context, which reflects the number of available spots for resident parking of the area which is served by Carpark A. Raw context storage and caching is organised at an entity instance -based level and follows the structure of a semantic vocabulary adopted for this particular entity type. In the current example, an entity instance is a particular parking garage A, while the entity type is a "*ParkingFacility*". The entity-related document is annotated according to MobiVoc semantic vocabulary [46], which is developed as part of bIoTope consortium.

To illustrate the content of L1 cache, consider a part of the CDQL query, which aims to find parking facilities with vacant spots: "*targetCarpark.capacity.currentValue > 0"*.

In order to process this expression, all the parking facilities in the search scope will be scanned to compare the number of vacant spots with zero. The number of available parking spots is transient, but still a cacheable parameter. These primitive low-level data items (contextual attributes) populate the L1 cache.

5.3.2 LEVEL 2 (L2) – FUNCTION EXECUTION RESULT CACHE

As described in [98], CDQL contains several main types of functions, namely aggregation functions, and situation functions. An aggregation function returns a value computed by performing a particular operation on one or more entities. Aggregation functions can be built-in or user-defined. Built-in functions can be generic (average, max, min) or domain-based (costOfParking). User-defined functions are registered and hosted by CoaaS as RESTful endpoints. An example of an aggregation function, which computes a value based on one entity, is "*costOfParking*" (Line 18). This function calculates the cost for a particular customer with a particular permit type for a specific slot of time and using other possible parameters. Another example is the "*availabilityParking"* function (Line 19), which computes the availability of parking in the garage based on the opening hours specification.

A situation function is represented by a Context Spaces Theory (CTS)-based description of a situation [63], [60]. For example, the "goodForWalking" function in Figure 5.2 (Line 11) computes the probability that the environment in a particular place is suitable for walking for a particular person.

The execution time of such functions can vary. In the case when the number of entities processed by the function is high, the execution time, as well as the amount of consumed resources, can also be very substantial. In the case when an external API needs to be called for doing the computation (e.g., Google weather API for the goodForWalking function), the cost of function execution can be high. To tackle this issue, we organised the second level (L2) of caching, which stores the results of executing aggregation functions. Each function is linked to statistical data stored in the CoaaS Performance Repository (PR). The data includes the frequency of using the function, used parameters, execution time, access restrictions, and other relevant metadata. Accordingly, results acquired after the execution together with the estimations of freshness are also stored in PR.

5.3.3 LEVEL 3 (L3) – CDQL REQUEST CACHE FUNCTION

As defined in [98], the CDQL query consists of one or more entity-based requests, which are joined and can also be thought of as sub-queries. These requests can be executed in sequential order or parallel, according to the CDQL query execution plan, which is constructed by the query engine. A request can include operations (equality, comparison, etc.) over raw context as well as operations over the results of functions. These operations can be connected by logical operators (and, or, not). A request is only a part of the full CDQL query. Correspondingly, the probability of a repeated request arrival is higher than a probability of a repeated full CDQL query arrival. For example, in Figure 5.2, two entities are defined to serve a carpark search query. The "targetLocation" entity is defined by address. Geographical coordinates, which are retrieved from a geocoder API to serve this particular query, can be cached and later reused for serving a completely different query in which the location will be defined by this address. From the performance perspective, reusing a CDQL request result is the second best possible cache hit after reusing the whole CDQL query result in level 4.

5.3.4 LEVEL 4 (L4) – FULL CDQL QUERY CACHE

The entire CDQL query result can also be cached and reused. The L4 cache hit is the best situation from the resource consumption perspective. However, it is least likely to happen, as all factors such as the whole CDQL query, access restrictions, and freshness requirements must match. Nonetheless, in a situation when a consumer reissues the query soon after the same query was issued or the same query comes from another user with a similar profile, the cached result can significantly improve the performance. Another possibility to improve the hit rate of L4 is the use of approximation techniques. For instance, a geo point can be replaced (or duplicated) by an area (geo box), making a particular cached query much broader, consequently increasing the hit rate.

5.3.5 CACHING PYRAMID AND CACHE SCAN ORDER

As shown in Figure 5.1, the four levels of cache form a pyramidal structure, where the first layer (L1) represents the bottom of the pyramid and the fourth level (L4) represents the top. The cache is scanned from the top to the bottom of the pyramid during the execution phase of the query. In this section, we show the possible options and steps that are passed during this process.

Option 1: L4 cache hit. When a query comes into the platform and is validated by the parser, it is passed to CSMS for checking the possibility of reusing the whole query. If the query and consumer profile match and the freshness/validity of results are considered to be satisfactory, the query results are retrieved from the storage and passed back to the CoaaS Query Engine (QE), which, in turn, passes the results to the consumer. In general, this is the fastest and least resource-consuming option.

Option 2: L3 hit (partial hit). If the whole query had no matches in the L4 cache, a corresponding signal is returned by CSMS to the QE. QE constructs an execution plan, which consists of entity-based requests execution and the order of this execution. Usually, requests are executed in a particular order, as the forming of full request often requires results of the previous ones for making the join. However, parallel request execution is also possible in certain cases, for example, to find an entity located between two geographical points, which are defined by addresses. As locations are not dependent on each other, they can be retrieved in parallel. When it becomes possible to execute a particular request, QE sends the request to CSMS. If the request can be fulfilled from the L3 cache, CSMS immediately returns the result. If not, CSMS processes the request as usual, which involves both: using L2 and L1 caches as well as calling the invoker to retrieve missing data from the remote producers (Option 2 and Option 3).

Option 3: Level 2 hit (partial hit). While a request is processed by CSMS, it needs to compute the values of aggregation and situation functions, if they are used in the request. If functions are not used, this step is omitted, and CSMS uses Option 4. If functions are used in a query, CSMS checks the cache for stored results of function execution, which might be reused. For example, if the cache contains a result of the "goodForWalking" function for a particular person in a specific area and the result is considered fresh enough, it is reused.

Option 4: Level 1 hit (partial hit). When the raw context value is needed to process the request, CSMS checks the raw context repository (L1 cache). If the value is missing or considered not fresh enough, CSMS calls the invoker to fetch the new value, uses it for processing the request and potentially caches the value. The decision on caching and refreshing in L1 is the main focus of this particular study.

In the next section, we define the concept of caching levels from the physical perspective.

5.4 CSMS PHYSICAL LEVELS OF CACHE

In the previous section, we have described four logical levels of cache that are directly aligned to the concepts of CoaaS and CDQL. However, there exists another dimension for cache allocation, which is the data and computation placement. In general, the cache can be stored on disk and in main memory. Moreover, on-disk storage can be divided into Hard Disk Drive (HDD) and Solid State Drive (SSD). Computations, in turn, can be run on processing nodes with different characteristics (fast, medium, slow instances). Consequently, the question of allocation of data and computational tasks also has its place in the choice of an optimal caching strategy. Both in-memory cache, as well as fast computation nodes, cost more to run. Finding an optimal cache placement strategy will help to meet the CoaaS constraints and will reduce the cost of running the process. The concept of an in-memory cache is shown graphically in Figure 5.1 as a blue bubble on the right, which intersects all the logical levels.

Initially, we considered the physical caching layers as our main priority. However, while running experimental queries within the bIoTope project, it became apparent that reducing data access time within CoaaS resulted in less productivity increase and cost reduction when compared to logical cache based on the NOD-NOR concept. Moreover, the technological changes, such as enhancements in SDD technology, can also decrease the benefit of physical level separation. We also found that even higher increase in cost efficiency could be achieved by optimising the usage of external context providers. Consequently, as our main priority, we have focused on the development of the cost model, as well as the CoaaS prototype for the logical levels.

The basement of the cache pyramid, which is the level of raw context attributes (L1), is the most essential part of the model. We define and discuss strategies and models for L1 in the next section.

5.5 A MODEL FOR A SINGLE CONTEXT ATTRIBUTE

When we started the investigation of potentially applicable caching strategies, we found that the nature of an easily scalable cloud system made classical approaches to caching (LRU, LFU, etc.) not suitable (refer to Chapter 2), as with the cloud concept there is no boundary to which we can fill the cache. It is infinite. Moreover, there is no clear way to attach these approaches to the cost of operation. For that reason, we have chosen the utility-function based approach, where a function is applied to every cached item in order to estimate the need to keep the item in a cache.

In particular, we have extended the Al-Turjman's Least Value First (LVF) policy [173], which was proposed for the Information-Centric Networking (ICN) caching nodes. It is a function-based caching approach, which takes into account the delay of data fetching, popularity and age parameters for making a decision about cache eviction. The proposed utility function assigns a value to each object. Al-Turjman in general defined a Delay Model, a Popularity Model, and an Age model. Details of the LVF approach can be found in Chapter 2. However, LVF strategy was developed in an early stage of work on IoT caching based on ICN architectures and is more theoretical, rather than applicable as a real methodology. Moreover, the LVF approach is also more suitable for fixed-size systems. Extending the LVF model brought us to defining the primary influential factors, from which we derived the Need-To-Refresh (NTR) formula. The NTR index, which is computed according to Eq. 5.1, has a simple physical meaning: it shows how strong the need to refresh a particular data item is; the higher the index, the better for the system to retrieve a data item and to drop the value of NTR index back to its possible minimum.

A formula for computing the NTR is presented below:

$$NTR = \alpha \times Latency + \beta \times Popularity + \gamma \times Unreliability + \\ + \delta \times FreshnessLoss + \varepsilon \times RetrievalCost + \\ + \epsilon \times ProcessingCost + \theta \times \frac{Penalty}{Price} \qquad \textbf{(Eq. 5.1)}$$

The parameters in the formula above are described and discussed in Section 5.5.1. Later in this chapter, we also show why we decided to switch from NTR approach to another methodology, which we call the refresh rate-based approach. However, the main criteria used for a rate-based strategy remained the same as those we defined for NTR. Despite the inapplicability of NTR to the management of non-fixed size systems, it is still useful for an initial explanation of the theory. The reasons are provided later in this chapter.

5.5.1 PARAMETERS INFLUENCING THE CACHE DECISION

The main criteria initially used for the NTR strategy consists of the following six items: (i) Freshness, (ii) Latency, (iii) Popularity, (iv) Retrieval cost, (v) Unreliability (Possible unavailability of a context provider), (vi) Processing cost.

All the listed parameters are directly related to individual data items. Once we have initially listed the parameters, which are influencing the "Need to Refresh" decision, we can understand how these parameters affect the value of NTR.

Graphically, these influences are presented in Figure 5.3. The up arrows indicate the parameters, which are directly proportional to the value of NTR. The down arrows indicate the parameters that are inversely proportional to the value of NTR. In particular, when the freshness, the retrieval cost and the processing cost are decreasing, the NTR is growing. On the contrary, the NTR is growing when the latency, popularity, unreliability and the penalty to price ratio are increasing.

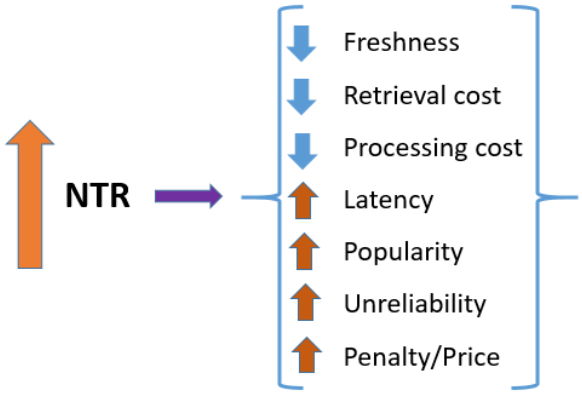


Figure 5.3 – Factors, which influence the need-to-refresh index of a data item

Below we provide the discussion of main parameters and their influence on cache decisions. We discuss both the influence on NTR-based approach, as well as on refresh rate-based (RRB) approach, which is chosen as our primary approach.

Freshness

The freshness of a data item is, probably, the most important, yet hard to compute parameter in the NTR formula.

Freshness can take a value between 0 and 1. It can be estimated as a result of monitoring historical values of a particular data item. Freshness close to 1 shows that a data item is considered to be very reliable (just fetched from a provider). Freshness loss in Eq. 5.1 is the inverse of freshness. The principal methodologies which are applicable to estimating the freshness of a data item are mentioned in Chapter 2. The lower the freshness, the higher the NTR. Obviously, when the freshness of a data item is low, the chance of getting a cache miss is higher. A cache miss will result in losses. Consequently, it is better to refresh a data item, as it has high NTR.

The notion of the data item freshness of a data item is the same in both NTR and refresh rate-based approaches.

Retrieval cost

Retrieval cost is the amount of money that is charged by the data provider for using its API. For instance, Google Weather API provides information about the weather for free only for a relatively small number of calls per day. For a production-scale system, the number of calls will be significantly higher, and each API call will have a cost.

In a simple case, when the producer is charging some amount for *n* number of API calls, the Retrieval cost (*RetrievalCost*) of one data item *di* is defined below:

$$RC_{di} = \frac{Cost_n}{n} \quad \textbf{(Eq. 5.2)}$$

The higher the retrieval cost, the less we want to retrieve a data item. Consequently, high retrieval cost is lowering the overall NTR, as well as the rate of retrievals for the rate-based approach.

Processing cost

By processing cost, we mean the cost of networking, storage, and computational resources that are consumed by a cloud system to perform all the needed computations over a single data item when it is retrieved.

Identifying the cost of processing a single data item is not a simple task, as the number of data items processed by one server node can be, in certain cases, measured in terms of billions per hour. At the same time, if the system retrieves data at a very high rate (up to every millisecond), even a minimal cost of processing a single data item will result in a substantial total. Every data item can be registered in zero to many situation monitoring subscriptions. As each subscription has individual parameters, each incurring a different amount of load during processing, estimating the resulting cost of processing an attribute becomes even harder. However, we can apply some assumptions, which will help in determining the cost. For instance, the total cost of running the part of the cluster that is responsible for serving the event processing pipeline is designated as $Cost_{processing\_hour}$. Accordingly, the processing cost (PC) of one data item *di* is depicted below:

$$PC_{di} = \frac{Cost_{processing\_hour}}{items\_processed} \quad \textbf{(Eq. 5.3)}$$

In the expression above the $items\_processed$ represents the total number of items processed by the event processing pipeline.

A higher cost of processing of data item corresponds to a lower NTR index, as it is more beneficial to avoid frequent retrievals of the data item to save on the cost of operation.

In practice, it means that there is no such thing as free data retrieval and caching. Even though the context provider is ready to provide data without any charges and does not limit the number of requests, the processing cost, especially in the case of frequent retrieval will result in higher overall cost. In the model, we consider the overall cost as a combination of the processing cost and the retrieval cost.

Latency

Any communication with a remote data source introduces unavoidable latency. However, the value of such latencies can significantly vary depending on the distance to the provider, the quality of the network and its load, the number of redirects and routers, the number of requests that a provider is serving at a particular time and many other technical issues. In general, high latency significantly reduces the possibility of getting data from a provider on the fly while serving an incoming query. It means that a particular data item, which is retrieved through a slow connection, is to be kept and maintained in the cache. We designate this latency parameter as $Latency_{di}$ for a data item *d.i.* The latency can be can be calculated as:

$$Latency_{di} = \frac{\sum_0^n Latency_n}{n} \qquad \textbf{(Eq. 5.4)}$$

In the expression above, n is the number of successful tries to fetch the data from a remote provider.

For NTR estimation, we are using a normalised latency, which is defined below:

$$Latency_{di\_norm} = \frac{Latency_{di}}{avg(Latency_{total})} \qquad \textbf{(Eq. 5.5)}$$

In the expression above, latency of access to a data item ($Latency_{di}$) is the time, which is needed by CoaaS to obtain a data item from a particular context provider, and the total latency ($Latency_{total}$) is a sum of average latencies of all data items.

The higher the latency, the higher the NTR, as the high latency increases the chance of being not able to serve a request in an ad-hoc fashion.

For a refresh rate-based approach, we use an average latency of access to the data item during the monitored period, which is statistically similar to the planning period.

Popularity

From the business perspective, CoaaS aims to keep customers satisfied by providing a reasonably high quality of service for a reasonably low cost. The first question to answer is how popularity affects the overall NTR value of a data item. If an item is popular (and is predicted to be popular), it means that the item will be used to serve many context queries and, in turn, definitely requires caching. Otherwise, the number of external calls will be very high causing latencies, network overloading, and high cost of retrieval and processing.

For the NTR strategy, we used the relative popularity of a data item (*d.i.*), which is designated as $RelPopularity_{di}$:

$$RelPopularity_{di} = \frac{RequestNumber_{di}}{ReqNumber_{total}} \quad \textbf{(Eq. 5.6)}$$

In the expression above, $RequestNumber_{di}$ represents the number of arriving requests to a particular data item during a monitored period, and $ReqNumber_{total}$ represents the number of all requests arriving at CoaaS during the same period.

For the refresh rate-based approach, instead of relative popularity, we use the arrival rate (number of arrivals) of requests to a particular data item, which is designated as λ. The arrival rate is computed over a period of time t between t1 and t2, and it follows a Poisson distribution:

$$Popularity_{di}(\mathrm{t} \geq \mathrm{t1}, \mathrm{t} < \mathrm{t2}) = \lambda \quad \textbf{(Eq. 5.7)}$$

Unreliability

As we are dealing with IoT entities serving as context providers, we should assume that some of these entities (e.g., mobile) can be connected to the network via unstable and slow (wireless) channels. All these factors lead to a high possibility of a context provider becoming unavailable during certain time period due to network bottlenecks or handover issues. The unavailability of a data provider, in the case of having no valid data in the cache, will cause CoaaS to either exclude the data from a particular provider from the result set, or to increase the overall time of serving the query, while trying to establish a connection. Both consequences are not acceptable especially if high QoS is needed. Therefore, we suggest that low reliability (high probability of node unavailability) will be a reason for the higher caching rate. In the simplest case, unreliability of a data item can be calculated as:

$$Unreliability = \frac{Fetch_{di_{fail}}}{Fetch_{di_{total}}} \quad \textbf{(Eq. 5.8)}$$

In the above formula, $Fetch_{di_{fail}}$ represents the number of unsuccessful attempts to fetch data from the provider, and $Fetch_{di_{total}}$ represents the total number of tries to get data during a planning period.

If the predicted possible period of unavailability is added to the amount of time since the data item is fetched and this total period is longer than the estimated expiry period, it is better to pre-fetch data before the moment of expiration. If the system remains confident the data is available on request until the query arrives, and it will not be able to retrieve new data because of its unavailability, the quality of service will inevitably suffer. To the best of our knowledge, including provider reliability into the caching decision, is a novel approach and was not considered in previous works.

The higher the provider unreliability, the higher is the NTR, as the chance of not being able to retrieve a data item from an external provider on the fly will increase because of its unreliability.

Penalty/Price ratio

The price of a data item access is the price which a context consumer pays to retrieve a data item. Penalty, on the other hand, is the amount of money which is returned to the consumer account when a data item request is not served in a specified period of time. The price of access to a data item and the penalty are defined in the SLA. The higher is the penalty (while the price is fixed), the higher is the penalty/price ratio, and, consequently, the higher is the need to refresh a data item, as not being able to properly serve a query will cause a significant losses.

5.5.2 DISCUSSION OF THE NTR-BASED APPROACH

After defining the main components of NTR formula (Eq. 5.1), we need to find the weight coefficients $\alpha, \beta, \gamma, \delta, \varepsilon, \epsilon, \theta$ for the NTR formula. In the case of a fixed size system that would be sufficient. For a non-fixed size system, we also need the threshold of NTR, above which the data item would be a candidate for refreshing.

However, after trying to proceed in this way we found that it is not appropriate. This is because after any given time duration the value of NTR may change, resulting in the need to re-compute the NTR and also to sort the data items in order to find which ones are above the NTR threshold. As the number of data items could be huge, the process can become very computationally tedious and expensive. Moreover, we could not find a proper methodology for estimating the threshold and weights, as the freshness is not connected to the levels defined by

SLAs, and the fact that the influence of different SLA levels on hits and misses is also not introduced, although, the list of main influential factors remain valid. Moreover, the NTR index, while it remains unused for the model, is useful to exemplify the whole concept of balancing the factors, which are influencing the caching decisions.

At this point, it becomes evident that a more reasonable approach would be to estimate the optimal rate of retrieval of a data item. By retrieval rate, we mean how much time the CSMS should wait between the moment of last retrieval of a data item and the moment of planned retrieval. We call this approach a rate-based approach. In the next section, we define the possible strategies and provide details on their usage.

5.6 ADDING LOGICAL AND PHYSICAL LEVELS OF CACHE TO THE MODEL

As described in previous sections, the cache pyramid in Figure 5.1 consists of two dimensions: the logical and the physical dimensions. The logical dimension matches the CDQL constructs, and the physical dimension matches available technological instances.

We start our discussion at the logical level. Due to the complexity of realisation of proactive strategy on all levels of cache, we have applied the reactive strategy at levels L2 – L4. This means, that even when a proactive decision is made to refresh a context attribute, all the cached results of requests, functions and full CDQL queries need not be recomputed. Any higher-level objects contain one or several context attributes from L1, where some operators were applied to these attributes. Consequently, the reusability of items from L2-L4 levels, also depends on the freshness properties of the basic L1 realisation. These higher-level objects are stored and can be reused until the expiry of any context attribute, which has the shortest lifetime for the least expensive SLA.

When a CDQL query arrives at CoaaS, CSMS tries to reuse the cached data starting from the top of the cache pyramid, which is the level of full queries (L4). A hit at L4 is the best case for CSMS, as there will be no need to spend processing resources on searching through the lower levels of cache and fetch data from external sources. Moreover, the processing resources which are needed for the Query Engine to validate the query, parse it and perform final joining of results are also saved.

In a case when a result of a query is not present in the cache, or the result is based on attributes with freshness below the SLA threshold, the Storage Query Execution Manager (SQEM) moves to the lower level – L3, and tries to obtain results for parts of the CDQL query one by one. If there are one or more misses, SQEM moves down the pyramid, until it hits the

lowest level L1. If there is a miss again, data is retrieved from the context provider. After the requests are processed and the query is served, corresponding objects are cached at L2-L4 levels. The two main benefits of storing not only the full CDQL query result, but also the partial query results, are (i) for cases when the same query arrives and parts of the query result are outdated, other partial results can be reused and only the outdated results have to be fetched from external sources, and (ii) for cases when different queries are involved, but parts of queries are intersecting, these partial results can be reused.

As with any caching, the described approach has several disadvantages. The first problem is that storage, retrieval, and frequent refresh of cached data will result in rising the cost of using cloud services. The second problem is that sequential scanning of all the levels from L4 to L1 also increases the latency of query serving time. The worst-case latency of full CDQL query servicing can be found as follows:

$$Latency_{CDQLquery} = Latency_{L4} + \sum_0^{n3} Latency_{L3} + \sum_0^{n2} Latency_{L2} + \sum_0^{n1} Latency_{L1} + \sum_0^{n\_ext} Latency_{extAttr_n} \quad \textbf{(Eq. 5.9)}$$

By worst-case latency, we mean the fully sequential search, which resulted in all levels being searched and no relevant cached data was found; eventually, the data is retrieved from the external context providers. The fully sequential search can be defined as sequential search both for the levels (L3 search starts after L4 has failed), and inside the levels (request *R2* is searched after the request *R1* was found or detected as missing).

In the Eq. 5.9, the indexes *n1* to *n4* represent the number of documents that need to be searched for a corresponding level L1 to L4. For instance, n3 represents the number of requests that constitute the query, and the corresponding number of documents with results that need to be searched in the storage; n2 represents the number of functions used in the query and the corresponding number of documents that need to be searched. Next, *n1* represents the number of entity instances, which need to be retrieved from the storage if the query should be assembled. Eventually, the latencies of the access to attributes retrieved from external sources on the fly, (*extAttr)* are taken into account.

Technically, it is possible to scan the cache in parallel, so that if the search in L4 fails, searching through L3, L2 and L1 are already performed or partly performed. However, this approach will cause a rise in processing cost, as the number of parallel queries will increase; moreover, in a case when the number of hits in L4 is high, all the queries to L3-L1 will be wasted.

At the same time, an increase in the latency, which is created by potential unsuccessful scanning of L4-L2 will cause a rise in the number and amount of penalties.

As a CDQL query arrives at the Query Engine and, then to CSMS, it is unknown if a particular request is cached or not. Consequently, we can avoid scanning only by switching off caching for a particular level and informing SQEM about it. Technically, it is possible to implement switching off caching at a smaller scale than at a whole level (e.g. entity type). However, that will increase the complexity of SQEM implementation and the model even more.

Now we can add the discussion of physical levels into the picture. As described above, the cache can be horizontally scaled among many server nodes (scaling out). Each server node has the following characteristics, which are associated with cost: (i) performance of the node (CPU frequency, number of cores, main memory), (ii) type of storage (in-memory, SSD, HDD). Characteristics of a server node have a direct influence on an important feature – the time of cache scanning. However, there exists another component, which is influencing the time of scanning – the amount of data that is handled by a single server node. Modern NoSQL databases use the *'sharding'* approach for horizontal partitioning of data for spreading the load. A *shard* is a partition of data that is kept on separate server node. In other words, the less cached data is stored at a server node (shard size), the faster the node finishes the search and the more parallel queries it can process. On the other hand, the more shards we create, the higher is the cost of supporting the storage and processing infrastructure.

The problem is now scoped to finding the balance between the potential income against penalties and the cost of computational and storage resources of a particular type. The resources are defined as the number of nodes of each type. Then, we can make a decision about using or not using each logical level of cache and which resources must be associated with each level. This decision can be made by constructing a corresponding linear programming model.

5.7 CONCLUSION

In this chapter, we focused on caching strategies and models. We have formulated and described three main cornerstones, which influence cache management decisions in CoaaS: the NOR-NOD concept, the non-fixed size of the caching and processing system, and the transient nature of IoT data. We have also introduced the levels of cache, both from the logical point of view and from the physical point of view. The logical view matches the CDQL constructs, such as raw context attributes, requests, functions, and full queries. The physical dimension

represents the technological possibilities and limits, as well as the costs caused by using these technologies. The choice here is between the available storage options (in-memory, HDD, SSD), as well as between the number and performance of the processing nodes.

We have shown how the cache efficiency can be represented as a function of components such as freshness of a cached data item, latency of access to a context provider, probability of unavailability of a provider, popularity of a data item, cost of data retrieval, cost of processing, penalty in case of a cache miss, and the price of serving a request. We have formulated a theoretical construct called the NTR index to represent our idea. Then, we have shown how the logical and physical levels of cache can be integrated into the optimisation model.

However, due to the impossibility of applying the value-based approach directly, we moved to a refresh rate -based approach, where the cache decision is expressed as the amount of time until the next retrieval after the end of the freshness period for the most expensive SLA.

In the next chapter, we analyse the refresh-based approach and corresponding strategies in detail. Then, we propose the methodology for cost prediction of a planning period for the refresh rate -based approach, taking the possibility of multiple SLAs into account.

Chapter 6: CSMS refresh rate -based caching strategies and models

6.1 INTRODUCTION

In the previous chapter, we have discussed how the management of the physical and logical levels of cache can bring the performance and efficiency benefits to the Context as a Service (CoaaS) platform.

We also discussed the factors, which are influencing the caching decisions at a level of a single context attribute. For that, we introduced a concept of the Need-to-Refresh (NTR) index, which is a function-based caching policy. However, while being useful as a theoretical construct, the NTR –based policy is not practical, due to three main reasons: (i) the high amount of required computations to keep the index regularly updated, (ii) the problem with finding the threshold, under which the proactive refreshing of an attribute is not beneficial, and (iii) the problem of connecting the generic NTR index to multiple Service Level Agreements (SLAs), which can be defined at the same time.

To address the problem, we have investigated the possibility of using the Refresh Rate-Based (RRB) policy for the cache management decisions in the CoaaS platform. The aim of the RRB policy is to find an optimal period for refreshing each context attribute. Then, a component responsible for retrieving the context from the external sources (CoaaS Invoker) can use the computed period for scheduling the retrievals.

This chapter is organised in the following way: In Section 6.2 we describe the refresh rate based approach for finding the optimal behaviour of the Context Storage Management System (CSMS), for efficient context caching and retrieval. In Section 6.3.1, we present the mathematical model, which aims to estimate the operation cost for the case of a single SLA. We extend this model for a case of multiple SLAs in Section 6.3.2, and Section 6.4 concludes the chapter.

6.2 REFRESH-RATE BASED CACHING STRATEGIES

In this section, we describe three main refresh-based caching strategies, which are possible for a simple context attribute. These strategies are the full coverage strategy, the reactive strategy, and the proactive strategy. The strategies are corresponding to the database mode, redirector mode, and NOD-NOR (Not only Database, Not only Redirector) modes of

CMP operation, (refer to Chapter 2 and Chapter 5). Each of the strategies can be beneficial in certain circumstances. Consequently, there is a need for detailed investigation of each strategy and development of methods for making optimal decisions.

6.2.1 FULL COVERAGE STRATEGY

The first strategy is *full coverage*, which is graphically represented in Figure 6.1. We use the following designations throughout the diagrams: a vertical orange arrow represents a retrieval of fresh data by CoaaS from a context provider; a vertical dark blue arrow represents a request to a data item arriving from context consumers to CoaaS at random moments of time. We assume that the arrivals of requests are happening in accordance with the Poisson distribution. Horizontal dotted lines represent the levels of freshness of a data item, which is stored in CoaaS. The line which is shown at the level of '*Freshness = 1*' represents the level of maximum freshness when the data item is just retrieved from a provider and cached in CSMS. The diagonal green lines, which are going diagonally downwards from the top of orange arrows, represent the loss of freshness of a data item over time. The second horizontal dotted line below the level of '*Freshness = 0.7'* represents the SLA threshold of freshness. A data item with a freshness level below this threshold cannot be reused when a request arrives from a context consumer. The height of the dark blue arrows (requests) depicts the desired level of freshness. The time between the level of maximum freshness until the moment of reaching the SLA threshold is called the *expiry period* (or *freshness period*), which is designated as *Exp. Period* in the diagrams and as *T* in subsequent mathematical expressions. In the situation when there are several SLAs being defined, expiry periods differ for each SLA.

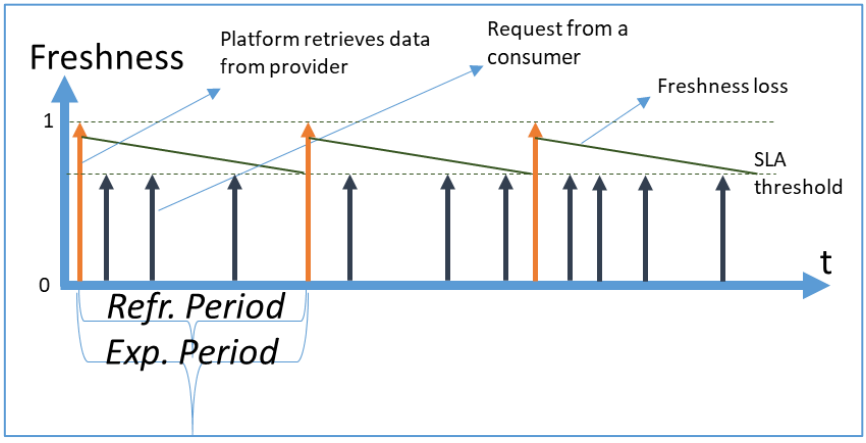


Figure 6.1 - Full coverage caching strategy

The time between retrievals is called a *refresh period* and designated as *Refr. Period*. We distinguish *planned refresh period* from the *real refresh period.* A *planned refresh period* is the time between planned retrievals of a data item by CoaaS from a context provider. A *real refresh period* is the average time between retrievals. The real refresh period can be smaller than a planned period, as possible cache misses can trigger immediate refreshment of a data item.

Another important notion is the *planning period*, which is designated as $PlanningPeriod$ in subsequent mathematical expressions. The *planning period* is the period of time for which the decision about the caching strategy and the corresponding parameters is made. The planning period contains many refresh periods. The most important feature of a planning period is that the distribution of arriving queries is expected to be Poisson distribution. In our discussion and evaluation, we use a planning period equal to one minute. However, it can be different depending on the patterns of request arrivals.

With a full coverage strategy, CoaaS is always able to serve a request to a particular context attribute out of the cache. In some sense, it can be viewed as a database mode of operation, where freshness is taken into account. The difference with the database mode is that while in real IoT database all the data contains the latest sensor readings, in CSMS's cache the attribute value contains a sensor reading that is fresh enough to cover the most expensive SLA. At the moment when the freshness of a context attribute becomes too low to serve a request for any SLA, a new attribute value is retrieved from the provider. Consequently, the *maximum reasonable refresh rate* ($MaxRate$) is the rate at which CSMS refreshes a data item as soon as its expiry period for the most expensive SLA is over:

$$MaxRate = PlanningPeriod/ExpPeriod \qquad \textbf{(Eq. 6.1)}$$

Refreshing at a higher rate will not improve cost-efficiency. On the contrary, it will reduce the efficiency, as the cost of retrieval will increase. With full coverage strategy, the penalty component will equal zero, as it is always possible to serve a request out of the cache (Refer to Chapter 5, Section 5.2, where the elements of cost efficiency were described). The estimation of the cost of operation during a planning period is trivial:

$$\begin{aligned} &TotalCost(FullCoverage) \\ &\quad = \lambda \times PlanningPeriod \times PriceReq \\ &\quad - (PlanningPeriod/ExpPeriod) \times PriceRetrieval \end{aligned} \qquad \textbf{(Eq. 6.2)}$$

In the expression above, λ is arrival rate of request in Poisson distribution t, which represents the popularity parameter (Section 5.5.1). The arrival rate and the length of the planning period should be measured in the same unit; for instance, in our simulations, we use the number of requests per second to characterise λ and the planning period is in seconds. $PriceReq$ is the price that a consumer pays for each request, and $PriceRetrieval$ is the price which CoaaS pays to retrieve a data item from a context provider. From here onwards, we assume that $PriceRetrieval$ includes both the price of the external provider API usage and the cost of internal cloud resources used to process the data item. Eventually, the total cost of operation for the full coverage strategy (Eq. 6.2) consists of one positive component and one negative component. The positive part is the income received by CoaaS while serving a certain number of expected requests for a specific price. The negative component is the number of retrievals multiplied by the cost of retrievals.

6.2.2 REACTIVE STRATEGY

The second strategy we need to discuss is the *reactive strategy,* which is graphically represented in Figure 6.2. The reactive strategy is the opposite of the full coverage strategy. The refresh rate is set to zero, meaning that CSMS does not retrieve data proactively. A retrieval can happen on the fly only in the case of a request arrival, which causes a cache miss. We state that the moment of planned retrieval is set to infinity.

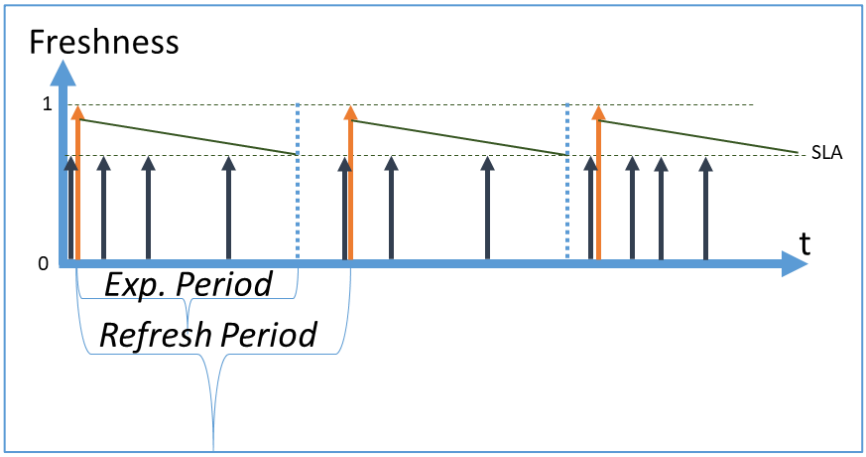


Figure 6.2 - Reactive caching strategy

In Figure 6.2, a request from a consumer to CoaaS (dark blue arrow) happens before the retrieval from an external provider (orange arrow). During the expiry period, all the arriving requests are served out of the cache. When the freshness (green diagonal line) reaches the SLA threshold, (this moment is shown with a dotted vertical blue line), the cache is not covering the

data item anymore. When a new request arrives after the moment of expiry, (blue vertical dotted line), it causes immediate retrieval, (the orange arrow occurs immediately after the request). Then, the process starts again. Every request which causes a retrieval is a *cache miss*, which incurs a penalty, (Refer to Section 5.2). At the same time, every request which arrives during the time when the cached data is fresh, is a *cache hit*. The percentage of hits is called the *hit rate*, and the percentage of misses is called the *miss rate*. We designate the hit rate as *HR* and the miss rate as *MR*. As can be seen from the figure, the expiry period and the refresh period are not equal, different from the full coverage strategy. The refresh period depends entirely on the arrival of the first cache miss, making it random. As such, we use the average refresh period over a planning period.

Eventually, we can estimate the cost of the planning period for the reactive strategy as follows:

$$\begin{aligned} &TotalCost(Reactive) \\ &\quad = \lambda \times PlanningPeriod \times PriceReq - \lambda \\ &\quad \times PlanningPeriod \times MR(T) \times PriceRetrieval - \lambda \\ &\quad \times PlanningPeriod \times MR(T) \times Penalty \end{aligned} \qquad \textbf{(Eq. 6.3)}$$

The expression above (Eq. 6.3) differs from the expression for the full coverage strategy (Eq. 6.2) as the second component represents the price paid by CoaaS for retrievals, by using the miss rate (MR). The third component represents the loss caused by penalties, which is also expressed using the miss rate.

Hit and miss rates for a reactive strategy depend only on the arrival rate of requests (λ) and the expiry period of a data item (T). The hit can be computed as described by Jung et al. [179]:

$$HR(T) = \frac{E[N(T)]}{E[N(T)] + 1} \qquad \textbf{(Eq. 6.4)}$$

In the expression above, E[N(T)] is the expectation of requests which arrive during the freshness period: $E[N(T)] = \lambda \times T$. Accordingly, based on the fact that MR(T) = 1 – HR(T), we can also compute the miss rate as follows [179]:

$$MR(T) = \frac{1}{E[N(T)] + 1} \qquad \textbf{(Eq. 6.5)}$$

The hit rate (HR) and miss rate (MR) are always less or equal to one. The total number of retrievals during the planning period ($RetrievalNum$) depends on the miss rate and the expected number of requests ($RequestNumber$) and can be computed as:

$$RetrievalNum = MR(T) \times RequestNumber \quad \textbf{(Eq. 6.6)}$$

6.2.3 PROACTIVE STRATEGY

The third main strategy is the *proactive strategy*. With this strategy, the moment of planned retrieval happens after the expiry period of the most expensive SLA, but before t = ∞. We call the time Δt between the end of the expiry period and the moment of *planned retrieval* a "gap". If a request arrives during this period, it "falls within the gap," and CSMS has a cache miss. The proactive strategy is the most sophisticated approach, as it requires adaptive mechanisms to estimate the optimal size of the gap in order to achieve the most efficient cost of operation. We designate the size or *the time of the gap* as *t_gap*. A diagram illustrating the proactive strategy and the concept of the gap is presented in Figure 6.3.

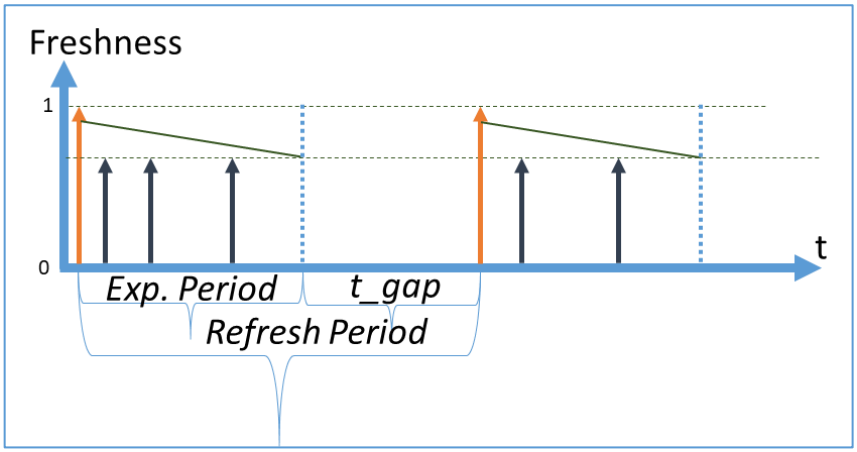


Figure 6.3 - Proactive caching strategy

As can be seen in the figure, CoaaS proactively retrieves a data item and requests are served out of the cache till the moment of expiry. The time elapsed from the moment of expiry (blue dotted vertical line) to the moment of the next planned retrieval (orange arrow) is the gap, and the size of this gap is shown as *t_gap*.

Figure 6.3 depicts the perfect scenario where there are no requests arriving during the gap. In this case, the *refresh period* is the same as the *planned refresh period*, which is equal to a sum of the expiry period and the gap size.

However, if there is a request arrival during the gap, a cache miss will occur, and there are a number of options to handle this issue, which needs to be discussed. These options include: (i) immediate retrieval to serve the request and no reuse of the retrieved data item (no reuse), (ii) immediate retrieval and reuse until the next planned retrieval (reuse without shift), (iii) retrieve and shift the time of the next planned retrieval (reuse with shift). Technically, it is possible to define more options, such as no retrieval and waiting until the moment of the next planned retrieval.

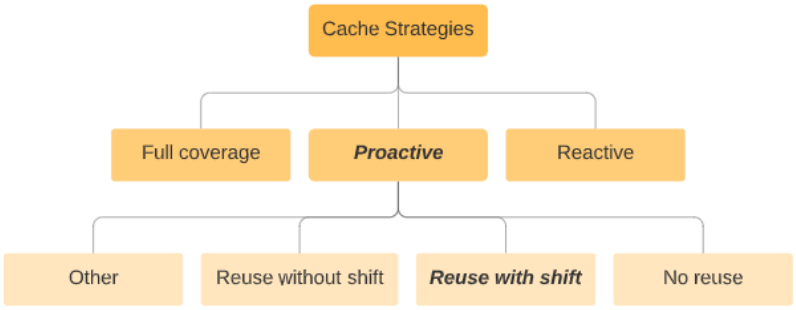


Figure 6.4 - Classification of approaches

The classification of possible options to deal with retrievals during the gap is presented in Figure 6.4. We have designated the strategies, which were in the focus of the current study with the bold font. The diagrams of these options are presented in Figure 6.5.

The "no reuse" option is depicted in Figure 6.5 (a). Requests, which arrive during the gap (shown with yellow arrows), cause retrievals. However, the retrieved items are not cached (not reused), and each new request causes yet another retrieval. At the end of the gap, a proactive retrieval is then initiated by CoaaS.

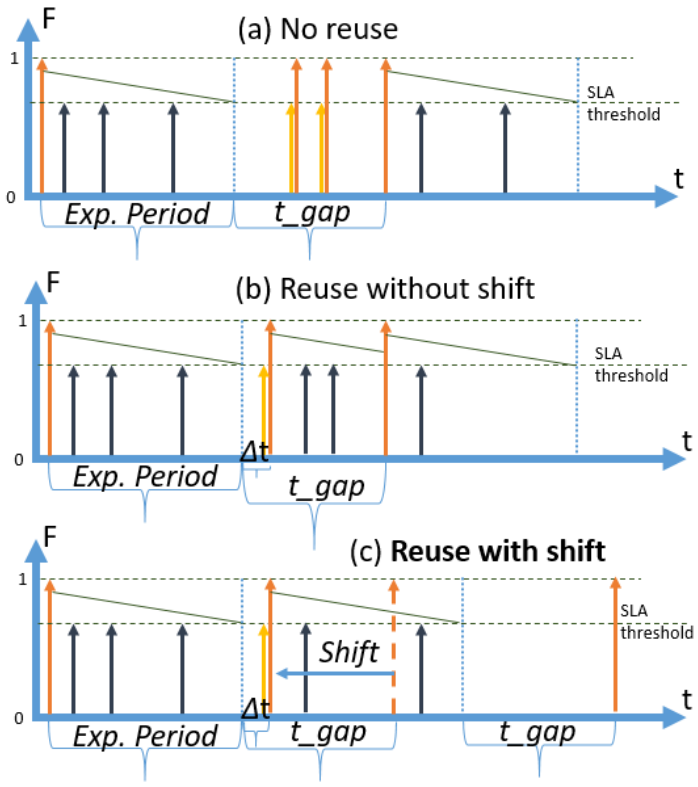


Figure 6.5 - Possible approaches to dealing with cache misses

The "reuse without shift" option is shown in Figure 6.5 (b). In this case, if a request arrives during the gap, it causes a retrieval, and the data item is cached. When the planned gap expires, CoaaS proactively retrieves a data item, despite the fact that the cached item is still fresh enough to serve queries.

The "reuse with shift" option is shown in Figure 6.5 (c). The difference between this and the previous option is that after the cache miss and data retrieval, the planned retrieval time is changed (shifted). The initial planned time is shown as a dotted orange arrow. As the planned retrieval is shifted, the initial planned retrieval is cancelled, and the gap time starts from the end of the expiry period.

While Options (a) and (b) have some benefits from the perspectives of technical simplicity resulting in more straightforward planning, the ultimate cost-efficiency can be achieved with the Option (c), as only this option allows reuse during the entire freshness period. Consequently, we have chosen Option (c) for our studies of proactive cache management. In subsequent discussions, the term "proactive strategy" will mean a "proactive strategy option reuse with shift".

To elaborate further on Option (c), the concept of *shifted retrievals* happening during a planning period is graphically represented in Figure 6.6. In Figure 6.6 (a), the planning period is depicted, where no query arrivals happen during the gaps. The end of the planning period is represented by a vertical purple line. As everything goes according to a plan, there are three complete refresh periods fitting in one planning period. In the second diagram (Figure 6.6 b), a more realistic situation is shown. Requests arrive during the gap, causing retrievals to happen before the planned moment. We call this a *shift*. Eventually, in the current example, four periods are fitting in one planning period, as the real refresh periods are shorter than the planned refresh periods.

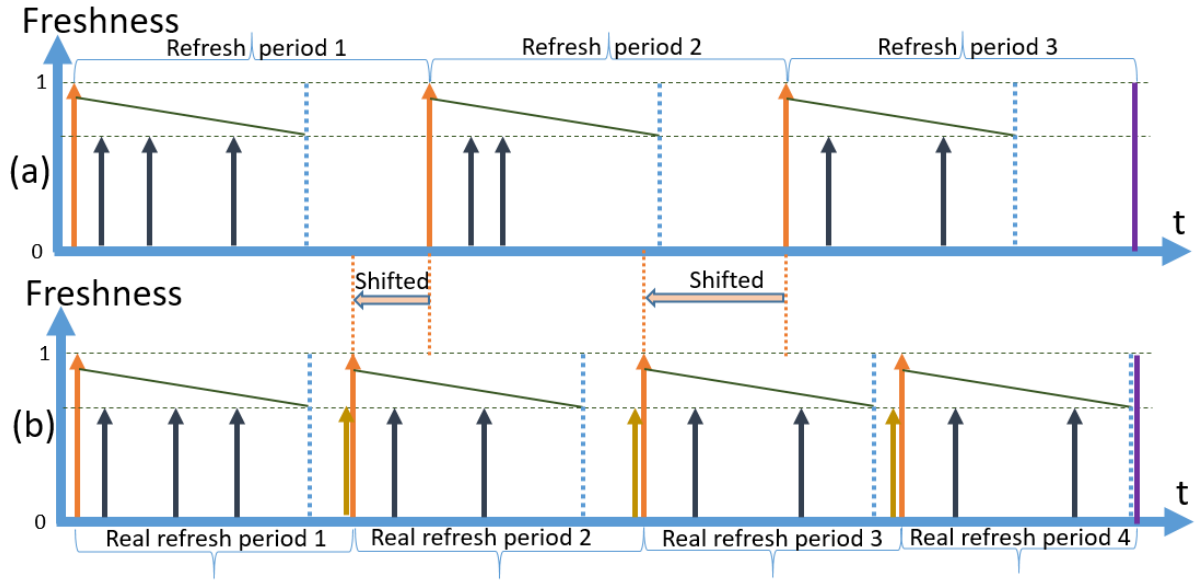


Figure 6.6 - Planned and real retrievals in the long run

6.2.4 STRATEGY CONSIDERATIONS WITH RESPECT TO MULTIPLE SLAS

For a situation with a single SLA, the optimal strategy is either full coverage or reactive. This means that using the proactive strategy and planning the gap is always suboptimal. However, there are two fundamental issues.

The first point is that we still need to decide between choosing the reactive or the full coverage strategy. For that, we need to estimate the cost of operation in both cases. While it is clear for the full coverage (Eq. 6.2), it is not as evident for the reactive strategy. The main issue with the reactive strategy is that at the moment when there is a cache miss (and a triggered retrieval), the process starts at this point again. Eventually, it is not obvious how to compute

the cost of operation for the planning period, as there will be hits, misses, and retrievals involved. We have demonstrated our solution in Eq. 6.3.

The second point is that when there are more than one SLAs defined, there is always a gap unless coverage of the most expensive SLA is chosen as a strategy. The concept of gaps appearing in the case of multiple SLAs is graphically presented in Figure 6.7. In this example, we have decided to provide full coverage only for SLA3 (designated with bold dotted line), which has the longest expiry period, compared to SLA1 and SLA2. In the third refresh cycle, there are no cache misses, and the full gaps for SLA1 and SLA2 are marked. We have provided this example to show how the gaps will inevitably appear in the case of a policy with multiple SLAs.

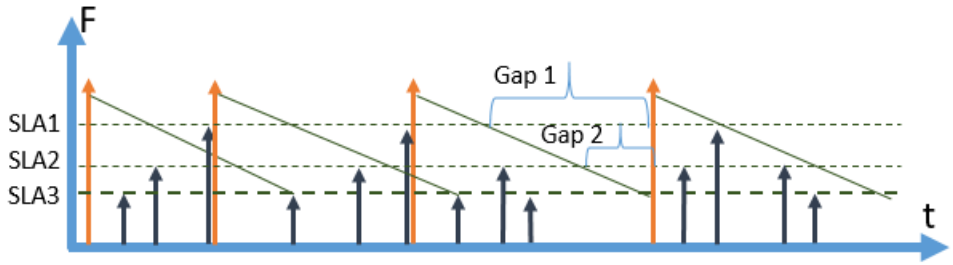


Figure 6.7 - Gaps in case of several SLAs

In a policy with multiple SLAs, the shift will happen in the same way as a policy with single SLA. When there is a cache miss, and a data item is retrieved on the fly, the whole process shifts to the left on the time axis, making the real number of refresh periods higher than the number of planned refresh periods, as shown in Figure 6.6 (b). Consequently, there is a need to estimate the cost of a planning period taking into account the number of hits, misses, and refreshes.

In this section, we have described three main strategies of caching in CSMS. For the proactive strategy, we have also described the concept of gaps as well as our approach to handling cache misses when they happen during the gap. We have illustrated our concepts on a policy with one SLA, as it is the most basic level. As we have shown, the decision process for a single SLA policy is not excessively complex in practice. However, there is still a need to develop a method to estimate cost-efficiency in a situation with gaps. Moreover, in the situation where multiple SLAs are defined, making a decision about the optimal caching rate becomes much harder, so a method for cost prediction becomes even more important.

In the next section, we will discuss and present our approach to predicting the planning period cost for policies with single SLA and multiple SLAs.

6.3 COST PREDICTION OF PLANNING PERIOD

In this section, we are addressing one of the most important questions – the cost prediction of a planning period. At first, we formally define the main notions and convey a generalised formula for cost estimation, when multiple SLAs are defined.

Let $SLA = \{SLA1, SLA2, SLA3 \ldots SLAn\}$ be the set of defined SLAs; each member of the set contains cost-related components, defined in Section 5.2. These components are (i) the request price, (ii) the retrieval cost (which consists of the service call cost, and the processing cost), and (iii) the penalty cost.

There are two more parameters defined for each SLA: the freshness period and the expected number of requests. The freshness period *T* defines for how long since the retrieval a data item can be reused. The expected number of requests represents the number of requests for a particular SLA, which we expect to arrive during a planning period based on the statistical data from the performance datastore.

Let $RequestPrice_s$ be the request price received by CoaaS for the all SLAn requests for each $s \in SLA$. For every $s \in SLA$, where *T* is the length of the freshness period, and t_g is a variable, representing the chosen gap size.

$$RequestPrice_s = HR(T, t_g) \times RequestNumber_s \times RequestPrice_s \quad \textbf{(Eq. 6.7)}$$

Let $PenaltyCost_s$ be the penalty cost incurred by all SLA misses for each $s \in SLA$

For every $s \in SLA$:

$$PenaltyCost = MR(T, t_g) \times RequestNumber_s \times PenaltyCost_s \quad \textbf{(Eq. 6.8)}$$

Let *CTR* (Cost of Triggered Retrievals) be the cost incurred by total number of retrievals caused by cache misses. CTR is defined for all the SLAs together, as any

Let *CAR* (Cost of Automatic Retrievals) be the cost incurred by the total number of retrievals occurred according to the planned time

Eventually, we can formulate a generalised formula for predicting the cost of a planning period:

$$CostOfPlannedPeriod = \\ = \sum_{s \in SLA} RequestPrice_s - \\ - \sum_{s \in SLA} PenaltyCost_s - CTR - CAR \quad \textbf{(Eq. 6.9)}$$

The main challenge in the application of the formula above is the dependence of the cost of penalties $PenaltyCost$ as well as the cost of triggered retrievals (CTR) on the miss rate. The number of automatic retrievals also depends on the number of cache misses and shifts which are caused by this misses.

In the next section, we research the problem of finding the needed components for the application of the generalised formula in real scenarios. We start with the most simple scenario – a single SLA policy, and then proceed to a more complex scenario, where two SLAs are defined.

6.3.1 A POLICY WITH ONE SLA

A detailed diagram of a planned refresh period for a policy with one SLA (1SLA) with a cache miss and a shift is presented in Figure 6.8. The process starts at time t = 0 with a retrieval, which is depicted by an orange arrow. The freshness of cached attribute equals 1 at this stage. The next retrieval is planned at time t2. The end of the freshness period happens at time t1, and the time between t1 and t2 is the size of a gap. When a request arrival (depicted by a black arrow) happens between t1 and t2 (black arrow), an unplanned retrieval is triggered. The time between the moment of the unplanned retrieval and the time of planned retrieval (depicted by the horizontal green arrow) is the shift. Note that now there is no real need for a refresh t2, as the value will still be fresh enough by that time.

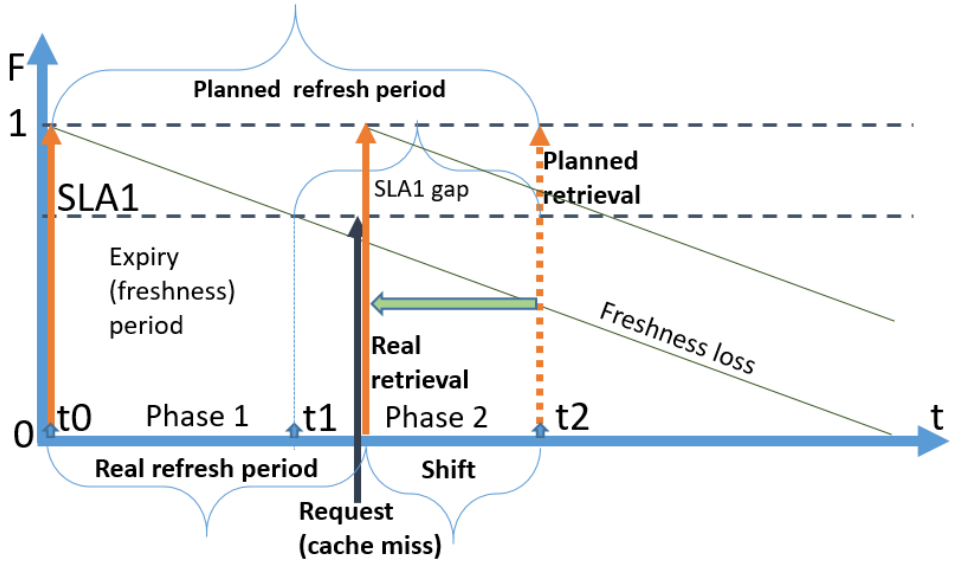


Figure 6.8 - Cache miss caused a shift of the retrieval moment

In Figure 6.9, the detailed diagram of one full refresh period (top graph), and the corresponding cumulative distribution of miss probability during one refresh period (bottom graph) is presented. During the first phase (Phase 1), when the freshness of the attribute value is above the SLA threshold, only cache hits are possible. After t1, the Phase 2 starts. In Phase 2, the probability of meeting the first request arrival, which will cause a cache miss, grows exponentially according to the feature of the Poisson process:

$$P_{miss}(t_{miss} \geq t1) = 1 - e^{-\lambda t}, \qquad \textbf{(Eq. 6.10)}$$

In the expression above, *t* is the time after *t1*.

When the second retrieval happens due to the moment of planned retrieval or a miss, the process will start from the beginning.

We should note that P_{miss} represents the *cumulative* probability. It represents the chance of encountering the arrival before the time *t*, but not exactly at time *t*. The possible range of P_{miss} is depicted in Figure 6.9 as $P_{miss}(SLA1)$. In the case, when a planned retrieval is set too far (reactive strategy), the cumulative probability of a cache miss will reach 1.

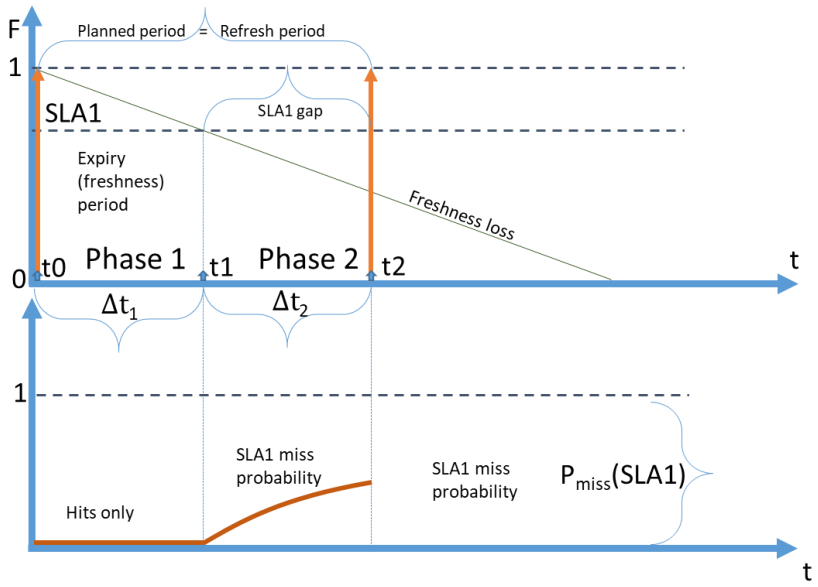


Figure 6.9 - Phases of one refresh period and the cumulative probability distribution of a miss

Our main interest is to estimate the cost of operation for a planning period. The general expression for estimating the cost is presented below:

$$CostOfPlanningPeriod = \sum_0^n RequestPrice - \sum_0^k PenaltiesCost - \sum_0^l CostOfTriggeredRetrievals - \sum_0^j CostOfAutoRetrievals \quad \textbf{(Eq. 6.11)}$$

In the expression above, *n* is the number of requests, *k* is the number of misses, *l* is the number of retrievals that are caused by a cache miss and *j* is the number of retrievals that happen according to the planned time. Here we separate the cost of triggered retrievals from the cost of auto retrievals for two main reasons: firstly, clarity of explanation and, secondly, the potential use for more complex scenarios, where automatic retrievals are done in advance and therefore can be served more cheaply, as they may potentially use another interface of retrieval or different processing facilities. In summary, we can say that the total cost of operation has one positive and two negative components. The positive component is the sum of prices, which are paid by consumers for requests to attributes. The negative components are (i) the sum of penalties, caused by cache misses, and (ii) the sum of retrieval costs.

The cost of serving requests in the case of full coverage (no gap) or case of the reactive strategy (infinite gap) was already described in Section 5.6.1 and Section 5.6.2 correspondingly. However, in the case when there is a finite gap (*proactive strategy*), the computation becomes not as obvious. The main issue is that every time a miss happens, the whole picture shifts to the left along the time axis. Eventually, it is not clear how many refresh

periods will fit in a planned period. Consequently, we can conclude that if we can find the hit rate, the miss rate, and the real number of refreshes, we would be able to estimate the cost.

Next, we can transform a formula for the cost of the planning period (Eq. 6.11) to a more usable form. The concept is based on finding the **H**it rate, the **M**iss rate, and the ratio of **R**efreshes to requests. Consequently, we called the formula of our approach the **HMR formula**. Details of the formula and its components are described below.

$$\begin{aligned} &CostOfPlanningPeriod = \\ &= HR(T, t_g) \times RequestNumber \times RequestPrice + \\ &+ MR(T, t_g) \times RequestNumber \times RequestPrice - \\ &- MR(T, t_g) \times RequestNumber \times Penalty - \\ &- RR(T, t_g) \times RequestNumber \times RetrievalPrice \end{aligned} \quad \textbf{(Eq. 6.12)}$$

The *hit rate (HR), miss rate (MR) and refresh ratio (RR)* are dependent on the length of the freshness period (*T*), and the size of a chosen gap, also referred to as gap time (t_g).

As $HR(t_g) + MR(t_g) = 1$, we can simplify the expression above by replacing the first two components with ($RequestNumber \times RequestPrice$), consequently:

$$\begin{aligned} &CostOfPlanningPeriod = \\ &= RequestNumber \times RequestPrice - \\ &- \mathrm{MR}(T, t_g) \times RequestNumber \times Penalty - \\ &- RR(T, t_g) \times RequestNumber \times RetrievalPrice \end{aligned} \quad \textbf{(Eq. 6.13)}$$

The positive component of the expression in Eq. 6.13 represents the income, which CoaaS receives from serving a certain number of requests ($RequestNumber$) for a specific price ($RequestPrice$), which arrive during the planning period. The two negative components are the penalties and the price of data retrieval. The penalties can be expressed as the number of all queries, multiplied by the miss rate and multiplied by the cost of each penalty. It means that for finding the penalty component, we need to find the miss rate.

While the first two components in Eq. 6.13 are intuitively able to be defined, the retrievals component is harder to define. The problem is that retrievals can happen because of cache misses, (triggered retrievals), as well as the result of successfully reaching the end of the gap, (planned retrievals). For the cost estimation, it does not matter what type of retrieval happens. We only need the number of all retrievals that are expected to occur during the planning period. We have found that a convenient way to find the number of all retrievals is to express them

through a ratio coefficient, which represents the ratio of retrievals to incoming requests. We call this coefficient the *Refresh ratio* and designate it as RR. It is vital to note that, while hit and miss rate can only take values between 0 and 1, the refresh ratio can take any positive value. Eventually, the third component of the Eq. 6.13 can be calculated as the refresh ratio multiplied by the number of requests and the price of one data retrieval.

We have described the meaning and the general components of the HRM formula; now, we can proceed to find the HR, MR, and RR, which are needed to apply the HRM formula.

To find the hit rate (HR(T, t_g)) for a policy with gaps we can use the expressions for HR(T) and MR(T), which we used for the reactive strategy (Eq. 6.4, Eq.6.5) together with the feature of the Poisson process, which describes the probability of meeting the first arrival after a random point in time ($P = 1 - e^{-\lambda t}$). Eventually, the hit rate and miss rate can be calculated as follows:

$$HR(T, t_g) = \frac{E[N(T)]}{E[N(T)] + 1 - e^{-\lambda\, t_g}} \quad \textbf{(Eq. 6.14)}$$

$$MR(T, t_g) = \frac{1 - e^{-\lambda\, t_g}}{E[N(T)] + 1 - e^{-\lambda\, t_g}} \quad \textbf{(Eq. 6.15)}$$

In the expression above, $N(T)$ represents the number of requests, which will arrive during the freshness period of a cached data item.

An expected number of requests per real refresh period is defined as:

$$E[ReqPerRealRefreshPeriod] = E[N(T)] + 1 - e^{-\lambda\, t_g}, \quad \textbf{(Eq. 6.16)}$$

In the formula above, by real refresh period, we mean the average time between retrievals, either triggered or automatic.

In this case, the refresh ratio (RR) is defined as:

$$RR(T, t_g) = \frac{1}{E[N(T)] + 1 - e^{-\lambda\, t_g}} \quad \textbf{(Eq. 6.17)}$$

Multiplying RR(T, t_g) by the expected number of requests during the planning period gives us the number of retrievals which happen during this period:

$$RetrievalNum = RR(T, t_g) \times RequestNum \quad \textbf{(Eq. 6.18)}$$

Now we have obtained all the required parts for the HMR formula, and it is possible to compute the overall cost of serving requests to a data item for a planned period for any given gap size t_g.

We have run a set of simulations to check the components and the whole HMR formula. The results of this simulation show that the results obtained analytically closely match the results of an experiment. The results of the simulations and corresponding analytical solutions are available in the evaluation chapter (Chapter 7).

In the next section, we are progressing from a simple 1SLA policy to the more advanced and realistic scenarios with multiple SLAs.

6.3.2 A POLICY WITH MULTIPLE SLA (2SLA)

In this section, we analyse a scenario with two SLAs as an example for situations with multiple SLAs. In Figure 6.10, a detailed diagram of one refresh period (top graph) and the corresponding cumulative probability of a miss (bottom graph) for a scenario with two SLAs are presented.

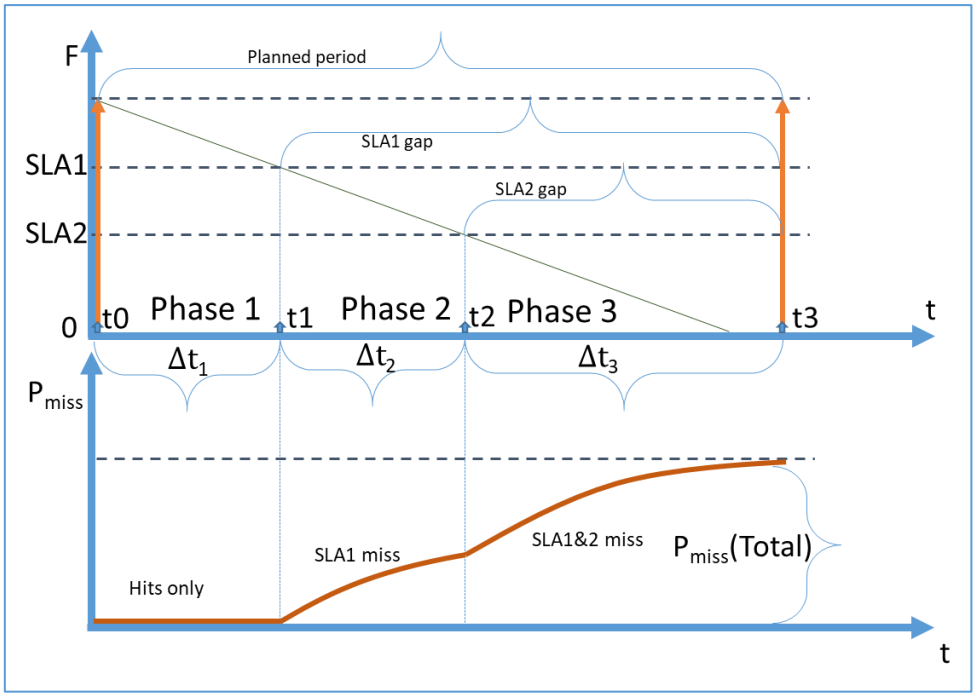


Figure 6.10 - Phases of one refresh period with 2SLA policy and probability of a miss

The main difference of the 2SLA policy (Figure 6.10), compared to the 1SLA policy (Figure 6.9), is that there is no single expiry period. SLA1 and SLA2 expiry periods are

overlapping; however, the SLA2 expiry period is longer. That leads to a situation when an SLA1 request arriving during the second part of the SLA2 freshness period causes a cache miss. In the diagram presented in Figure 6.10, we have identified 3 phases of the process. Phase 1 is the state when all the SLAs are covered by the cache, and a miss cannot happen. Phase 2 is the state when SLA1 is not covered (SLA1 gap), but the data is still fresh enough for SLA2. In phase 3, both SLAs are not covered (SLA1&2 gap) and any arrival of a request would cause a cache miss.

The bottom graph shown in Figure 6.10 represents how the cumulative probability of getting a cache miss is changing over time. During Phase 1, the probability equals zero, then, during Phase 2, it grows exponentially, as SLA1 requests can cause cache misses. In Phase 3, the cumulative probability is also growing exponentially, but faster, and both SLA1 and SLA2 requests can cause a cache miss.

We can rewrite the formula of the cost for the planned period in the following way:

$$\begin{aligned} CostOfPlannedPeriod = {} & \sum_0^n RequestPrice_{SLA1} + \\ & - \sum_0^m RequestPrice_{SLA2} - \sum_0^k PenaltiesCost_{SLA1} - \\ & - \sum_0^r PenaltiesCost_{SLA2} - \sum_0^l CostOfTriggeredRetrievals - \\ & - \sum_0^j CostOfAutoRetrievals \end{aligned} \qquad \textbf{(Eq. 6.19)}$$

In the expression above, *n* is the number of SLA1 requests, *m* is the number of SLA2 requests, *k* is the number of SLA1 misses, r is the number of SLA2 misses, *l* is the number of retrievals that are caused by a cache miss and *j* is the number of retrievals that occur according to the planned time.

Similarly to the process we described for the 1SLA policy, we can rewrite the previous expression to an expanded form using the **H**it rates, **M**iss rates, and the **R**efresh ratio, we call this the HMR-2SLA formula. The critical point is that we need to use separate hit and miss rates for each SLA, as penalties are different. However, the number of refreshes is common for all the SLAs, as any cache miss (SLA1 or SLA2) will cause a retrieval with an associated cost.

$$
\begin{aligned}
&CostOfPlanningPeriod = \\
&= HR_{SLA1}(T, t_g) \times RequestNumber_{SLA1} \times RequestPrice_{SLA1} + \\
&+ HR_{SLA2}(T, t_g) \times RequestNumber_{SLA2} \times RequestPrice_{SLA2} + \\
&+ MR_{SLA1}(T, t_g) \times RequestNumber_{SLA1} \times RequestPrice_{SLA1} + \\
&+ MR_{SLA2}(T, t_g) \times RequestNumber_{SLA2} \times RequestPrice_{SLA2} - \\
&- MR_{SLA1}(T, t_g) \times RequestNumber_{SLA1} \times Penalty_{SLA1} - \\
&- MR_{SLA2}(T, t_g) \times RequestNumber_{SLA2} \times Penalty_{SLA2} - \\
&- RefreshRatio(T, t_g) \times RequestNumber_{SLA1\&2} \times RetrievalPrice
\end{aligned}
\qquad \textbf{(Eq. 6.20)}
$$

We also can rewrite the above in a more compact way:

$$
\begin{aligned}
&CostOfPlanningPeriod = RequestNumber_{SLA1} \times RequestPrice_{SLA1} + \\
&+ RequestNumber_{SLA2} \times RequestPrice_{SLA2} - \\
&- MR_{SLA1}(T, t_g) \times RequestNumber_{SLA1} \times Penalty_{SLA1} - \\
&- MR_{SLA2}(T, t_g) \times RequestNumber_{SLA2} \times Penalty_{SLA2} - \\
&- RR(T, t_g) \times RequestNumberAll \times RetrievalPrice
\end{aligned}
\qquad \textbf{(Eq. 6.21)}
$$

The meanings of the components are similar to those we defined for the cost formula for 1SLA policy (Eq. 6.13). The difference for a policy with two SLAs is that the miss rates should be found for SLA1 and SLA2 separately.

To find hit rates, miss rates, and the refresh ratio, we need to describe the possible scenarios of cache misses. Misses can be caused by SLA1 and SLA2. SLA1 request arrival can cause a miss in phase 2 and phase 3. Illustrations of possible options where SLA1 request causes a miss in 2SLA scenario are presented in Figure 6.11 and Figure 6.12.

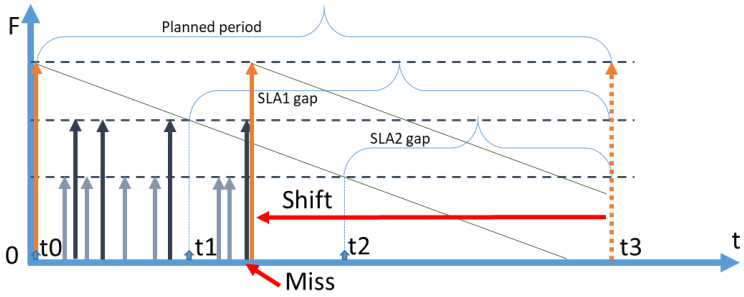


Figure 6.11 - SLA1 request causes a miss during the second phase

In Figure 6.11, a situation where an SLA1 request is arriving during Phase 2 and causing a cache miss is shown.

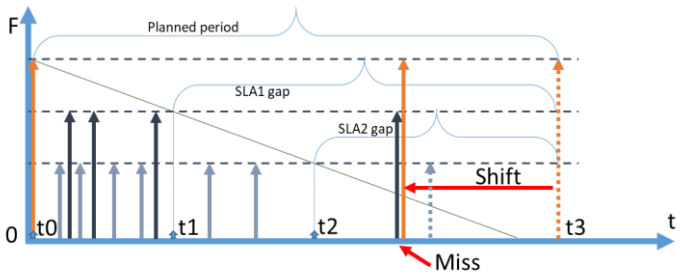


Figure 6.12 - SLA1 request causes a miss during the third phase

In Figure 6.13, an SLA1 request arrives during the third phase, which is also causing a cache miss.

An SLA2 request can cause a miss only during the third phase, as it is represented in Figure 6.13.

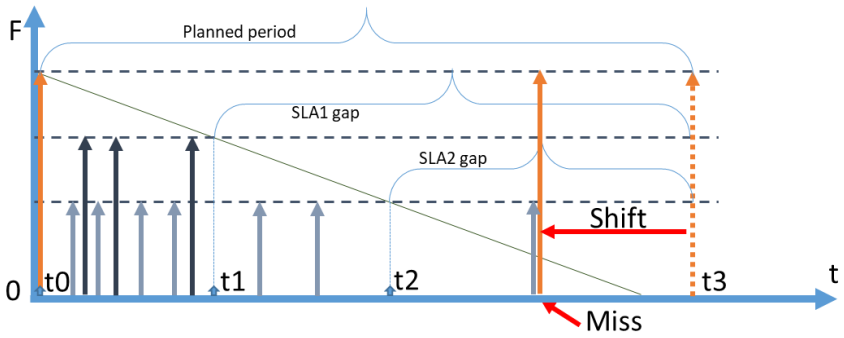


Figure 6.13 - SLA2 request causes a miss during the third phase

Now we have a situation in which the problem falls into three branches, which must be treated differently. As it is shown in Figure 6.10, each refresh period consists of three phases. In phase 1, all the incoming requests are served from the cache without any misses. In phase two, all SLA2 requests are served from the cache. However, any SLA1 request is causing a cache miss and retrieval with a corresponding shift. In phase 3, any incoming request causes a cache miss and retrieval.

Next, we describe what happens if we put the moment of an automatic retrieval inside each of the phases. Computations in phase 1 and phase 2 are trivial; however, phase 3 requires more effort to estimate the cost.

In phase 1, retrieving a data item before the end of phase 1 is not efficient, as the increase in the cost of retrieval is not compensated by an increase in income. Consequently, it makes sense to put the retrieval time only at the end of the expiry period of SLA1. It will provide full coverage for all SLAs. Consequently, it is easy to calculate the expected cost of a planned period as follows:

$$\begin{aligned} CostOfPlannedPeriod \\ &= RequestNumber_{SLA1} \times RequestPrice_{SLA1} \\ &+ RequestNumber_{SLA2} \times RequestPrice_{SLA2} \\ &- (PlanningPeriod/ExpPeriod_{SLA1}) \times RetrievalPrice \end{aligned} \quad \textbf{(Eq. 6.22)}$$

In phase 2 the moment of planned retrieval lays between the end of expiry period SLA1 and the end of expiry period SLA2, all the SLA2 requests are served from the cache, but any SLA1 request causes a miss and requires a retrieval. It means that we can look at this scenario in the same way as we did in a scenario for 1SLA. The only difference with the 1SLA scenario, is that the total cost will increase by the amount of average SLA2 queries served between the SLA1 expiry period and first SLA1 request.

In phase 3, the moment of planned retrieval is set after the end of both SLA1 and SLA2 expiry periods. It means that during phase 1 all the requests are served from the cache; during phase 2 only SLA2 requests are served from the cache, and during phase 3 any incoming request will cause a cache miss and a retrieval.

We provide a detailed analysis of the most complicated scenario when the retrieval is planned for phase 3.

As can be seen in Figure 6.10, starting from the beginning of phase 2 until the beginning of phase 3, the probability of a cache miss grows exponentially ($P = 1 - e^{-\lambda_1 t}$), where λ_1 is the arrival rate of SLA1 requests.

Since the beginning of phase 3, the probability of a miss also grows exponentially. However, the growth is steeper, as the probability is now influenced by both arriving processes.

According to a property of the Poisson process, the superposition of two Poisson processes is also a Poisson process. The intensity of the resulting process, according to [199] is: $\lambda = \lambda_1 + \lambda_2$. Another interesting property of two independent interfering Poisson processes is that a probability of meeting an arrival from the first process earlier than an arrival from the second process is [199]:

$$P\{X_1 < X_2\} = \frac{\lambda_1}{\lambda_1 + \lambda_2} \quad \textbf{(Eq. 6.23)}$$

We will be using this property in our discussion.

First, we can derive an expression for the probability of meeting a cache miss before the moment of automatic retrieval ($t = t_{ar}$). The expression will consist of a probability of SLA1 request arrival during the second phase (Δt_2) summed with a probability of any request arrival during Δt_3, which is multiplied by the probability of zero SLA1 requests arrivals during the Δt_2:

$$PmissPh3(t_{ar}) = \left(1 - e^{-\lambda_1 1 \Delta t_2}\right) + \left(1 - e^{-(\lambda_1+\lambda_2)t_{ar}}\right) \times (1 - \left(1 - e^{-\lambda_1 \Delta t_2}\right)) \quad \textbf{(Eq. 6.24)}$$

Separately graphs of the first and the second phase are presented in Figure 6.14 left and right graphs correspondingly.

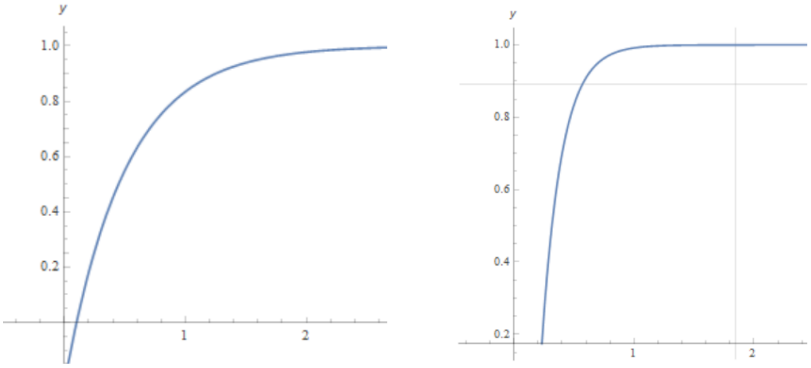


Figure 6.14 - Probability of a miss during Phase 2 (left graph), and Phase 3(right graph)

An example graph of the whole expression is presented in Figure 6.15.

This expression also allows calculation of an average amount of all misses before any (either triggered or automatic) refresh.

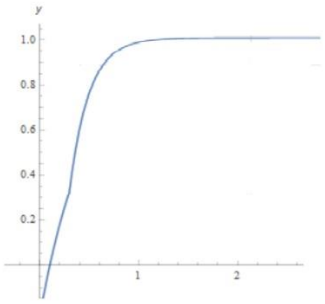


Figure 6.15 - Cumulative probability of a miss at Phase 3 for 2SLA policy PmissPh3(t_{ar})

After we have identified the cumulative distribution function of miss probability, we can move to finding the hit rate. First, we can derive expressions for total hit ($HitRate_{Total}$) and miss rates. By total hit rate, we mean the hit rate calculated for all the requests without distinguishing them by SLA. These parameters are not used in the cost formula directly, but they will be required for finding separate rates for both SLAs. A total hit rate can be calculated as the expected number of hits during one refresh period divided by the expected number of requests during one real refresh period:

$$HitRate_{Total}(T, t_g) = \frac{E[Hits_{Total}(T, t_g)]}{E[ReqNumber_{Total}(T, t_g)]} \quad \textbf{(Eq. 6.25)}$$

Next, we need to define the numerator and the denominator of the expression above.

The expected number of hits per request period consists of the number of hits during phase 1, plus the number of hits which can happen during phase2, (there can be no hits at phase 3). Finding the expectancy of total hits during phase 1 is trivial:

$$E[Hits_{Ph1}(T, t_g)] = (\lambda_1 + \lambda_2) \times \Delta t_1 \quad \textbf{(Eq. 6.26)}$$

However, finding the expectancy of total hits during the second phase requires more effort. It will also consist of two parts: (i) the number of hits which happened in a case when there were no SLA1 requests during phase 2, plus (ii) the number of hits which occurred in a case where there was an SLA1 request during phase 2.

The number of hits during phase 2 in case there were no SLA1 requests during this phase ($HitNumPh2|noSLA1req$) can be found as:

$$HitNumPh2|noSLA1req = \left(P(t_{miss} > t_2)\right) \times E[Hits_{SLA2}(\Delta t_2)] \quad \textbf{(Eq. 6.27)}$$

We also need to consider a scenario when an SLA1 miss happened during phase 2. The number of hits, which occurred before the SLA1 request happened at phase 2 ($HitNumPh2|withSLA1req$), can be found as:

$$HitNumPh2|withSLA1req = \int_{t=t_1}^{t_2} fmiss\big(SLA1(t > t_1)\big) \times E[Hits_{SLA2}(t - t_1)]dt, \quad \textbf{(Eq. 6.28)}$$

In the expression above, $fmiss\big(SLA1(\mathrm{t} > t_1)\big)$ can be found as follows:

$$fmiss\big(SLA1(\mathrm{t} > t_1)\big) = \lambda 1 e^{-\lambda_1(\mathrm{t}-\mathrm{t}_1)} \quad \textbf{(Eq. 6.29)}$$

The full expression for finding the expectation of hits during the refresh period is presented below:

$$\begin{aligned} E\big[Hits_{Total}(T, t_g)\big] &= E[Hits(\Delta t_1)] + \big(P(t_{miss} > t_2)\big) \times E[Hits_{SLA2}(\Delta t_2)] \\ &+ \int_{t=t_1}^{t_2} (fmiss(t - t_1) \times E[Hits_{SLA2}(t - t_1)])dt \\ &= (\lambda_1 + \lambda_2) \times \Delta t_1 + \Big(1 - \big(1 - e^{-\lambda_1 \Delta t_2}\big)\Big) \times (\lambda_2 \Delta t_2) \\ &+ \int_{t=t_1}^{t_2} \lambda_1 e^{-\lambda_1(t-t_1)}\, \lambda_2(t - t_1)dt \end{aligned} \quad \textbf{(Eq. 6.30)}$$

The integral at the end of the expression above can be solved as:

$$\int_{t=t_1}^{t_2} \lambda_1 e^{-\lambda_1(t-t_1)}\, \lambda_2(t - t_1)dt = \frac{\lambda_2 - \lambda_2 e^{\lambda_1(t_1-t_2)}(\lambda_1(t_2 - t_1) + 1)}{\lambda_1} \quad \textbf{(Eq. 6.31)}$$

Graphically, an example of this integral is represented in Figure 6.16.

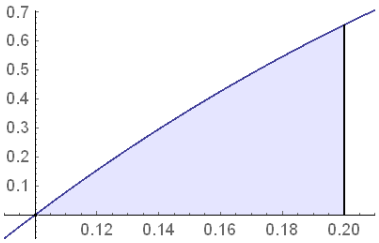


Figure 6.16 - Graphical representation of *HitNumPh2|withSLA1req*

Now we can move to find the denominator of the $HitRate_{Total}(T, t_g)$ expression, which is the number of expected requests during a refresh period ($E[ReqNumber_{Total}(T, t_g)]$).

We need to consider 3 parts: requests which happened during the first phase, requests that occurred during the second phase, and, possibly, requests that arrived during the second phase, if there were no SLA1 requests during phase 2.

$$E[ReqNumber_{Total}(T, t_g)] = E[Hits(\Delta t2)] + \left(P(t_{miss} > t_2)\right) \times E[Hits_{SLA2}(\Delta t_2)] + \int_{t=t_1}^{t_2} \left(fmiss(t - t_1) \times (1 + E[Hits_{SLA2}(t - t_1)])\right) dt + P(t_{miss} > t_2)) \times (E[Miss_{Total}(\Delta t_3)]) \quad \textbf{(Eq. 6.32)}$$

If we put previously defined parts into the expression, we receive a final expression for the expectation of requests during the refresh period:

$$E\left[ReqNumber_{Total}(T, t_g)\right] = (\lambda_1 + \lambda_2) \times \Delta t_1 + \left(1 - \left(1 - e^{-\lambda_1 \Delta t_2}\right)\right) \times (\lambda_2 \Delta t_2) + \int_{t=t_1}^{t_2} \lambda_1 e^{-\lambda_1 (t - t_1)} \lambda_2 (t - t_1) dt + (1 - (1 - e^{-\lambda_1 \Delta t_2})) \times (1 - e^{-(\lambda_1 + \lambda_2)\Delta t_3}) \quad \textbf{(Eq. 6.33)}$$

The integral can be solved in the following way:

$$\int_{t=t_1}^{t_2} \lambda_1 e^{-\lambda_1 (t - t_1)} \lambda_2 (t - t_1) dt = \frac{-e^{\lambda_1 (t_1 - t_2)} (\lambda_1 \lambda_2 (t_2 - t_1) + \lambda_1 + \lambda_2) + \lambda_1 + \lambda_2}{\lambda_1} \quad \textbf{(Eq. 6.34)}$$

Its graphical representation is shown in Figure 6.17.

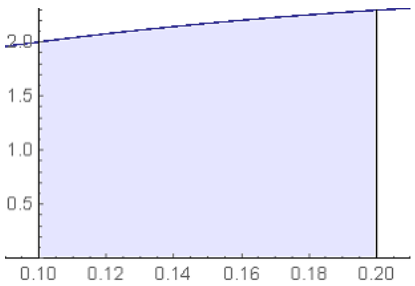


Figure 6.17 - Graphical representation of integral

Once we have found the expected number of all requests during one refresh period ($E[ReqNumber_{Total}(T, t_g)]$), we can easily express the refresh ratio:

$$RR(T, t_g) = \frac{1}{E[ReqNumber_{Total}(T, t_g)]} \quad \textbf{(Eq. 6.35)}$$

After we have derived an equation for finding the total hit rate and the fraction of triggered requests in a 2SLA policy, we need to move towards finding hit rates for SLA1 and SLA2 separately, as these are the only missing parts for finding the expected cost of a planned period.

The hit rate for SLA1 in 2SLA policy can be defined in a similar way to what we did for total hit rate:

$$HR_{SLA1}(T, t_g) = \frac{E[Hits_{SLA1}(T, t_g)]}{E[ReqNumber_{SLA1}(T, t_g)]} \quad \textbf{(Eq. 6.36)}$$

In the expression above, the number of SLA1 hits ($Hits_{SLA1}$) and the number of SLA1 requests. Request ($ReqNumber_{SLA1}$) are the expected average values per one real refresh period.

We start with finding the hit rate for SLA1. Finding $E[Hits_{SLA1}(T, t_g)]$ is trivial as hits can happen only during the first phase.

For finding the number of SLA1 requests during one real refresh period ($E[ReqNumber_{SLA1}(T, t_g)]$) we need to consider situations that can happen during all the three phases.

During phase 1, there is no possibility to get a miss; consequently, the amount of requests is equal to the amount of SLA1 hits. Unfortunately, in phase 2, we cannot use the same approach we used for finding the hit rate for policy with one SLA. The situation is more complex, as there is a chance that there will be no requests during the second phase, but the first request which will happen during the third phase will be an SLA1 request.

In the second phase, a miss can occur only once, and if it occurs, it can be only an SLA1 miss, as SLA2 is still in the boundaries of its freshness period. Consequently, in phase 2, the probability of meeting an SLA1 request, (as well as the expectation of the number of requests), is just a probability of the first arrival for a Poisson process with $\lambda = \lambda_1$:

$$P(tmiss \leq t_2) = 1 - e^{-\lambda_1 \Delta t_2} \quad \textbf{(Eq. 6.37)}$$

We are using Δt_2, as during the first phase, the probability was equal to zero and only started to grow as the process reached the point t_1.

In phase 3, the expected number of requests is also not more than one, as any request will trigger a refresh. However, SLA2 requests in this phase can also interfere. The expected number of SLA1 requests in phase 3 will be a product of three components: (i) the probability that there was no SLA1 requests during phase 2, (ii) the probability of the arrival of an SLA1 request before an SLA2 request in phase 3, (iii) and the probability of arrival of any (SLA1&2) requests during the phase 3 before the moment of automatic retrieval.

For the second component, we use a property of interfering Poisson processes, which states that a resulting process is also a Poisson process with intensity $\lambda = \lambda 1 + \lambda 2$ and probability of arrival of the $P(X_1 < X_2) = \frac{\lambda_1}{\lambda_1+\lambda_2}$.

Eventually, the expected number of SLA1 requests during one refresh period can be found as:

$$\begin{aligned} E\left[ReqNumber_{SLA1}(T, t_g)\right] \\ &= E[Hits_{SLA1}(\Delta t_1)] + P_{miss}(t_{missSLA1} \le t_2) \\ &+ P_{missSLA1}(t_{miss} > t_2) \times P(X_1 < X_2) \\ &\times P_{missSLA1\&2}(t_{miss} < t_3) \\ &= \lambda_1 \Delta t_1 + \left(1 - e^{-\lambda_1 1 \Delta t_2}\right) + e^{-\lambda_1 \Delta t_2} \times \frac{\lambda_1}{\lambda_1 + \lambda_2} \times (1 \\ &- e^{-(\lambda_1 + \lambda_2)\Delta t_3}) \end{aligned} \quad \textbf{(Eq. 6.38)}$$

Thus, we obtained the nominator and denominator for the $HR_{SLA1}(T, t_g)$ expression (Eq. 6.36).

Next, we can move to calculating the hit rate for SLA2. Similarly to the equation for HR(SLA1):

$$HR_{SLA2}(T, t_g) = \frac{E\left[Hits_{SLA2}(T, t_g)\right]}{\left(E\left[ReqNumber_{SLA2}(T, t_g)\right]\right)} \quad \textbf{(Eq. 6.39)}$$

On the other hand, now we have a more natural way to find the hit rate of SLA2. For the SLA1, we have to use the hits and requests expectations for a refresh period. However, for the last SLA, we can use an option, which is based on using all the hits and requests during the whole planning period:

$$HR_{SLA2}(T, t_g) = \frac{TotalHits_{SLA2}(T, t_g)}{TotalReqNumber_{SLA2}(T, t_g)} \quad \textbf{(Eq. 6.40)}$$

We already found expressions for Total hit ratio and SLA1 hit ratio. By multiplying them by the number of corresponding requests, we can get the total hit number and SLA1 hit number:

$$TotalHits_{SLA1\&2}(T, t_g) = HR_{Total} \times TotalReqNumber_{SLA1\&2} \quad \textbf{(Eq. 6.41)}$$

$$TotalHits_{SLA1}(T, t_g) = HR_{SLA1} \times TotalReqNumber_{SLA1} \quad \textbf{(Eq. 6.42)}$$

Now we can subtract the amount of hits SLA1 from the amount of all hits:

$$TotalHits_{SLA2}(T, t_g) = TotalHits_{SLA1\&2} - TotalHits_{SLA1} \quad \textbf{(Eq. 6.43)}$$

Eventually, we can find the hit rate for SLA2:

$$HR_{SLA2} = \frac{TotalHits_{SLA2}}{TotalReqNumberSLA2} \quad \textbf{(Eq. 6.44)}$$

Miss rate for SLA2 is, obviously:

$$MR_{SLA2} = 1 - HR_{SLA2} \quad \textbf{(Eq. 6.45)}$$

We have obtained all the required components for the total cost formula, which allows computing the estimated cost of a planned period depending on the chosen size of a gap.

Moreover, we can use the derived expression for the optimisation model to find the optimal gap size, which corresponds to the lowest cost of cache operation. We managed to convert the initial problem into a two-dimensional problem, where the cost of a planning period depends only on the choice of the gap between the end of the freshness period of the most expensive SLA, and the moment of planned retrieval. Hence, finding the optimal value is trivial. The process of estimating the cost of operation according to the algorithm and equations from this section, as well as finding the optimal size of the gap, is presented in Chapter 7 (Evaluation). Results of simulations, which are proving the correctness of the method, are also provided. In the next section, we will describe how higher levels of cache are taken into account.

6.4 CONCLUSION

In this chapter, we focused on the refresh rate -based caching strategies and models, where the cache decision is expressed as the amount of time until the next retrieval, after the end of the freshness period for the most expensive SLA. We introduced three main caching strategies for a single context attribute, which are (i) full coverage, (ii) reactive, and (iii) proactive strategies. We have shown how to estimate the cost for the first and the second

strategy based on the existing body of knowledge. The proactive strategy required the development of methods for cost estimation, which were presented in Section 6.3. While requiring a more sophisticated methodology, when realised, the proactive strategy can help to reach the maximum cost efficiency of the CSMS operation. The evaluation of the proposed methods is presented in Chapter 7.

Chapter 7: Evaluation

7.1 INTRODUCTION

In this chapter, we demonstrate how the proposed refresh rate-based models presented in the previous chapter can be used to reduce the load on the IoT Context Management Platform (CMP).

In Section 7.2, we evaluate the proposed caching models. The mathematical derivations of the models were presented in Chapter 6. In order to assess the correctness of the model from a practical side, we run a set of simulations to show how our theoretical findings are matching the results that we obtain from a simulation. In Section 7.2.1, we describe the setup of the experimental environment. In Section 7.2.2, we run simulations for 1SLA policy and in Section 7.2.3 for 2SLA policy to demonstrate how the predicted parameters at each phase match the results of our experiments. We also illustrate how a proactive caching strategy reduces the cost of CSMS operation in the case of an optimal choice of the retrieval rate.

7.2 REFRESH RATE -BASED CACHING MODELS EVALUATION

In this section, we show how the caching strategies and models, which were discussed and derived in Chapter 6, can be tested by a set of simulations. We describe a testing environment used; then we show an example of an analytic solution for 1SLA policy and 2SLA policies as well as the results of a simulation for these policies with similar parameters. Afterwards, we provide graphs which allow comparison of how the predicted values of the cost of operation are matching the values, which are obtained from the simulation for a wide variety of input parameters to show how the proposed theoretical method is matching the experimental results. We also provide graphical analysis of several cases, where each strategy can be beneficial for the cost efficiency of CSMS operation.

7.2.1 DESIGN AND EVALUATION OF THE SIMULATION TOOL

First, we describe the design of the environment, which is used for our tests and simulations. To simulate the query load, which would be coming from a query engine to CSMS in real life, we need to generate a stream of CDQL requests, which are aiming to access a context attribute with a certain level of freshness. According to the theoretical foundations and practical assumptions which we described in Chapter 2, Chapter 5 and Chapter 6, the distribution of request inter-arrival times should follow the Poisson law. We have chosen

jMeter, a widely accepted open-source tool used for load testing. The JMeter-based request generator was configured to produce a certain amount of requests per minute using the Poisson timer. Configuration scripts allow changing the freshness period for different SLAs as well as the gap size and all the costs. The process of running a test is shown in Figure 7.1.

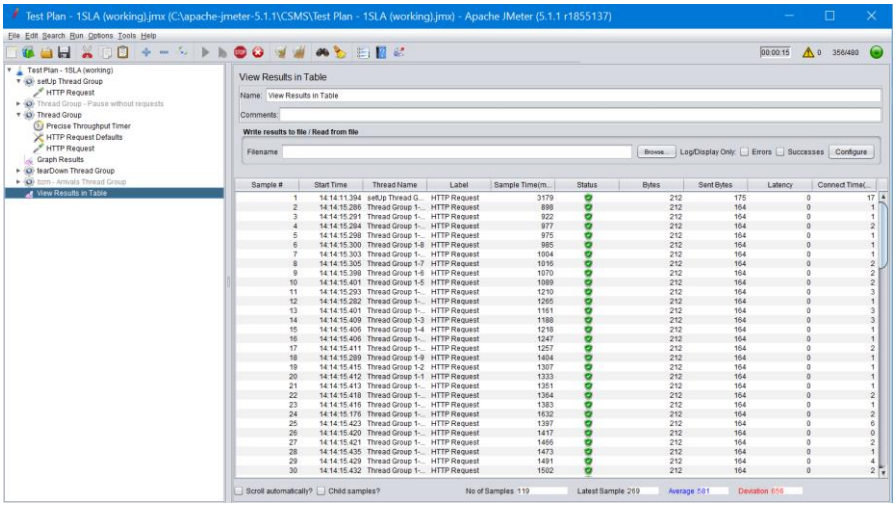


Figure 7.1 - JMeter –based request generator

The configuration of timers in jMeter is not entirely straightforward, which was causing a necessity to check if the resulting set of request inter-arrival times is exponentially distributed. To test this, we have run a test and built a graph of inter-arrival times, which is presented in Figure 7.2. According to the visualised results, the arrival process of requests corresponds to a Poisson process, and we can rely on the results of the simulation.

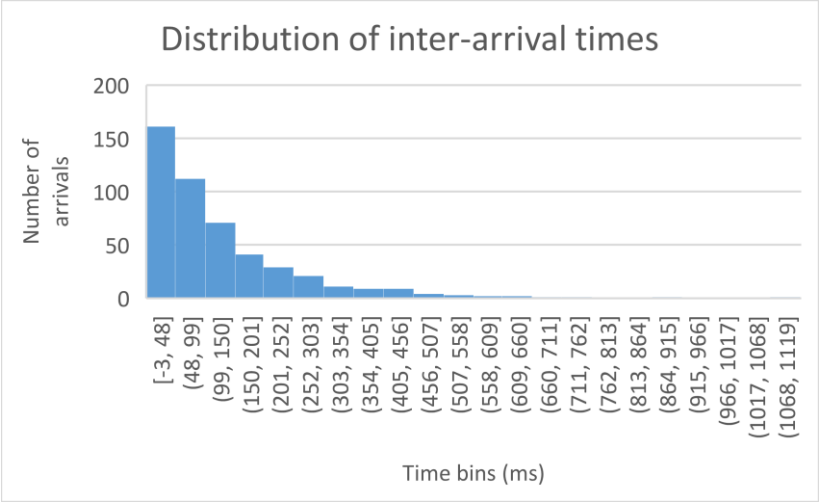


Figure 7.2 - Distribution of inter-arrival times of generated requests

Apart from the request generator, we needed a tool which will be registering requests and counting all the relevant parameters, such as hits and misses for all the SLAs, as well as the

number of refreshes and the overall cost of operation. For that reason, we developed a cache evaluation tool, which is implemented as a JavaEE web service. We intentionally separated experimental measurements from the real CoaaS platform prototype, to avoid potential influence caused by the interference of unforeseen factors. Eventually, the simulation environment consists of two main modules – the request generator, which is issuing requests, and the evaluation tool, which accepts arriving requests and computes the number of hits, misses, and associated costs based on the defined SLAs.

7.2.2 EVALUATION OF THE HMR METHOD FOR 1SLA POLICY

In this section, we provide an example of the analytical estimation of the cost for a planning period for a context attribute in a situation, where only one SLA is defined. Then, we compare each of the predicted components with the results of a simulation.

At first, we need to define the SLA by specifying the following components: the expiry period of the data item, the price of a request for a consumer, and the penalty, which is paid by CoaaS in the case when a cache miss happens. We also need to define additional input parameters like the cost of retrieval of a data item from a context provider to CoaaS, the length of a planning period and the expected number of request arrivals during the planning period. Then, we need to choose a gap size, which is the variable, directly influencing the strategy and, in turn, the final cost. The definitions of the terms above are provided in Chapter 6.

Let the SLA be the following:

Expiry period = 100 ms = 0.1 sec.

Request price = 80

Penalty = 180

Let the additional parameters be the following:

Retrieval cost = 75

Planning period = 60 sec.

Request arrival rate (λ) = 8 requests/sec

Let us choose the gap size t_g = 150ms = 0.15, (we measure the gap size between the end of the freshness period and the moment of retrieval).

We are deliberately not choosing a gap size which corresponds to the reactive or the proactive strategies, as these options are much simpler and, moreover, the proactive strategy is our main scope of interest due to a possibility of achieving the lowest cost of operation.

The result of a simulation for the provided input parameters is presented in Figure 7.3.

```
1  RequestNumber = 480 (number of incoming requests)
2  Hits = 257
3  Misses = 223
4  Hit rate (HR) = 0.54
5  Miss rate (MR) = 0.46
6  Refresh ratio RR(T, tg) = 0.66
7  RetrievalNum = 318 (All retrievals)
8  Automatic retrievals = 96
9  Triggered retrievals = 222
10 Retrieval cost = 23850.0
11 Penalty Cost = 35520.0
12 Total losses = 59370.0
13 Income = 38320.0
14 CostOfPlanningPeriod = -21050.0 (Overall cost)
15 Test Status: STOPPED
16 Expiry period: 100.0
17 Gap size = 150.0
18 Fetch Period: 250.0 (Expiry period + gap size)
19 Lambda: 8.01864406779661
```

Figure 7.3 - Results of the simulation for an 1SLA policy

Now we can compare the simulation results to the analytical solution.

According to the simplified version of the HMR formula (Eq. 6.13), the cost of a planning period for a proactive caching strategy with shifts can be found as:

$$\begin{aligned} CostOfPlanningPeriod \\ &= RequestNumber \times RequestPrice - \mathrm{MR}(T, t_g) \\ &\times RequestNumber \times Penalty - RR(T, t_g) \\ &\times RequestNumber \times RetrievalPrice \end{aligned} \qquad \textbf{(Eq. 7.1)}$$

The number of requests is, obviously, equal to:

$$RequestNumber = \lambda \times PlanningPeriod = 8 \times 60 = 480 \qquad \textbf{(Eq. 7.2)}$$

First, we need to find the hit and miss rates (Eq. 6.14), which can be achieved as follows:

$$HR(T, t_g) = \frac{E[N(T)]}{E[N(T)] + 1 - e^{-\lambda t_g}} = \frac{8 \times 0.1}{8 \times 0.1 + 1 - e^{-8 \times 0.15}} = 0.534 \qquad \textbf{(Eq. 7.3)}$$

Consequently, the number of hits during the planning period equals:

$$TotalHits = 0.534 * 480 = 256.3 \quad \textbf{(Eq. 7.4)}$$

The miss rate can be found as subtracting the hit rate from one, or as follows (Eq. 6.15):

$$MR(T, t_g) = \frac{1 - e^{-\lambda t_g}}{E[N(T)] + 1 - e^{-\lambda t_g}} = \frac{1 - e^{-8 \times 0.15}}{8 \times 0.1 + 1 - e^{-8 \times 0.15}} = 0.466 \quad \textbf{(Eq. 7.5)}$$

The number of misses during the planning period equals to:

$$TotalHits = 0.466 * 480 = 223.6 \quad \textbf{(Eq. 7.6)}$$

Then we need to find the ratio of retrievals to requests RR(T, t_g) (Eq. 6.17):

$$RR(T, t_g) = \frac{1}{E[N(T)] + 1 - e^{-\lambda t_g}} = \frac{1}{8 \times 0.1 + 1 - e^{-8 \times 0.15}} = 0.67 \quad \textbf{(Eq. 7.7)}$$

Consequently, the number of all retrievals during the planning period (Eq. 6.18) is:

$$RetrievalNum = RR(T, t_g) \times RequestNum = 0.67 \times 480 = 321 \quad \textbf{(Eq. 7.8)}$$

Eventually, the overall cost of the planning period is equal to:

$$\begin{aligned} CostOfPlanningPeriod &= 480 \times (80 - 0.466 \times 160 - 0.67 \times 75) \\ &= -21426.5 \end{aligned} \quad \textbf{(Eq. 7.9)}$$

As can be seen, the results of each step and the overall result are closely matching the results of the simulation.

We have completed the evaluation of a method for estimating the cost of the planning period for 1SLA policy, and demonstrated the matching of simulation results for the chosen gap size. Next, we build graphs of predicted and simulated results for various gap sizes, to show how the described method is matching the simulation results in a wide range of possible gap sizes.

In Figure 7.4, the graphs for predicted hit rate, miss rate, and refresh ratio are presented, while in Figure 7.5 the results of simulations for the corresponding gap sizes are presented. In Figure 7.6 and Figure 7.7, the results of estimated cost and cost, which was received as a result of the simulation are presented. As can be seen from the graphs, both predicted and simulated results are closely matching. We use the following colour codes and designations: hit rate and simulated hit rate are depicted in green and designated as HR(t) and SHR(t), miss rate and simulated miss rate are depicted in blue and designated as MR(t) and SMR(t), and refresh ratio

is depicted in yellow and designated as R(t) and SR(t). The overall predicted and simulated cost of planning period is depicted in orange.

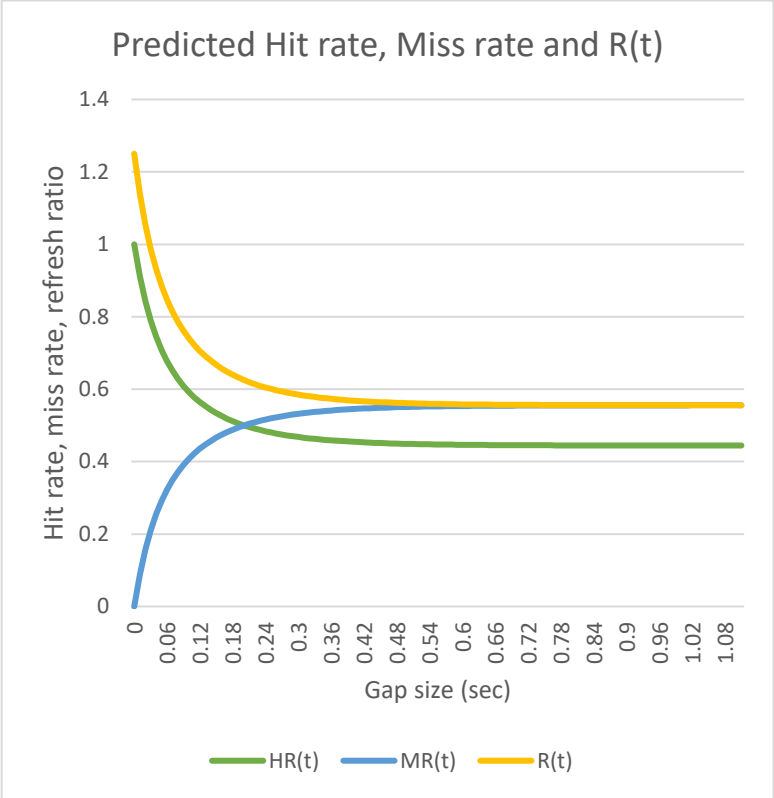


Figure 7.4 - Predicted Hit rate, Miss rate, and Refresh-Request ratio

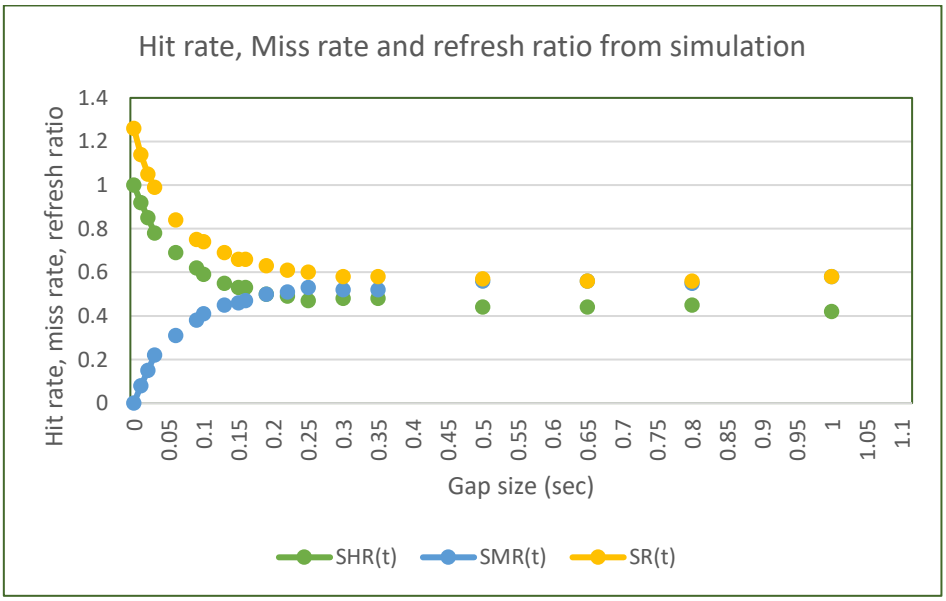


Figure 7.5 - Hit rate, Miss rate, and Refresh-Request ratio acquired from simulation

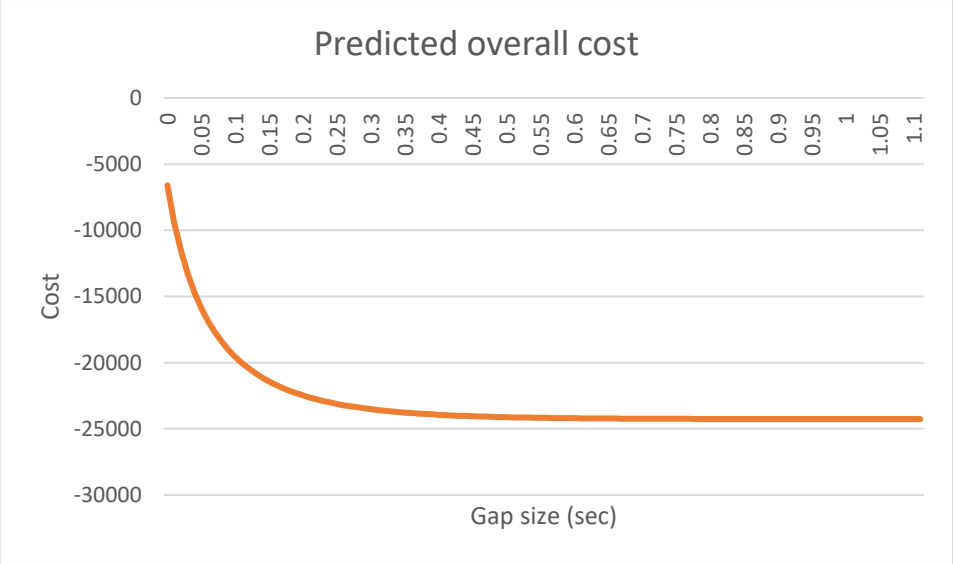


Figure 7.6 - Predicted total cost of operation

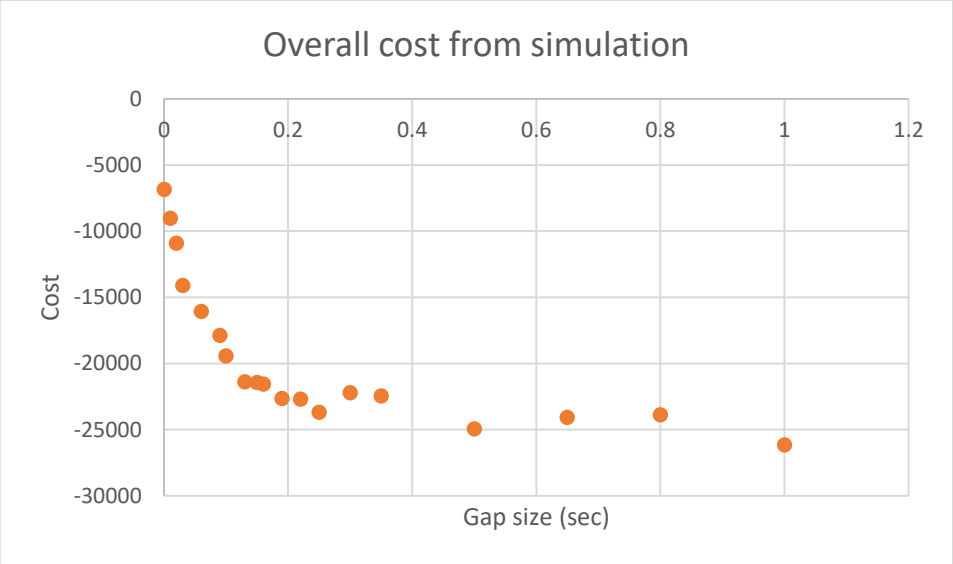


Figure 7.7 - Total cost of operation acquired from the simulation

In this set of experiments, we got negative costs. Technically, the cost of operation can be both, positive and negative, as we have included the income into the equation. The main objective is to keep the cost of operation as high as possible. When the highest achievable cost is still negative, a change in the pricing strategy might be required.

In the described experiment, the maximum of the objective function is achieved with a gap size which equals to zero, meaning that the full coverage is the most beneficial strategy for the current SLA and source behaviour. As it was said, in 1SLA policy, the maximum is always located at reactive strategy or full coverage strategy, and never in between. However, in a case

with multiple SLAs, the situation becomes more complicated. In any case, an analyst needs a methodology for informed decision while making and adaptation of the pricing strategy and SLA negotiation.

We have demonstrated the matching of results for 1SLA policy. In the next section, we are evaluating the model for more advanced policies.

7.2.3 EVALUATION OF HMR APPROACH FOR 2SLA POLICY

In this section, we evaluate the method for estimation of the cost of a planning period in case of a 2SLA policy according to the methodology which was proposed in Chapter 6.

To define the task, we need to add several parameters to the formulation of the task for the 1SLA evaluation. In the SLA section, we need to add separate SLA1 and SLA2 expiry periods and different costs of access to context attributes for SLA1 and SLA2 from the consumer side. We also need to define such parameters as the expected rate of request arrivals for SLA1 and SLA2 during the planning period.

Let the SLA be the following:

SLA1 Expiry period = 100 ms = 0.1 sec.

SLA2 Expiry period = 250 ms = 0.25 sec. (Expiry period is measured from t = 0, refer to diagrams presented in Chapter 6)

Request price SLA1 = 80

Request price SLA2 = 40

Penalty SLA1 = 160

Penalty SLA2 = 80

Let the additional parameters be the following:

Retrieval cost = 75

Planning period = 60 sec.

Request arrival rate SLA1 ($\lambda 1$) = 4 requests/sec

Request arrival rate SLA2 ($\lambda 2$) = 8 requests/sec

For the current evaluation, we are choosing the most complex scenario, when the moment of planned retrieval is located in phase 3. In other words, there is a time gap between the longest expiry period, (cheapest SLA, which is SLA2), and time at which a retrieval is planned.

Let us choose the gap size t_g = 100ms = 0.1 sec. We measure the gap size starting from the end of the freshness period of SLA2.

The results of the simulation for 2SLA policy are presented in Figure 7.8. We will be comparing each step in the analytical solution with the result of a simulation to verify the correctness of the method.

```
1  RequestNumber (total) = 718 (number of incoming requests)
2  RequestNumber (Phase 1) = 308
3  RequestNumber (Phase 2) = 315
4  RequestNumber (Phase 3) = 92
5  Hits (total) = 519
6  Hits (total Phase 1) = 307
7  Hits (total Phase 2) = 211
8  Hits (total Phase 3) = 0
9  Misses (total) = 200
10 Hit rate (total) = 0.72
11 Miss rate (total) = 0.28
12 RetrievalNum = 244 (All retrievals)
13 Automatic retrievals = 44
14 Triggered retrievals = 200
15 RequestNumber (SLA1) = 240
16 Hits (SLA1) = 103
17 Misses (SLA1) = 137
18 Misses (SLA1) In Phase 2 = 108
19 Misses (SLA1) In Phase 3 = 29
20 Hit rate SLA1 (MR_SLA1) = 0.42916666666666664
21 Miss rate SLA1 (MR_SLA1) = 0.5708333333333333
22 Triggered retrievals (SLA1) = 137
23 Requests (SLA2) = 477
24 Hits (SLA2) = 416
25 Misses (SLA2) = 63
26 Hit rate SLA2 (HR_SLA2) = 0.8684759916492694
27 Miss rate SLA2 (MR_SLA2) = 0.1315240083507307
28 Triggered retrievals (SLA2) = 63
29 Retrieval cost (total) = 18300.0
30 Penalty Cost (total) = 26960.0
31 Total losses = 45260.0
32 Income (total) = 38280.0
33 CostOfPlanningPeriod = -6980.0 (Overall cost)
34 Test Status: STOPPED
35 Expiry period (SLA1): 100.0
36 Expiry period (SLA2): 250.0
37 Gap size = 100.0 (after the end of SLA2 expiry)
38 Fetch Period: 350.0
39 Lambda (total): 12.169491525423728
40 Lambda (SLA1): 4.067796610169491
41 Lambda (SLA2): 8.084745762711865
```

Figure 7.8 - Simulation results for 2SLA strategy

According to Chapter 6 (Eq. 6.21), the cost of a planning period can be found as:

$$
\begin{aligned}
&CostOfPlanningPeriod \\
&\quad = RequestNumber_{SLA1} \times RequestPrice_{SLA1} \\
&\quad + RequestNumber_{SLA2} \\
&\quad \times RequestPrice_{SLA2} - MR_{SLA1}(T, t_g) \\
&\quad \times RequestNumber_{SLA1} \times Penalty_{SLA1} - MR_{SLA2}(T, t_g) \\
&\quad \times RequestNumber_{SLA2} \times Penalty_{SLA2} - RR(T, t_g) \\
&\quad \times \ RequestNumberAll \times RetrievalPrice
\end{aligned}
\qquad \textbf{(Eq. 7.10)}
$$

For being able to apply the expression above, we need to find the separate hit rate and miss rate for SLA1 and SLA2. We also need to find the refresh to request ratio, which is common for both SLAs. However, according to the method defined in Chapter 6, we need to start with finding the total hit and miss rate first.

The total hit rate can be found (Eq. 6.25) as follows:

$$HR(T, t_g) = \frac{E[Hits_{Total}(T, t_g)]}{E[ReqNumber_{Total}(T, t_g)]} \qquad \textbf{(Eq. 7.11)}$$

In the expression above $E[Hits_{Total}(T, t_g)]$ is an expected number of hits of all SLAs requests arriving during one refresh period, and $E[ReqNumber_{Total}(T, t_g)]$ is the expected number of requests of all SLAs during one refresh period.

The expected number of all hits on one refresh period (Eq. 6.30, Eq. 6.31) can be found as:

$$
\begin{aligned}
&E\left[Hits_{Total}(T, t_g)\right] \\
&\quad = E[Hits(\Delta t_1)] + \left(P(t_{miss} > t_2)\right) \times E[Hits_{SLA2}(\Delta t_2)] \\
&\quad + \int_{t=t_1}^{t_2} (fmiss(t - t_1) \times E[Hits_{SLA2}(t - t_1)])dt \\
&\quad = (\lambda_1 + \lambda_2) \times \Delta t_1 + \left(1 - \left(1 - e^{-\lambda_1 \Delta t_2}\right)\right) \times (\lambda_2 \Delta t_2) \\
&\quad + \int_{t=t_1}^{t_2} \lambda_1 e^{-\lambda_1 (t - t_1)} \lambda_2 (t - t_1) dt \qquad \textbf{(Eq. 7.12)} \\
&\quad = (\lambda_1 + \lambda_2) \times \Delta t_1 + \left(1 - \left(1 - e^{-\lambda_1 \Delta t_2}\right)\right) \times (\lambda_2 \Delta t_2) \\
&\quad + \frac{\lambda_2 - \lambda_2 e^{\lambda_1 (t_1 - t_2)} (\lambda_1 (t_2 - t_1) + 1)}{\lambda_1} \\
&\quad = (4 + 8) \times 0.1 + (1 - e^{-4 \times 0.15}) \\
&\quad + \frac{8 - 8e^{4(0.1 - 0.25)} (4(0.24 - 0.1) + 1)}{4} = 2.11
\end{aligned}
$$

The obtained result matches the simulation results:

$$Avg\left[Hits_{Total}(T, t_g)\right]_{sim} = \frac{519}{244} = 2.1$$

We can also separately test two parts of the expression above. The first part represents the number of hits during phase 1 (Eq. 6.26):

$$E[Hits_{Ph1}(T, t_g)] = (\lambda_1 + \lambda_2) \times \Delta t_1 = (4 + 8) \times 0.1 = 1.2 \qquad \textbf{(Eq. 7.13)}$$

From the simulation results, dividing the number of hits during phase 1 by the number of refreshes gives us:

$$Avg[Hits_{Ph1}]_{sim} = \frac{307}{244} = 1.2, \qquad \textbf{(Eq. 7.14)}$$

The received result matches the expectation.

The second part represents the expectancy of total hits during the second phase (Eq. 6.27-Eq. 6.29).

$$
\begin{aligned}
&E\left[Hits_{TotalPh2}(T, t_g)\right] \\
&\quad = \left(1 - \left(1 - \ e^{-\lambda 1 \Delta t 2}\right)\right) \times (\lambda 2 \Delta t 2) \\
&\quad + \frac{\lambda_2 - \ \lambda_2 e^{\lambda_1 (t_1 - t_2)} (\lambda_1 (t_2 - t_1) + 1)}{\lambda_1} \\
&\quad = (1 - (1 - e^{-4 \times 0.15})) \times (8 \times 0.15) \\
&\quad + \frac{8 - \ 8e^{4(0.1 - 0.25)} (4(0.25 - 0.1) + 1)}{4} = 0.9
\end{aligned}
\qquad \textbf{(Eq. 7.15)}
$$

From the simulation results, dividing the number of hits during phase 2 by the number of refreshes gives us:

$$
Avg\left[Hits_{Ph2}(T, t_g)\right]_{sim} = \frac{211}{244} = 0.86 \qquad \textbf{(Eq. 7.16)}
$$

The received result matches the expectation.

We have successfully tested the expectancy of total hits during one refresh period.

The second component of the expression for finding the total hit rate is the expected number of arriving requests during one refresh period (Eq. 6.32 - Eq. 6.34), which can be found as follows:

$$
\begin{aligned}
&E\left[ReqNumber_{Total}(T, t_g)\right] \\
&\quad = \ (\lambda_1 + \ \lambda_2) \times \Delta t_1 + \left(1 - \left(1 - \ e^{-\lambda_1 \Delta t_2}\right)\right) \times (\lambda_2 \Delta t_2) \\
&\quad + \ \int_{t=t_1}^{t_2} \lambda_1 e^{-\lambda_1 (t - t_1)} \ \lambda_2 (t - t_1) dt + \ (1 - (1 - e^{-\lambda_1 \Delta t_2})) \\
&\quad \times (1 - e^{-(\lambda_1 + \lambda_2) \Delta t_3}) \\
&\quad = \ (\lambda_1 + \ \lambda_2) \times \Delta t_1 + \left(1 - \left(1 - \ e^{-\lambda_1 \Delta t_2}\right)\right) \times (\lambda_2 \Delta t_2) \\
&\quad + \frac{-e^{\lambda_1 (t_1 - t_2)} (\lambda_1 \lambda_2 (t_2 - t_1) + \lambda_1 + \lambda_2) + \lambda_1 + \lambda_2}{\lambda_1} \\
&\quad + \ \left(1 - \left(1 - e^{-\lambda 1 \Delta t 2}\right)\right) \times \left(1 - e^{-(\lambda 1 + \lambda 2) \Delta t 3}\right) \\
&\quad = \ (4 + 8) \times 0.1 + \left(1 - (1 - \ e^{-4 \times 0.15})\right) \times (8 \times 0.15) \\
&\quad + \frac{-e^{4(0.1 - 0.25)} (4 \times 8(0.25 - 0.1) + 4 + 8) + 4 + 8}{4} \\
&\quad + \ (1 - (1 - e^{-4 \times 0.15})) \times (1 - e^{-(4 + 8) \times 0.1}) = \ 2.93
\end{aligned}
\qquad \textbf{(Eq. 7.17)}
$$

From the simulation results, we can see that the average number of requests per one refresh period equals to:

$$Avg\left[ReqNumber_{Total}(T,t_g)\right]_{sim} = \frac{718}{244} = 2.94 \quad \textbf{(Eq. 7.18)}$$

The result is closely matching the result predicted by the analytical solution above.

Then, the total hit rate equals to:

$$HR_{Total}(T,t_g) = \frac{E[Hits_{Total}(T,t_g)]}{E[ReqNumber_{Total}(T,t_g)]} = \frac{2.11}{2.94} = 0.72, \quad \textbf{(Eq. 7.19)}$$

The received result is matching the result of a simulation.

Total miss rate is, obviously:

$$MR_{Total} = 1 - HR_{Total} = 1 - 0.72 = 0.28 \quad \textbf{(Eq. 7.20)}$$

The refresh rate (Eq. 6.35) can be found as:

$$RR(T,t_g) = \frac{1}{E[ReqNumber_{Total}(T,t_g)]} = \frac{1}{2.93} = 0.34 \quad \textbf{(Eq. 7.21)}$$

which is matching the result of the simulation:

$$RR_{sim} = \frac{FetchNumber}{ReqNumber_{Total}} = \frac{244}{718} = 0.34 \quad \textbf{(Eq. 7.22)}$$

We have validated the method for finding total hit and miss rate for the 2SLA policy.

Next, we can find the separate hit rate for SLA1 (Eq. 6.36):

$$HR_{SLA1}(T,t_g) = \frac{E[Hits_{SLA1}(T,t_g)]}{E[ReqNumber_{SLA1}(T,t_g)]} \quad \textbf{(Eq. 7.23)}$$

The expected number of hits during one refresh period can be found in a simple way, as SLA1 hits can happen only during phase 1:

$$E[Hits_{SLA1}(T,t_g)] = \lambda_1 \Delta t_1 = 4 \times 0.1 = 0.4 \quad \textbf{(Eq. 7.24)}$$

This matches the result of the simulation:

$$Avg\left[Hits_{SLA1}(T,t_g)\right]_{sim} = \frac{Hits_{SLA1}}{FetchNumber} = \frac{103}{244} = 0.42 \quad \textbf{(Eq. 7.25)}$$

The expected number of SLA1 requests during one refresh period (Eq. 6.38) can be found as:

$$
\begin{aligned}
&E\left[ReqNumber_{SLA1}(T, t_g)\right] \\
&\quad = E[Hits_{SLA1}(\Delta t_1)] + P_{miss}(t_{missSLA1} \le t_2) \\
&\quad + P_{missSLA1}(t_{miss} > t_2) \times P(X_1 < X_2) \\
&\quad \times P_{missSLA1\&2}(t_{miss} < t_3) \\
&\quad = \lambda_1 \Delta t_1 + \left(1 - e^{-\lambda_1 1 \Delta t_2}\right) + e^{-\lambda_1 \Delta t_2} \times \frac{\lambda_1}{\lambda_1 + \lambda_2} \times (1 \\
&\quad - e^{-(\lambda_1 + \lambda_2)\Delta t_3}) \\
&\quad = 4 \times 0.1 + (1 - e^{-4 \times 0.15}) + e^{-4 \times 0.15} \times \frac{4}{4 + 8} \\
&\quad \times \left(1 - e^{-(4+8) \times 0.1}\right) = 0.97
\end{aligned}
\qquad \textbf{(Eq. 7.26)}
$$

The result of this step is also closely matching the simulation:

$$
\begin{aligned}
Avg\left[ReqNumber_{SLA1}(T, t_g)\right]_{sim} &= \frac{ReqNumber_{SLA1}}{FetchNumber} = 240/244 \\
&= 0.98
\end{aligned}
\qquad \textbf{(Eq. 7.27)}
$$

Eventually, the hit rate for SLA1 is equal to:

$$
HR_{SLA1}(T, t_g) = \frac{E[Hits_{SLA1}(T, t_g)]}{E\left[ReqNumber_{SLA1}(T, t_g)\right]} = \frac{0.4}{0.98} = 0.408 \qquad \textbf{(Eq. 7.28)}
$$

The miss rate for the SLA1 can be found as follows:

$$
MR_{SLA1}(T, t_g) = 1 - HR_{SLA1}(T, t_g) = 1 - 0.4 = 0.6 \qquad \textbf{(Eq. 7.29)}
$$

Hit and miss rates for SLA1 which were obtained by simulation are also close to 0.4 and 0.6; consequently, we have successfully tested expressions for SLA1 hit and miss rates.

Next, we can find the hit and miss rate for SLA2.

To find the hit rate for SLA2, we are using the approach based on the whole planning period (refer to Chapter 6), which is more straightforward than the method we used for SLA1. However, we only can use this approach for the last SLA, but not for all of them. The hit rate for SLA2 (Eq. 6.39) can be found as:

$$
HR_{SLA2}(T, t_g) = \frac{TotalHits_{SLA2}(T, t_g)}{TotalReqNumber_{SLA2}(T, t_g)} \qquad \textbf{(Eq. 7.30)}
$$

As we already know the total hit rate, we can use it to find the total number of hits based on Eq. 6.41:

$$TotalHits_{SLA1\&2}(T, t_g) = HR_{Total} \times TotalReqNumber_{SLA1\&2} = 0.72 \times 720 = 518.4 \quad \textbf{(Eq. 7.31)}$$

Then, we find the number of hits for SLA1 (Eq. 6.42) in a similar manner:

$$TotalHits_{SLA1}(T, t_g) = HR_{SLA1} \times TotalReqNumber_{SLA1} = 0.4 \times 240 = 96 \quad \textbf{(Eq. 7.32)}$$

Based on that, we find the number of hits for SLA2 (Eq. 6.43):

$$TotalHits_{SLA2}(T, t_g) = TotalHits_{SLA1\&2} - TotalHits_{SLA1} = 518.4 - 96 = 422.4 \quad \textbf{(Eq. 7.33)}$$

Next, we find the hit and miss rates for SLA2 (Eq. 6.44):

$$HR_{SLA2} = \frac{TotalHits_{SLA2}}{TotalReqNumberSLA2} = \frac{422.4}{480} = 0.88 \quad \textbf{(Eq. 7.34)}$$

$$MR_{SLA2} = 1 - HR_{SLA2} = 1 - 0.88 = 0.12 \quad \textbf{(Eq. 7.35)}$$

The results from the simulation are: $HR_{SLA2}sim = 0.87$ and $MR_{SLA2}sim = 0.13$, which means the predicted values are very close to the simulation results.

Now we have obtained all the components and we can estimate the cost of a planned period (Eq. 6.21):

$$\begin{aligned} CostOfPlanningPeriod &= RequestNumber_{SLA1} \times RequestPrice_{SLA1} \\ &+ RequestNumber_{SLA2} \\ &\times RequestPrice_{SLA2} - MR_{SLA1}(T, t_g) \\ &\times RequestNumber_{SLA1} \times Penalty_{SLA1} - MR_{SLA2}(T, t_g) \\ &\times RequestNumber_{SLA2} \times Penalty_{SLA2} - RR(T, t_g) \\ &\times RequestNumberAll \times RetrievalPrice \\ &= 240 * 80 + 480 * 40 - 240 * 0.6 * 160 - 480 * 0.12 \\ &* 80 - 0.34 * 720 * 75 = -7608 \end{aligned} \quad \textbf{(Eq. 7.36)}$$

The result is close to the value of the overall cost obtained from the simulation. Thus, we have finalised the evaluation of the proposed method for estimating the cost of a planning period for 2SLA policy for a chosen gap size. Next, we will present the results of the analytical solution and its graphical comparison with the result of a simulation, similarly how we did it for 1SLA policy.

We use the following colour codes and designations: hit rate and simulated hit rate are designated as HRn(t) and SHRn(t) and depicted in dark green and light green for SLA1 and SLA2 correspondingly. Miss rate and simulated miss rate are designated as MRn(t) and SMRn(t) and are depicted in light blue and dark blue correspondingly. The refresh ratio is depicted in yellow and designated as R(t) and SR(t). The overall predicted and simulated cost of planning period for 2SLA policy is depicted in orange.

From the graphs presented in Figure 7.9, Figure 7.10, Figure 7.11 and Figure 7.12, it can be seen that the shapes and numerical values of the results of the analytical solution are closely matching the results of simulations.

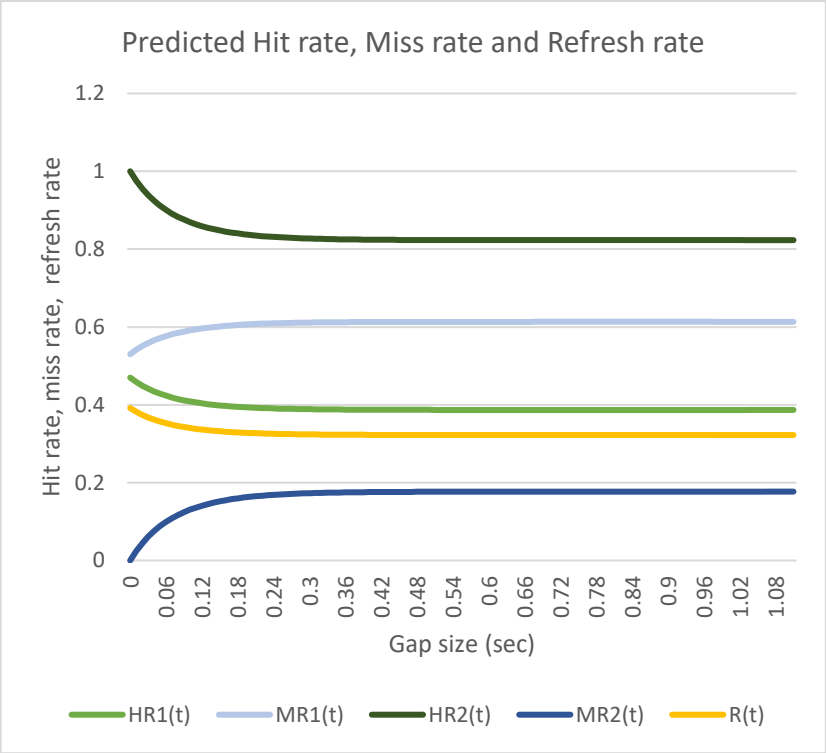


Figure 7.9 - Predicted Hit rate, Miss rate, and Refresh-Request ratio

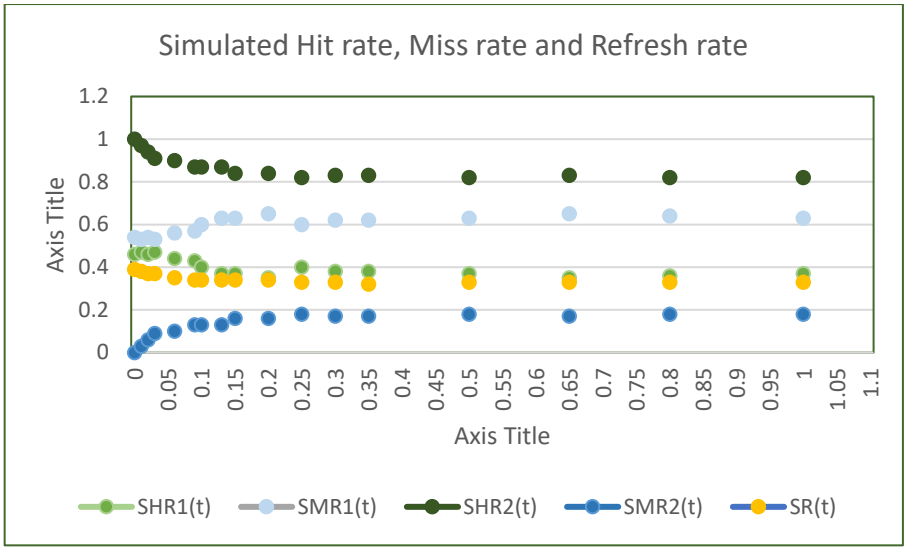


Figure 7.10 - Hit rate, Miss rate, and Refresh-Request ratio acquired from the simulation

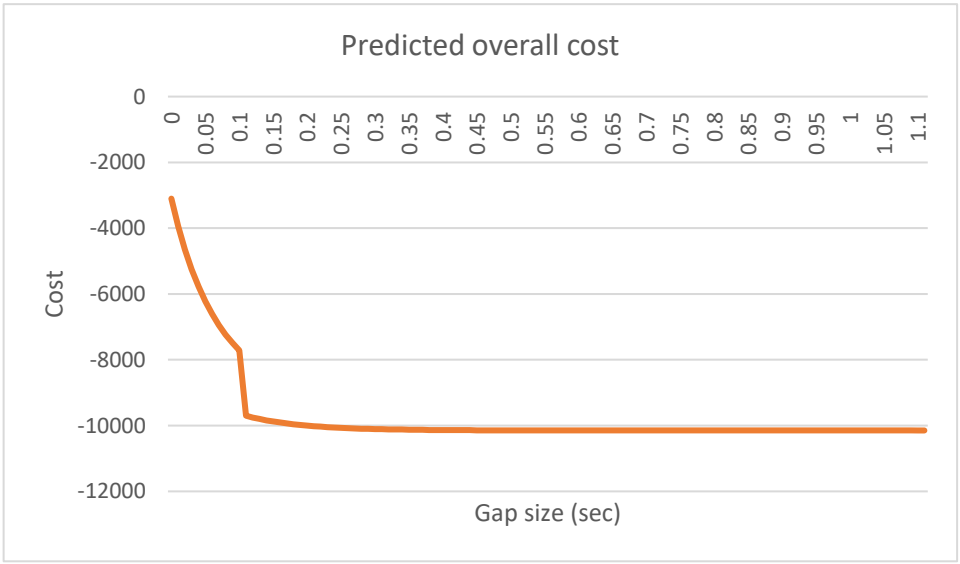


Figure 7.11 - Predicted total cost of operation

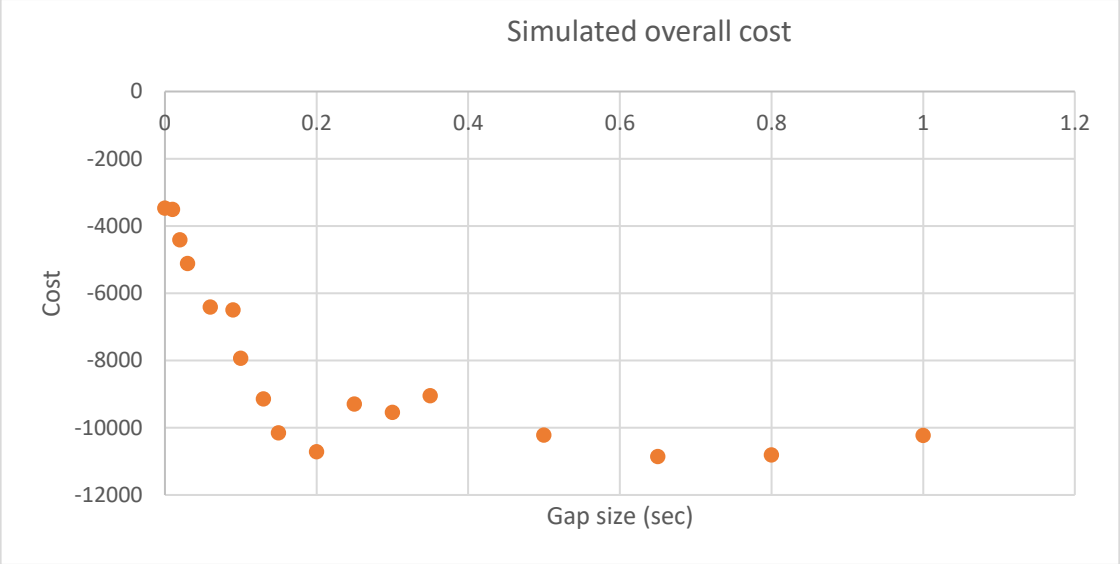


Figure 7.12 - Total cost of operation acquired from the simulation

We have demonstrated the matching of predicted with analytical solutions results with the results of a simulation for 2SLA policy.

In the described experiment, the maximum of the objective function is achieved with a gap size which equals to zero, meaning that the full coverage is the most beneficial strategy for defined SLAs and input parameters. Next, we demonstrate how a change in only one parameter can dramatically change the situation.

For instance, let the retrieval price be equal to 85. A corresponding graph is presented in Figure 7.13. The most efficient point of operation is still with gap size equal to zero, the shape of the graph is slightly different to the shape shown in Figure 7.11, where the price of retrieval was equal to 75.

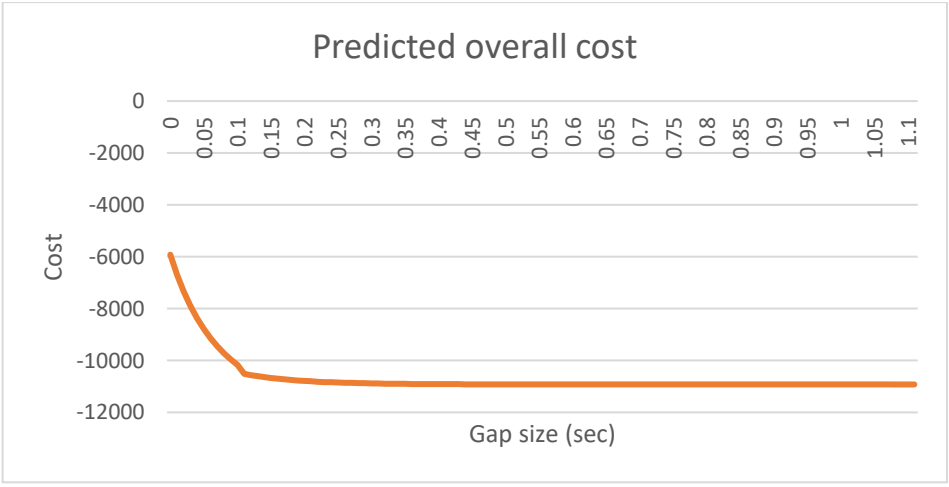


Figure 7.13 - Predicted cost for retrieval price = 85

However, if we choose the retrieval price equal to 100, the shape of the graph is changing, as it is shown in Figure 7.14. The point of maximal efficiency is still at gap size equal to zero (full coverage strategy). The cost of operation grows with the increase in gap size. However, after gap size = 0.1, we can see a steep decrease in the cost, and the cost is keeping almost stable afterwards.

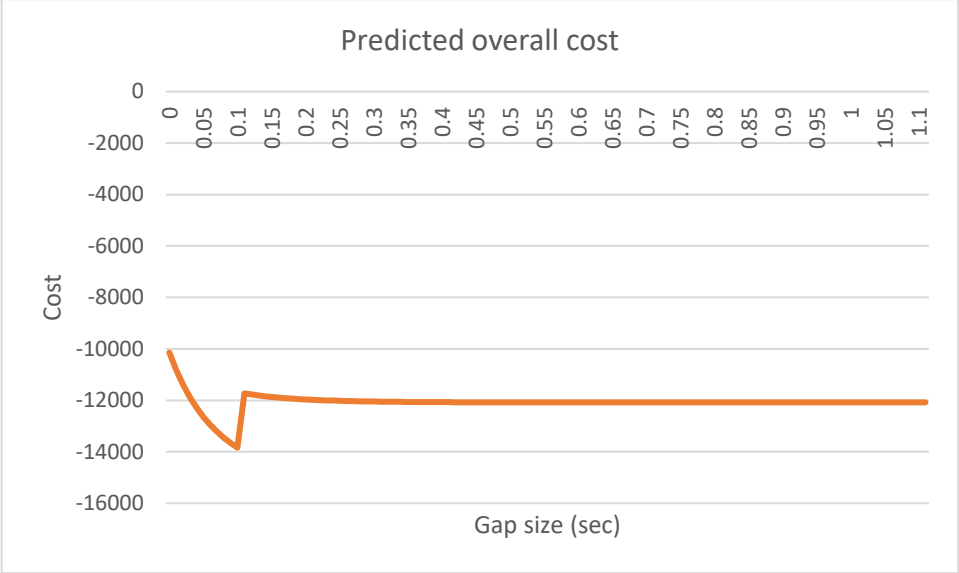


Figure 7.14 - Predicted cost for retrieval price = 100

Next, we can analyse another graph, assuming that the retrieval price is equal to 160. The corresponding graph is presented in Figure 7.15. Here, the situation is completely different. With gap size equalling zero, the objective function has a local minimum, then the cost grows, and after gap size 0.1, there is a steep decrease in cost. After this point, cost slightly grows again and then keeps stable. In general, we can say that in this situation, a proactive strategy is more beneficial than a full coverage strategy or a reactive strategy.

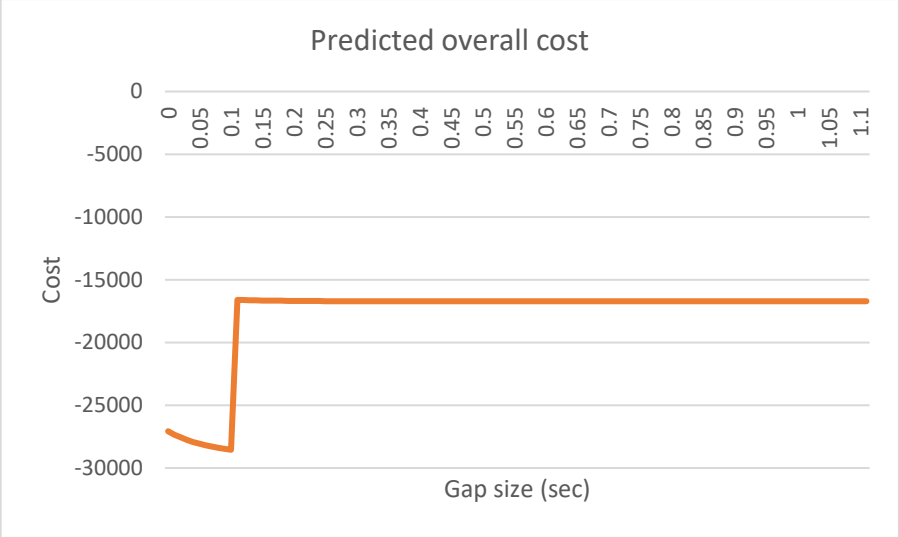


Figure 7.15 - Predicted cost for retrieval price = 160

With the input parameters used, the benefit of the proactive strategy is not huge over the cost of the reactive strategy. However, if we change the penalty for SLA1 to 600, as it is shown in Figure 7.16, the cost in the optimal point (t = 0.1) is significantly higher than with the full coverage strategy or with the reactive strategy. Consequently, this strategy is significantly more beneficial with given input parameters.

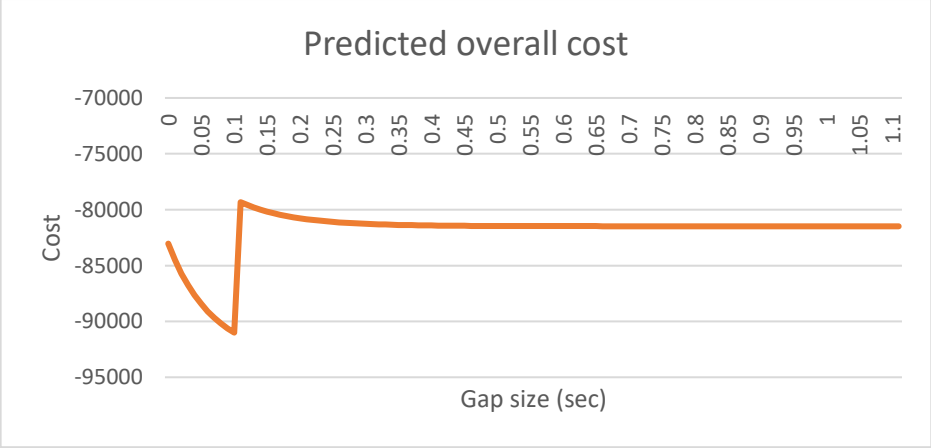


Figure 7.16 - Predicted cost for retrieval price = 160 and SLA1 penalty = 600

Next, we returned the SLA2 penalty parameter back to 160 and changed the price of retrieval to 300. The result is presented in Figure 7.17. In this case, the cost goes down exponentially till t = 0.1, and then there is a steep decrease. After the steep decrease, the cost slightly decreases and then keeps at the same level. The provided situation is an example where the reactive strategy is beneficial.

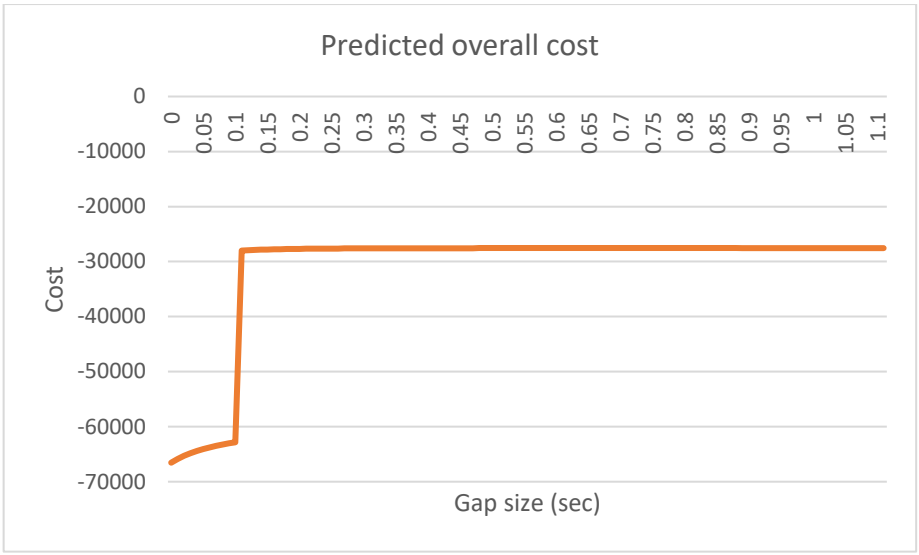


Figure 7.17 - Predicted cost for retrieval price = 300

With the current input parameters, the benefit of the reactive strategy is not huge over proactive retrieval with gap size t=0.1.

However, if we again change the input parameters and set $\lambda 2 = 80$, the difference, which is shown in Figure 7.18, becomes more obvious. In this case, it is clear from the graph that the most efficient strategy is to use a reactive strategy.

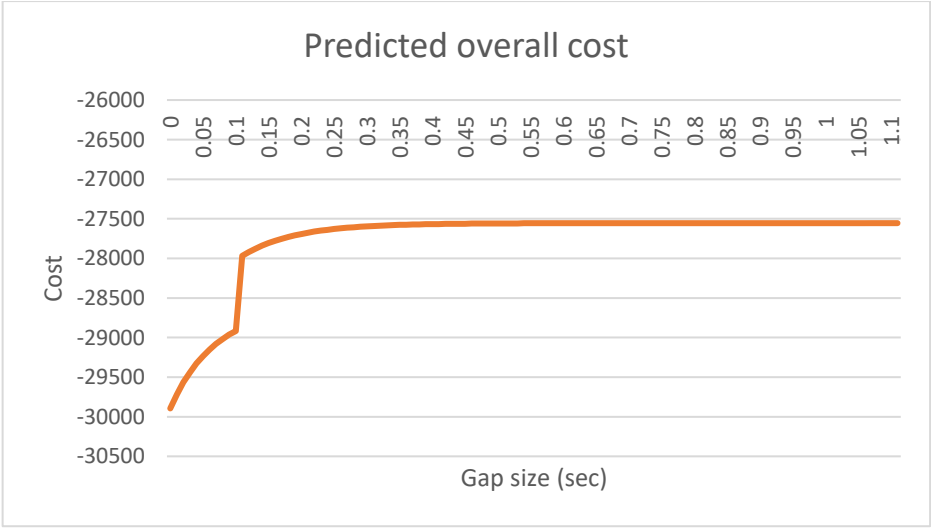


Figure 7.18 - Predicted cost for retrieval price = 300 and $\lambda 2 = 80$

With the graphs above, we have demonstrated how different can be the behaviour of the objective function with various input parameters. This behaviour can be significantly influenced by the proper choice between available strategies, and, in the case where a proactive strategy is chosen, the choice of the right size of a gap.

7.3 CONCLUSION

In this chapter, we have evaluated the CSMS refresh rate-based caching strategies by comparing the analytical methods, which we proposed in Chapter 6 with the results of the simulation. We have described the simulation environment, which consists of a specially configured JMeter-based request injection tool and custom request registration tool. Then, we provided a step by step evaluation of our methods for 1SLA and 2SLA policies with fixed chosen gap size, to show how the results of each expression are matching the simulation results. We also provided a graphical comparison of the results for various gap sizes and other input parameters, in order to prove the workability of the proposed methods with a variety of parameters. We have shown that there exist input parameters with which every strategy (full coverage, proactive and reactive) is beneficial, and it is essential to have a methodology for the optimal decision making.

We found that the proposed methods are producing reliable results which are closely matching the results of a simulation.

Chapter 8: Conclusion

8.1 INTRODUCTION

In this chapter we present a summary of the project and contributions. We also discuss the directions for extending the proposed solution and theory.

This chapter is structured in the following way: In Section 8.1, we briefly bring to mind the evolution of the bIoTope and CoaaS projects, which formed the background for this research. Section 8.2 provides the discussion of research outcomes, including the architecture of CSMS, its implementation and integration with other parts of the CoaaS platform. This section also provides the analysis of obtained results in the area of cache management strategies, which were investigated in this project. Section 8.3 contains the discussion of the directions for future work.

8.1.1 CONTEXT-AS-A-SERVICE (COAAS) PLATFORM AND BIOTOPE PROJECT

The Context-as-a-Service (CoaaS) platform project started as a part of the bIoTope project[15] in 2016. This project is a globally distributed community, which aims to foresee the future and propose steps to achieve it. The bIoTope project brought together representatives of various industries and sectors from different countries. These participants represented different types of mindset, as they were coming from such areas as academia, governmental services, managers of large and small enterprises, and founders of start-ups. We found that building the ecosystem consisting of devices requires, first of all, building the ecosystem of people that have intersecting interests. This ecosystem facilitated the crystallisation of concept, as well as our understanding of the desires and possibilities of different parties.

At the side of the CoaaS group, we had to define the scope of CoaaS and the concepts of primary interfaces. The Context Storage Management System (CSMS) project started at the same time with the development of the CoaaS platform and the Context Definition and Query Language (CDQL). We provided the essential background of CoaaS and CDQL in the first part of Chapter 3.

[15] https://biotope-project.eu/

During the bIoTope project, we worked in close contact with representatives of the BMW group, who provided valuable insights, which helped to shape the scope of the CoaaS platform. This discussion also resulted in a joint publication [60]. The CoaaS group had participated in facilitating such IoT scenarios as searching for carparks and electric charging stations. Aalto University, Helsinki, Finland hosted the test of the scenario. CoaaS was also used to facilitate the real trial of preconditioning of an electric connected BMW i3. The tests were conducted in Melbourne, Australia and hosted by Data 61, CSIRO. This work was also influenced by Eccenca GmbH[16], who provided the MobiVoc [46] semantic vocabulary, used to structure the descriptions of the carpark entity.

The CoaaS group also participated in facilitating the scenario of IoT-enabled waste management (WM), as well as in support of the scenario, where the intersection of WM with BMW vehicle routing was considered. The smart waste management scenario and the corresponding semantic vocabulary was maintained by the ITMO University, St. Petersburg, Russia.

In general, we can state that during the time of the project, CoaaS has moved from the stage of a not clearly defined concept to the stage of a working prototype, which has been tested with real consumers. The project meetings, consortium assemblies, and implementations and tests of scenarios provided the view on how the IoT ecosystem can evolve, and how the CoaaS platform should advance to comply with the challenges. Then, it became possible to move towards the individual part of the project.

We saw the challenges for the data storage and processing component of CoaaS in two main directions: (i) researching and developing the main modules, which can govern the storage of data in the CoaaS platform and facilitate the work of the Query Engine, and (ii) achieving the high efficiency of data storage and processing modules.

The data storage and processing component was called the Context Storage and Management System (CSMS). In the next Section, we summarise the details of the research conducted in this dissertation.

[16] https://www.eccenca.com/en/index.html

8.2 RESEARH OUTCOMES

In this section, we discuss the results and key contributions made by this dissertation, which were achieved during the development of the CSMS. In Section 8.2.1, we discuss the architectural side of the project. In Section 8.2.2, we discuss the results achieved in the area of caching strategies applicable for modern CMPs.

8.2.1 CONTEXT STORAGE AND MANAGEMENT SYSTEM: RESEARCH, ARCHITECTURE AND IMPLEMENTATION

In this section, we summarise the main contributions of this thesis in the area of architectures of storage component of IoT platforms, and the decisions made during the implementation of CSMS and its integration with other CoaaS platform components.

In Chapter 3, to address RQ1.1, we analysed the requirements to CSMS, which were dictated by the use cases, decisions made at the beginning of the CoaaS project, CDQL language, and state of the art in the area of IoT and context management platforms. This analysis led to the design of the CSMS architecture, which was presented in the second part of Chapter 3 to answer RQ1.

In Chapter 4, the details of design and implementation of CSMS were presented to address RQ1.2. The main component of CSMS was the Storage Query Execution Manager (SQEM). It was the entry point for CDQL requests which arrived from the CoaaS Query Engine (QE). SQEM was responsible for facilitating the execution of context request in the storage system. SQEM converted the query to the format of the corresponding underlying datastore. Moreover, it facilitated the process of treating the expiration and refreshment of context attributes, as well as handling the process of execution of functions which were contained in a CDQL query.

The main storage modules that were proposed and developed included: (i) the Context Service Description Repository (CSDR), which was responsible for finding the right sources of context to serve the query, (ii) the Context Repository (CR), which contained the cached context attributes, and (iii) the subscriptions repository (SR). SR was a part of a Situation Module (SM), which also contained the event stream processing engine. SM received context updates from providers, and all the events were percolated through the subscriptions, in order to trigger the execution of a registered query (CDQL push-based query), when the situation was detected.

The aforementioned principal components are essential for the functioning of the CSMS. Other parts of the architecture are needed for achieving the higher efficiency, for instance, cache management and proactive retrieval of raw data.

8.2.2 CACHING STRATEGIES AND MODELS FOR CSMS

In this section, we summarise our findings in the area of cache management and efficiency optimisation of CSMS operation, which were presented in Chapter 5-7 and helped to answer RQ1.

To address RQ1, we defined the main concepts and influencing parameters. The proposed theory and model were based on three main concepts: (i) Not only Database-Not only Redirector (NoD-NoR) mode of CMP operation, (ii) the unlimited amount of resources in a scalable cloud system, unlike fixed size systems, and (iii) the possibility to define requirements to the quality and cost of context through SLAs, established between a CMP and a context consumer, as well as between a CMP and a context provider. The SLA-based requirements included the freshness, latency of access, retrieval cost, processing cost, penalty and price of access. The computed parameters included the popularity, and reliability of context providers.

The NoD-NoR concept defined a system, which not only could answer queries based on the data from the internal storage, but also could retrieve the needed data from external providers on the fly. The second concept stated that as most modern middleware systems were hosted in the cloud, it was not efficient to develop caching strategies for getting the best performance from a certain amount of resources. On the contrary, it was beneficial to develop strategies which aimed at higher cost efficiency, as the affordability of using cloud resources and external services became the only limitation. The third concept linked the losses and income of the platform. Based on this link, the cost-efficiency model was developed.

To answer RQ2.3, we also defined the physical and logical dimensions of cache. The physical dimension represented the choice of caching data using the in-memory storage or using the disk-based storage. It also took into account the amount of data which should be handled by one server node. While the in-memory systems could reduce the latency of access, the cost of operation was also much higher than the operation cost of a disk-based system. The amount of data handled by one server node influenced the ingestion rate and search time. Spreading the load among several nodes helped to reduce the latency; however, using too many server nodes increased the cost of operation.

The logical dimension of cache represented caching the results of the CDQL query execution, based on the components of the query. The cache pyramid consisted of four levels, starting from the level of raw context attributes, and including results of functions execution, the results of CDQL requests execution, and, at the highest level, the results of full CDQL query execution.

To answer RQ2.2, in Chapter 6, we researched the ways to build a model for an individual context attribute. The three main strategies were (i) full coverage, (ii) proactive, and (iii) reactive. The proactive strategy defined the retrieval of an attribute from the external system after specific time since the data item had expired. This meant there was a time gap between periods when the incoming query could be served out of the cache. First, we developed models for a policy where only one SLA was defined (1SLA policy) and shown that for that type of policy, the proactive strategy was never beneficial. We also developed a method for predicting the cost of operation with any chosen strategy and any selected gap.

Then, we investigated the process taking into account the possibility of more than one SLA defined (nSLA). We illustrated the process on a 2SLA policy example. We showed that the model became significantly more complicated. Moreover, with nSLA policies, the proactive strategy could be the most cost-efficient, if the right gap size was chosen. We also developed a model to predict the cost of operation for 2SLA policy for a any selected gap size. We evaluated and illustrated the results in Chapter 7.

8.3 FUTURE WORK

In this section, we outline the future directions of the research which we see as the most promising for the advancement of CMPs in general, and CoaaS with CSMS in particular.

Real integrations

There is nothing more valuable than feedback from real consumers. As the preliminary work in the CMP area is already completed, it is the time to bring the prototypes of scenarios to a much bigger scale. The results achieved form the real large scale integrations will open new research gaps; these results will also reinforce the theory and practice of CMPs. We also expect the development of new business models and the development of corresponding ways of defining the agreements, (SLAs), between parties in the IoT ecosystems. If new agreements differ from those which were used as the base for the development of the caching model, corresponding adjustments would have to be made.

Taking access control into account

The current implementation of CoaaS, as well as many other prototypes of CMPs, does not support a flexible mechanism for access control. However, in real IoT ecosystem-wide scenarios, such mechanisms will have the highest importance, as otherwise, it will be hard to reach the agreement of context providers to participate in the ecosystem. The access control mechanisms will also have a massive impact on the design and efficiency of CSMS.

Extending the analytical framework: predictions and freshness

The freshness parameter, which was used for building and evaluating the caching model in Chapters 5-7, is an essential component for cache management in transient IoT environments. However, estimating the freshness of data items is not an easy task and requires a solid analytical framework to be built and integrated as a part of CSMS.

Another side of the analytical framework is supporting the scenarios, where predictions of future situations are needed.

Benchmarking and standardisation

Nowadays, every research group or company which proposes an IoT or a CMP platform, shows the results of performance evaluation on scenarios, which are beneficial for this particular platform. This situation leads to complexities in comparison of these platforms for an external integrator. Developing a solid, credible benchmarking methodology for IoT CMPs requires substantial R&D effort in both academic and industry IoT communities. We believe that mature benchmarking can significantly improve the evidence-based competition in the field of CMPs.

Standardisation of context definition and querying is always in the main scope of any IoT conference or meetup. The work in this direction should be continued with the active involvement of researchers and standardisation units.

Development of more advanced caching models

As the business strategies change or become more complex, there will be a need to adjust the current cache management strategy or develop a new model. Another direction is extending the model so that it would use the links between the context attributes. In the current model, all the context attributes were looked as individually. For instance, it might happen that several attributes can only be retrieved together from the provider for a joint price. It can also happen

that certain attributes are often requested by consumers in a certain order. Taking these complexities into account can enhance the cost efficiency of CSMS operation.

Application of ML techniques

As we have shown, the development of caching strategies, even for a single context attribute is a very labour-intensive and time consuming task. However, business requirements can change fast, and there is a need for adapting the cache strategies accordingly. This can potentially be achieved with the application of modern machine learning techniques. For instance, the application of reinforcement learning to CSMS caching tasks, and evaluation of its performance, is an interesting research area. At the same time, maintaining the understanding of processes through obtaining an analytical solution, instead of purely relying on the ML decisions, is also worth the effort, in our opinion.

Solving the inverse problem

In this thesis we have searched for the optimal cache management decisions for the known price of IoT services. However, the problem of defining the price both from the side of the providers of context and from the side of a CMP might also be an issue. For that, solving the inverse problem might be required, as well as the development of visualising instruments and dashboards for the administrators and management of the IoT infrastructure.

References

[1] C. Perera, A. Zaslavsky, P. Christen, and D. Georgakopoulos, “Context aware computing for the internet of things: A survey,” *IEEE Commun. Surv. Tutorials*, vol. 16, no. 1, pp. 414–454, 2014.

[2] X. Li, M. Eckert, J. F. Martinez, and G. Rubio, “Context aware middleware architectures: Survey and challenges,” *Sensors (Switzerland)*, vol. 15, no. 8, pp. 20570–20607, 2015.

[3] A. Tersteeg, “The Three Phases of the IoT Revolution and the Resources Developers Need to Get Started | Intel® Software,” 2017. [Online]. Available: https://software.intel.com/en-us/blogs/2017/05/02/the-three-phases-of-the-iot-revolution-and-the-resources-developer-need-to-get. [Accessed: 16-Dec-2019].

[4] ETSI, “ETSI - ETSI ISG CIM group releases first specification for context exchange in smart cities,” 2018. [Online]. Available: https://www.etsi.org/news-events/news/1300-2018-04-news-etsi-isg-cim-group-releases-first-specification-for-context-exchange-in-smart-cities. [Accessed: 26-Sep-2018].

[5] FIWARE, “FIWARE Platform,” 2019. [Online]. Available: https://www.fiware.org/. [Accessed: 05-Dec-2017].

[6] A. K. Dey and G. D. Abowd, “Towards a Better Understanding of Context and Context-Awareness,” *Comput. Syst.*, vol. 40, no. 3, pp. 304–307, 1999.

[7] S. Kubler *et al.*, “IoT Platforms initiative,” in *Digitising the Industry Internet of Things Connecting the Physical, Digital and Virtual Worlds*, 2016.

[8] A. Hassani, A. Medvedev, A. Zaslavsky, P. D. Haghighi, P. P. Jayaraman, and S. Ling, “Efficient Execution of Complex Context Queries to Enable Near Real-Time Smart IoT Applications,” *Sensors 2019, Vol. 19, Page 5457*, vol. 19, no. 24, p. 5457, Dec. 2019.

[9] A. Medvedev *et al.*, “Situation Monitoring in Context-as-a-Service IoT Platfor,” in *Global IoT Summit 2018*, 2018.

[10] A. Hassani, P. Delir Haghighi, P. P. Jayaraman, A. Zaslavsky, S. Ling, and A. Medvedev, “CDQL: A Generic Context Representation and Querying Approach for Internet of Things Applications,” in *Proceedings of the 14th International Conference on Advances in Mobile Computing and Multi Media - MoMM ’16*, 2016, pp. 79–88.

[11] F. Paganelli and D. Giuli, “An evaluation of context-aware infomobility systems,” in *Context-Aware Mobile and Ubiquitous Computing for Enhanced Usability: Adaptive Technologies and Applications*, IGI Global, 2009, pp. 338–361.

[12] P. Sotres, J. Lanza, L. Sánchez, J. R. Santana, C. López, and L. Muñoz, “Breaking vendors and city locks through a semantic-enabled global interoperable internet-of-things system: A smart parking case,” *Sensors (Switzerland)*, 2019.

[13] LoRaWAN, “About LoRaWAN,” *What is the LoRaWAN™ Specification?*, 2019. [Online]. Available: https://lora-alliance.org/about-lorawan. [Accessed: 16-Dec-2019].

[14] J. B. Kenney, “Dedicated short-range communications (DSRC) standards in the United States,” *Proc. IEEE*, 2011.

[15] J. Navarro-Ortiz, S. Sendra, P. Ameigeiras, and J. M. Lopez-Soler, “Integration of LoRaWAN

and 4G/5G for the Industrial Internet of Things," *IEEE Commun. Mag.*, 2018.

[16] A. Medvedev, A. Hassani, A. Zaslavsky, P. D. Haghighi, S. Ling, and P. P. Jayaraman, "Benchmarking IoT context management platforms: High-level queries matter," in *Global IoT Summit, GIoTS 2019 - Proceedings*, 2019, pp. 1–6.

[17] G. Schiele, M. Handte, and C. Becker, "Pervasive Computing Middleware," in *Handbook of Ambient Intelligence and Smart Environments*, 2010.

[18] M. Wagner, R. Reichle, and K. Geihs, "Context as a service - Requirements, design and middleware support," in *2011 IEEE International Conference on Pervasive Computing and Communications Workshops (PERCOM Workshops)*, 2011, pp. 220–225.

[19] J. Hong and J. Landay, "An Infrastructure Approach to Context-Aware Computing," *Human-Computer Interact.*, vol. 16, no. 2, pp. 287–303, Dec. 2001.

[20] A. K. Dey, "Understanding and using context," *Pers. Ubiquitous Comput.*, vol. 5, no. 1, pp. 4–7, Feb. 2001.

[21] M. Bazire and P. Brézillon, "Understanding Context Before Using It," Springer Berlin Heidelberg, 2005, pp. 29–40.

[22] P. Brézillon and A. J. Gonzalez, "Context in Computing," *Igarss 2014*, no. 1, pp. 1–571, 2014.

[23] P. Dourish, "What we talk about when we talk about context," *Pers. Ubiquitous Comput.*, vol. 8, no. 1, pp. 19–30, Feb. 2004.

[24] K. Henricksen and J. Indulska, "A software engineering framework for context-aware pervasive computing," in *Second IEEE Annual Conference on Pervasive Computing and Communications, 2004. Proceedings of the*, 2004, pp. 77–86.

[25] M. Knappmeyer, S. L. Kiani, E. S. Reetz, N. Baker, and R. Tonjes, "Survey of Context Provisioning Middleware," *IEEE Commun. Surv. Tutorials*, vol. 15, no. 3, pp. 1492–1519, 2013.

[26] M. Baldauf, S. Dustdar, and F. Rosenberg, "A survey on context-aware systems," *Int. J. Ad Hoc Ubiquitous Comput.*, vol. 2, no. 4, p. 263, 2007.

[27] M. Knappmeyer, S. L. Kiani, C. Frà, B. Moltchanov, and N. Baker, "ContextML: A light-weight context representation and context management schema," *ISWPC 2010 - IEEE 5th Int. Symp. Wirel. Pervasive Comput. 2010*, no. April 2016, pp. 367–372, 2010.

[28] M. Botts, "Sensor Model Language (SensorML) v2.0," *OGC Implementation Specification*, 2013. [Online]. Available: http://www.opengeospatial.org/standards/sensorml. [Accessed: 31-May-2016].

[29] K. Aberer, M. Hauswirth, and A. Salehi, "The Global Sensor networks middleware for effcient and fexible deployment and interconnection of sensor networks," *Report, Ec. Polytech. Fed. Lausanne*, 2006.

[30] K. Aberer, M. Hauswirth, and A. Salehi, "A Middleware For Fast and Flexible Sensor Network Deployment," *Proc. 32nd Int. Conf. Very large data bases*, 2006.

[31] K. Aberer, M. Hauswirth, and A. Salehi, "Infrastructure for data processing in large-scale interconnected sensor networks," in *Proceedings - IEEE International Conference on Mobile Data Management*, 2007.

[32] J. Soldatos *et al.*, "OpenIoT: Open Source Internet-of-Things in the Cloud," in *Interoperability and Open-Source Solutions for the Internet of Things*, vol. 9001, I. Podnar Žarko, K. Pripužić,

and M. Serrano, Eds. Cham: Springer International Publishing, 2015, pp. 13–25.

[33] Mongodb, “MongoDB,” *2016*, 2016. [Online]. Available: https://www.mongodb.com/. [Accessed: 29-May-2016].

[34] D. C. Faye, O. Curé, and G. Blin, “A survey of RDF storage approaches,” *Rev. Africaine la Rech. en Inform. Math{é}matiques Appliqu{é}es*, vol. 15, no. 1, p. 25, 2012.

[35] Paul Andlinger, “Graph DBMS increased their popularity by 500% within the last 2 years,” *3 March 2015*, 2015. [Online]. Available: http://db-engines.com/en/blog_post/43. [Accessed: 31-May-2016].

[36] A. Boytsov and A. Zaslavsky, “ECSTRA – Distributed Context Reasoning Framework for Pervasive Computing Systems,” Springer Berlin Heidelberg, 2011, pp. 1–13.

[37] A. Padovitz, S. W. Loke, and A. Zaslavsky, “The ECORA framework: A hybrid architecture for context-oriented pervasive computing,” *Pervasive Mob. Comput.*, vol. 4, no. 2, pp. 182–215, 2008.

[38] Hibernate.org, “Hybernate,” 2016. [Online]. Available: http://hibernate.org/. [Accessed: 28-May-2016].

[39] C. Ireland, D. Bowers, M. Newton, and K. Waugh, “A classification of object-relational impedance mismatch,” *Proc. - 2009 1st Int. Conf. Adv. Databases, Knowledge, Data Appl. DBKDA 2009*, pp. 36–43, 2009.

[40] A. Medvedev, A. Zaslavsky, M. Indrawan-Santiago, P. Delir Haghighi, and A. Hassani, “Storing and indexing IoT context for smart city applications,” in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 2016, vol. 9870 LNCS, pp. 115–128.

[41] A. Hassani *et al.*, “Context-as-a-Service Platform: Exchange and Share Context in an IoT Ecosystem,” in *2018 IEEE International Conference on Pervasive Computing and Communications Workshops (PerCom Workshops)*, 2018, pp. 385–390.

[42] M. Antunes, D. Gomes, and R. Aguiar, “Semantic-based publish/subscribe for M2M,” *Proc. - 2014 Int. Conf. Cyber-Enabled Distrib. Comput. Knowl. Discov. CyberC 2014*, no. October, pp. 256–263, 2014.

[43] W3C-SSN, “Semantic Sensor Network Ontology.” [Online]. Available: https://www.w3.org/2005/Incubator/ssn/ssnx/ssn. [Accessed: 01-Dec-2017].

[44] Schema.org, “Schema.org main page.” [Online]. Available: http://schema.org/. [Accessed: 09-Dec-2019].

[45] Schema.org, “Schema.org for IoT.” [Online]. Available: http://schema.org/.

[46] Mobivoc, “Open Mobility Vocabulary — MobiVoc.” [Online]. Available: https://www.mobivoc.org/en/index.html. [Accessed: 19-Dec-2017].

[47] I. Sosunova, A. Zaslavsky, T. Anagnostopoulos, P. Fedchenkov, O. Sadov, and A. Medvedev, “SWM-PnR: ontology-based context-driven knowledge representation for IoT-enabled waste management,” in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 2017, vol. 10531 LNCS, pp. 151–162.

[48] FIWARE, “NGSIv2 API Walkthrough - Fiware-Orion.” [Online]. Available: https://fiware-orion.readthedocs.io/en/master/user/walkthrough_apiv2/index.html. [Accessed: 19-Dec-

2017].

[49] “The Open Group.” [Online]. Available: https://www.opengroup.org/.

[50] The Open Group, “Open Messaging Interface (O-MI), an Open Group Internet of Things (IoT) Standard,” 2017. [Online]. Available: http://www.opengroup.org/iot/omi/p8.htm. [Accessed: 10-Dec-2019].

[51] The Open Group, “Open Data Format (O-DF), an Open Group Internet of Things (IoT) Standard,” 2017. [Online]. Available: http://www.opengroup.org/iot/odf/index.htm. [Accessed: 10-Dec-2019].

[52] M. Sporny, G. Kellogg, and M. Lanthaler, “JSON-LD 1.0 - A JSON-based Serialization for Linked Data,” *W3C Recommendation*, 2014. [Online]. Available: https://www.w3.org/TR/json-ld/. [Accessed: 23-Jun-2016].

[53] ETSI, “ETSI launches new group on Context Information Management for smart city interoperability,” 2017. [Online]. Available: http://www.etsi.org/news-events/news/1152-2017-01-news-etsi-launches-new-group-on-context-information-management-for-smart-city-interoperability. [Accessed: 25-Sep-2017].

[54] FIWARE, “NGSI-LD How to - Fiware-DataModels.” [Online]. Available: https://fiware-datamodels.readthedocs.io/en/latest/ngsi-ld_howto/index.html. [Accessed: 10-Dec-2019].

[55] Cantera José Manuel, “Towards schema.fiware.org,” 2016. [Online]. Available: https://www.fiware.org/2016/09/02/towards-schema-fiware-org/. [Accessed: 19-Dec-2017].

[56] “BIG IoT – Bridging the Interoperability Gap of the Internet of Things.” [Online]. Available: http://big-iot.eu/. [Accessed: 09-Dec-2019].

[57] I. P. Z. S, Soursos, “symbIoTe: Symbiosis of Smart Objects Across IoT Environments,” in *in Digitising the Industry – Internet of Things Connecting the Physical, Digital and Virtual Worlds, IERC Cluster Book 2016*, 2016, pp. 303 – 307.

[58] “Home page - FP7 - Research - Europa.” [Online]. Available: https://ec.europa.eu/research/fp7/index_en.cfm. [Accessed: 10-Dec-2019].

[59] O. Tuesday, B. Prime, and M. Tony, “Towards a theory of context,” pp. 1–27, 2010.

[60] A. Medvedev *et al.*, “Situation modelling, representation, and querying in context-as-a-service IoT platform,” in *2018 Global Internet of Things Summit, GIoTS 2018*, 2018, pp. 1–6.

[61] A. Padovitz, A. Zaslavsky, and S. W. Loke, “A unifying model for representing and reasoning about context under uncertainty,” *11th Int. Conf. Inf. Process. Manag. Uncertain. Knowledge-Based Syst.*, no. July, 2006.

[62] P. Delir Haghighi, S. Krishnaswamy, A. Zaslavsky, and M. M. Gaber, “Reasoning about Context in Uncertain Pervasive Computing Environments,” in *Smart Sensing and Context*, Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 112–125.

[63] A. Padovitz, S. W. Loke, and A. Zaslavsky, “Towards a theory of context spaces,” in *Proceedings - Second IEEE Annual Conference on Pervasive Computing and Communications, Workshops, PerCom*, 2004, pp. 38–42.

[64] A. Padovitz, S. W. Loke, A. Zaslavsky, B. Burg, and C. Bartolini, “An Approach to Data Fusion for Context Awareness,” Springer Berlin Heidelberg, 2005, pp. 353–367.

[65] Predix, “Predix developer network, Services and software,” *Predix developer network,*

Services and software, 2016. [Online]. Available: https://www.predix.io/catalog/services/. [Accessed: 30-Aug-2016].

[66] Predix, “Predix Architecture,” 2016. [Online]. Available: https://www.predix.com/sites/default/files/ge-predix-architecture-r092615.pdf. [Accessed: 09-Sep-2016].

[67] TibboSystems, “Tibbo Agregate IoT Integration platform,” 2016. [Online]. Available: http://aggregate.tibbo.com/. [Accessed: 24-Aug-2016].

[68] TibboSystems, “AggreGate Performance and Scalability Facts,” 2016. [Online]. Available: http://aggregate.tibbo.com/technology/architecture/performance.html. [Accessed: 24-Aug-2016].

[69] ThingWorx, “ThingWorx IoT Technology Platform,” 2016. [Online]. Available: https://www.thingworx.com/platforms/. [Accessed: 16-Aug-2016].

[70] “Amazon Kinesis,” 2016. [Online]. Available: https://aws.amazon.com/kinesis/. [Accessed: 24-Jun-2016].

[71] Andrew Foster, “Enhanced data storage capabilities for IBM Watson IoT Platform,” 2016. [Online]. Available: https://developer.ibm.com/iotplatform/2016/07/25/enhanced-data-storage-capabilities-for-ibm-watson-iot-platform/. [Accessed: 16-Aug-2016].

[72] B. Moltchanov and O. R. Rocha, “Generic enablers concept and two implementations for European Future Internet test-bed,” in *2014 International Conference on Computing, Management and Telecommunications (ComManTel)*, 2014, pp. 304–308.

[73] F. Ramparany, F. G. Marquez, J. Soriano, and T. Elsaleh, “Handling smart environment devices, data and services at the semantic level with the FI-WARE core platform,” in *2014 IEEE International Conference on Big Data (Big Data)*, 2014, pp. 14–20.

[74] FIWARE, “FIWARE Semantic Application Support Generic Enabler,” 2016. [Online]. Available: https://forge.fiware.org/plugins/mediawiki/wiki/fiware/index.php/Semantic_Application_Support_-_Users_and_Programmers_Guide. [Accessed: 16-Aug-2016].

[75] OpenStack, “Open Stack Swift 2.9.1.” [Online]. Available: http://docs.openstack.org/developer/swift/. [Accessed: 16-Aug-2016].

[76] FIWARE, “Object Storage GE - FIWARE Implementation,” 2016. [Online]. Available: http://catalogue.fiware.org/enablers/object-storage-ge-fiware-implementation. [Accessed: 16-Aug-2016].

[77] OpenLink, “OpenLink Software: Virtouso.” [Online]. Available: https://virtuoso.openlinksw.com/. [Accessed: 05-Jan-2020].

[78] “Apache Cassandra,” 2016. [Online]. Available: http://cassandra.apache.org/. [Accessed: 29-May-2016].

[79] Redis, “Redis in-memory data structure store.” [Online]. Available: https://redis.io/. [Accessed: 10-Dec-2019].

[80] “Apache Hadoop.” [Online]. Available: http://hadoop.apache.org/. [Accessed: 16-Aug-2016].

[81] “Apache Spark.” [Online]. Available: https://spark.apache.org/. [Accessed: 01-Dec-2017].

[82] “Apache Kafka.” [Online]. Available: https://kafka.apache.org/. [Accessed: 01-Dec-2017].

[83] Pivotal Software Inc., “Messaging that just works -- RabbitMQ,” 2019. [Online]. Available:

https://www.rabbitmq.com/. [Accessed: 10-Dec-2019].

[84] ZeroMQ, “ZeroMQ Messaging Queue,” 2019. [Online]. Available: https://zeromq.org/. [Accessed: 10-Dec-2019].

[85] Elastic.co, “Elasticsearch,” 2016. [Online]. Available: https://www.elastic.co/. [Accessed: 29-May-2016].

[86] Sphinx, “Sphinx - Open Source Search Engine.” [Online]. Available: http://sphinxsearch.com/. [Accessed: 10-Dec-2019].

[87] WSO2, “Complex Event Processor | WSO2 Inc,” *WSO2*. [Online]. Available: https://wso2.com/products/complex-event-processor/. [Accessed: 10-Dec-2019].

[88] EsperTech, “Esper.” [Online]. Available: http://www.espertech.com/esper/. [Accessed: 01-Dec-2017].

[89] OpenTSDB, “OpenTSDB Database,” 2016. [Online]. Available: http://opentsdb.net/index.html. [Accessed: 30-May-2016].

[90] InfluxDB, “InfluxDB: Purpose-Built Open Source Time Series Database | InfluxData,” 2019. [Online]. Available: https://www.influxdata.com/. [Accessed: 10-Dec-2019].

[91] Elasticsearch, “Logstash: Collect, Parse, Transform Logs | Elastic,” *Elasticsearch*, 2019. [Online]. Available: https://www.elastic.co/pt/products/logstash. [Accessed: 10-Dec-2019].

[92] Graphite, “Graphite.” [Online]. Available: https://graphiteapp.org/. [Accessed: 10-Dec-2019].

[93] Timescale, “Time-series data simplified | Timescale.” [Online]. Available: https://www.timescale.com/. [Accessed: 10-Dec-2019].

[94] PostgreSQL, “PostgreSQL: The world’s most advanced open source database,” 2013. [Online]. Available: https://www.postgresql.org/. [Accessed: 10-Dec-2019].

[95] neo4j, “Neo4j Database.” [Online]. Available: https://neo4j.com/. [Accessed: 22-Jul-2016].

[96] Amazon, “Amazon AWS,” *Business*, vol. 2011, no. 19 August. 2011.

[97] Google LLC, “Cloud IoT Core,” *Google Cloud Platform*, 2018. [Online]. Available: https://cloud.google.com/iot-core/. [Accessed: 10-Dec-2019].

[98] A. Hassani *et al.*, “Context Definition and Query Language: Conceptual Specification, Implementation, and Evaluation,” *Sensors 2019, Vol. 19, Page 1478*, vol. 19, no. 6, p. 1478, Mar. 2019.

[99] TPC, “Transaction Processing Performance Council,” 2016. [Online]. Available: http://www.tpc.org/. [Accessed: 16-Aug-2016].

[100] TPC, “TPCx-IoT.” [Online]. Available: http://www.tpc.org/tpcx-iot/default.asp. [Accessed: 23-Jan-2019].

[101] M. A. A. da Cruz, J. J. P. C. Rodrigues, A. K. Sangaiah, J. Al-Muhtadi, and V. Korotaev, “Performance evaluation of IoT middleware,” *J. Netw. Comput. Appl.*, 2018.

[102] P. Salhofer and F. H. Joanneum, “Evaluating the FIWARE Platform: A Case-Study on Implementing Smart Application with FIWARE,” in *Proceedings of the 51st Hawaii International Conference on System Sciences*, 2018, pp. 5797–5805.

[103] C. Pereira, J. Cardoso, A. Aguiar, and R. Morla, “Benchmarking Pub/Sub IoT middleware platforms for smart services,” *J. Reliab. Intell. Environ.*, vol. 4, no. 1, pp. 25–37, 2018.

[104] A. Medvedev *et al.*, “Data ingestion and storage performance of IoT platforms: Study of OpenIoT,” in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 2017, vol. 10218 LNCS, pp. 141–157.

[105] J. W. Williams, K. S. Aggour, J. Interrante, J. McHugh, and E. Pool, “Bridging high velocity and high volume industrial big data through distributed in-memory storage & analytics,” in *Proceedings - 2014 IEEE International Conference on Big Data, IEEE Big Data 2014*, 2015, pp. 932–941.

[106] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, “Benchmarking cloud serving systems with YCSB,” in *Proceedings of the 1st ACM symposium on Cloud computing - SoCC ’10*, 2010, p. 143.

[107] Yahoo, “Yahoo! Cloud Serving Benchmark (YCSB) github page,” 2016. [Online]. Available: https://github.com/brianfrankcooper/YCSB/wiki. [Accessed: 10-Sep-2016].

[108] C. Bizer, A. Schultz, Z. Pan, and J. Heflin, “Berlin SPARQL Benchmark (BSBM) Specification - V3.1,” 2011. [Online]. Available: http://wifo5-03.informatik.uni-mannheim.de/bizer/berlinsparqlbenchmark/spec/index.html. [Accessed: 14-Sep-2016].

[109] R. Nambiar, “Benchmarking Internet of Things (CISCO),” *03/08/2015*, 2015. [Online]. Available: http://blogs.cisco.com/datacenter/industry-standards-for-benchmarking-iot. [Accessed: 12-Sep-2016].

[110] G. Malim, “Looking for a Benchmarking Framework for IoT platforms,” *16/02/2016*, 2016. [Online]. Available: http://www.iotglobalnetwork.com/iotdir/2016/02/16/looking-for-a-benchmarking-framework-for-iot-platforms-1031/. [Accessed: 13-Sep-2016].

[111] K. Vandikas and V. Tsiatsis, “Performance Evaluation of an IoT Platform,” in *2014 Eighth International Conference on Next Generation Mobile Apps, Services and Technologies*, 2014, pp. 141–146.

[112] PROBE-IT, “PROBE-IT benchmarking framework,” 2016. [Online]. Available: http://www.probe-it.eu/?page_id=1036. [Accessed: 13-Sep-2016].

[113] F. Le Gall, S. V. Chevillard, A. Gluhak, and Z. Xueli, “Benchmarking internet of things deployments in smart cities,” in *Proceedings - 27th International Conference on Advanced Information Networking and Applications Workshops, WAINA 2013*, 2013, pp. 1319–1324.

[114] “EEMBC IoT Benchmark,” 2016. [Online]. Available: http://www.eembc.org/iot/about.php. [Accessed: 13-Sep-2016].

[115] M. Arlitt, M. Marwah, G. Bellala, A. Shah, J. Healey, and B. Vandiver, “IoTAbench,” in *Proceedings of the 6th ACM/SPEC International Conference on Performance Engineering - ICPE ’15*, 2015, pp. 133–144.

[116] S. Mubeen, S. A. Asadollah, A. V. Papadopoulos, M. Ashjaei, H. Pei-Breivold, and M. Behnam, “Management of Service Level Agreements for Cloud Services in IoT: A Systematic Mapping Study,” *IEEE Access*, 2018.

[117] Amazon, “AWS_IoT_Core_Pricing_Calculator.” 2019.

[118] EVRYTHNG, “EVRYTHNG Platform SLA,” *2018*. [Online]. Available: https://evrythng.com/sla/. [Accessed: 04-Dec-2018].

[119] Micosoft, “Microsoft Azure SLA for IoT Hub,” 2018. [Online]. Available: https://azure.microsoft.com/en-us/support/legal/sla/iot-hub/v1_0/. [Accessed: 04-Dec-2018].

[120] Google LLC, “Google Cloud IoT Core Service Level Agreement,” 2018. [Online]. Available: https://cloud.google.com/iot/sla. [Accessed: 04-Dec-2018].

[121] P. P. Jayaraman, K. Mitra, S. Saguna, T. Shah, D. Georgakopoulos, and R. Ranjan, “Orchestrating Quality of Service in the Cloud of Things Ecosystem,” in *2015 IEEE International Symposium on Nanoelectronic and Information Systems*, 2015, pp. 185–190.

[122] Y. Einav, “Amazon Found Every 100ms of Latency Cost them 1% in Sales,” *Gigaspaces*, 2019. [Online]. Available: https://www.gigaspaces.com/blog/amazon-found-every-100ms-of-latency-cost-them-1-in-sales/. [Accessed: 28-Oct-2019].

[123] M. Habib ur Rehman, B. Chee Sun Liew, P. Prakash Jayaraman, B. Teh Ying Wah, and B. U. Samee Khan, “Big Data Reduction Methods: A Survey,” *Data Sci. Eng.*, pp. 1–20, Dec. 2016.

[124] H. Herodotou, F. Dong, and S. Babu, “No one (cluster) size fits all,” *Proc. 2nd ACM Symp. Cloud Comput. - SOCC ’11*, pp. 1–14, 2011.

[125] D. Didona, P. Romano, S. Peluso, and F. Quaglia, “Transactional Auto Scaler: Elastic Scaling of In-memory Transactional Data Grids,” *Proc. 9th Int. Conf. Auton. Comput.*, vol. 9, no. 2, pp. 125–134, 2012.

[126] X. Dutreilh, S. Kirgizov, O. Melekhova, J. Malenfant, N. Rivierre, and I. Truck, “Using Reinforcement Learning for Autonomic Resource Allocation in Clouds: towards a fully automated workflow,” in *7th International Conference on Autonomic and Autonomous Systems (ICAS 2011)*, 2011, pp. 67–74.

[127] M. Kim, M. Shin, and S. Park, “Take me to SSD,” in *Proceedings of the 31st Annual ACM Symposium on Applied Computing - SAC ’16*, 2016, pp. 965–971.

[128] Y. Yamato and Yoji, “Use case study of HDD-SSD hybrid storage, distributed storage and HDD storage on OpenStack,” in *Proceedings of the 19th International Database Engineering & Applications Symposium on - IDEAS ’15*, 2014, pp. 228–229.

[129] J. Do, D. Zhang, J. M. Patel, D. J. DeWitt, J. F. Naughton, and A. Halverson, “Turbocharging DBMS buffer pool using SSDs,” *Proc. 2011 Int. Conf. Manag. data - SIGMOD ’11*, p. 1113, 2011.

[130] R. V. Arumugam, Q. Xu, H. Shi, Q. Cai, and Y. Wen, “Virt Cache: Managing Virtual Disk Performance Variation in Distributed File Systems for the Cloud,” in *2014 IEEE 6th International Conference on Cloud Computing Technology and Science*, 2014, pp. 210–217.

[131] K. Lee, Y.-J. Lee, H. Choi, Y. D. Chung, and B. Moon, “Parallel Data Processing with MapReduce: A Survey,” *ACM SIGMOD Rec.*, vol. 40, no. 4, p. 11, Jan. 2011.

[132] J. P. and P. R. and P. R. and L. Rodrigues, “AUTOPLACER: Scalable Self-Tuning Data Placement in Distributed Key-value Stores,” *Icac*, pp. 119–131, 2013.

[133] Ming Li, D. Ganesan, and P. Shenoy, “PRESTO: Feedback-Driven Data Management in Sensor Networks,” *IEEE/ACM Trans. Netw.*, vol. 17, no. 4, pp. 1256–1269, Aug. 2009.

[134] A. M. Hendawi and M. F. Mokbel, “Panda : A Predictive Spatio-Temporal Query Processor,” *Int. Conf. Adv. Geogr. Inf. Syst.*, 2012.

[135] S. Chaudhuri and V. R. Narasayya, “Self-tuning database systems: a decade of progress,” *Proc. 33rd Int. Conf. Very large data bases*, 2007.

[136] B. Ciciani *et al.*, “Automated workload characterization in cloud-based transactional data grids,” in *Proceedings of the 2012 IEEE 26th International Parallel and Distributed Processing*

Symposium Workshops, IPDPSW 2012, 2012, pp. 1525–1533.

[137] G. Graefe, S. Idreos, H. Kuno, and S. Manegold, "Benchmarking adaptive indexing," in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 2011, vol. 6417 LNCS, pp. 169–184.

[138] M. Lühring, K. U. Sattler, K. Schmidt, and E. Schallehn, "Autonomous management of soft indexes," in *Proceedings - International Conference on Data Engineering*, 2007.

[139] N. Bruno and S. Chaudhuri, "An online approach to physical design tuning," in *Proceedings - International Conference on Data Engineering*, 2007.

[140] S. Brin and L. Page, "The anatomy of a large-scale hypertextual Web search engine," *Comput. Networks ISDN Syst.*, vol. 30, no. 1–7, pp. 107–117, Apr. 1998.

[141] A. Salehi *et al.*, "SensorDB: a virtual laboratory for the integration, visualization and analysis of varied biological sensor data.," *Plant Methods*, 2015.

[142] P. P. Jayaraman, A. Yavari, D. Georgakopoulos, A. Morshed, and A. Zaslavsky, "Internet of things platform for smart farming: Experiences and lessons learnt," *Sensors (Switzerland)*, 2016.

[143] L. I. Gómez, B. Kuijpers, and A. A. Vaisman, "Aggregation languages for moving object and places of interest," in *Proceedings of the 2008 ACM symposium on Applied computing - SAC '08*, 2008, p. 857.

[144] H. Zhang, G. Chen, B. C. Ooi, K.-L. Tan, and M. Zhang, "In-Memory Big Data Management and Processing: A Survey," *IEEE Trans. Knowl. Data Eng.*, vol. 27, no. 7, pp. 1920–1948, Jul. 2015.

[145] P. Di Sanzo, D. Rughetti, B. Ciciani, and F. Quaglia, "Auto-tuning of cloud-based in-memory transactional data grids via machine learning," in *Proceedings - IEEE 2nd Symposium on Network Cloud Computing and Applications, NCCA 2012*, 2012, pp. 9–16.

[146] W. Ali, S. M. Shamsuddin, and A. S. Ismail, "A survey of web caching and prefetching," *International Journal of Advances in Soft Computing and its Applications*. 2011.

[147] T. Koskela, J. Heikkonen, and K. Kaski, "Web cache optimization with nonlinear model using object features," *Comput. Networks*, 2003.

[148] P. Cao and S. Irani, "Cost-aware WWW proxy caching algorithms," *Proc. USENIX Symp. Internet Technol. Syst. USENIX Symp. Internet Technol. Syst.*, 1997.

[149] L. Cherkasova, "Improving WWW proxies performance with Greedy-Dual-Size-Frequency caching policy," 1998.

[150] S. Romano and H. ElAarag, "A neural network proxy cache replacement strategy and its implementation in the Squid proxy server," *Neural Comput. Appl.*, 2011.

[151] H. ElAarag and S. Romano, "Improvement of the neural network proxy cache replacement strategy," in *Spring Simulation Multiconference 2009 - Co-located with the 2009 SISO Spring Simulation Interoperability Workshop*, 2009.

[152] A. P. Foong, Y. H. Hu, and D. M. Heisey, "Logistic regression in an adaptive Web cache," *IEEE Internet Comput.*, 1999.

[153] W. Tian, B. Choi, and V. V. Phoha, "An adaptive web cache access predictor using neural network," in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 2002.

[154] K. Tirdad, F. Pakzad, and A. Abhari, “Cache replacement solutions by evolutionary computing technique,” in *Spring Simulation Multiconference 2009 - Co-located with the 2009 SISO Spring Simulation Interoperability Workshop*, 2009.

[155] K. Y. Wong, “Web cache replacement policies: A pragmatic approach,” *IEEE Netw.*, 2006.

[156] S. Podlipnig and L. Böszörmenyi, “A survey of Web cache replacement strategies,” *ACM Comput. Surv.*, 2003.

[157] J. Domenech, J. A. Gil, J. Sahuquillo, and A. Pont, “Using current web page structure to improve prefetching performance,” *Comput. Networks*, 2010.

[158] X. Chen and X. Zhang, “Popularity-based PPM: An effective Web prefetching technique for high accuracy and low storage,” in *Proceedings of the International Conference on Parallel Processing*, 2002, vol. 2002-January, pp. 296–304.

[159] Z. Ban, Z. Gu, and Y. Jin, “An online PPM prediction model for web prefetching,” in *International Conference on Information and Knowledge Management, Proceedings*, 2007.

[160] X. Chen and X. Zhang, “A popularity-based prediction model for web prefetching,” *Computer (Long. Beach. Calif).*, 2003.

[161] Y. Jiang and M. Wu, “Web prefetching: costs, benefits and performance,” *7Th Int. Work. Web*, 2002.

[162] A. Venkataramani, P. Yalagandula, R. Kokku, S. Sharif, and M. Dahlin, “The potential costs and benefits of long-term prefetching for content distribution,” *Comput. Commun.*, 2002.

[163] B. Wu and A. D. Kshemkalyani, “Objective-optimal algorithms for long-term web prefetching,” *IEEE Trans. Comput.*, 2006.

[164] Y. F. Huang and J. M. Hsu, “Mining web logs to improve hit ratios of prefetching and caching,” *Knowledge-Based Syst.*, 2008.

[165] N. K. Papadakis, D. Skoutas, K. Raftopoulos, and T. A. Varvarigou, “STAVIES: A system for information extraction from unknown Web data sources through automatic Web wrapper generation using clustering techniques,” *IEEE Trans. Knowl. Data Eng.*, 2005.

[166] V. Safronov and M. Parashar, “Optimizing Web servers using Page rank prefetching for clustered accesses,” in *Information Sciences*, 2003.

[167] J. Domènech, J. A. Gil, J. Sahuquillo, and A. Pont, “Web prefetching performance metrics: A survey,” *Perform. Eval.*, 2006.

[168] J. Domènech, A. Pont, J. Sahuquillo, and J. A. Gil, “A user-focused evaluation of web prefetching algorithms,” *Comput. Commun.*, 2007.

[169] M. Meddeb, A. Dhraief, A. Belghith, T. Monteil, and K. Drira, “How to Cache in ICN-Based IoT Environments?,” in *2017 IEEE/ACS 14th International Conference on Computer Systems and Applications (AICCSA)*, 2017, pp. 1117–1124.

[170] M. Meddeb, A. Dhraief, A. Belghith, T. Monteil, and K. Drira, “Cache coherence in Machine-To-Machine Information Centric Networks,” in *Proceedings - Conference on Local Computer Networks, LCN*, 2015.

[171] R. Liu, W. Wu, H. Zhu, and D. Yang, “M2M-oriented QoS categorization in cellular network,” in *7th International Conference on Wireless Communications, Networking and Mobile Computing, WiCOM 2011*, 2011.

[172] S. Makridakis and M. Hibon, “ARMA models and the Box-Jenkins methodology,” *J. Forecast.*, 1997.

[173] F. M. Al-Turjman, A. E. Al-Fagih, and H. S. Hassanein, “A value-based cache replacement approach for Information-Centric Networks,” in *Proceedings - Conference on Local Computer Networks, LCN*, 2013.

[174] D. Chiu, A. Shetty, and G. Agrawal, “Elastic cloud caches for accelerating service-oriented computations,” in *2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2010*, 2010.

[175] T. Banditwattanawong, “From web cache to cloud cache,” in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 2012.

[176] L. M. Vaquero, L. Rodero-Merino, and R. Buyya, “Dynamically scaling applications in the cloud,” *Comput. Commun. Rev.*, 2011.

[177] N. Le Scouarnec, C. Neumann, and G. Straub, “Cache policies for cloud-based systems: To keep or not to keep,” in *IEEE International Conference on Cloud Computing, CLOUD*, 2014.

[178] S. Tibken, “CES 2019: Moore’s Law is dead, says Nvidia’s CEO,” 2019. [Online]. Available: https://www.cnet.com/news/moores-law-is-dead-nvidias-ceo-jensen-huang-says-at-ces-2019/. [Accessed: 12-Dec-2019].

[179] J. Jung, A. W. Berger, and Hari Balakrishnan, “Modeling TTL-based Internet caches,” 2004.

[180] E. Cohen and H. Kaplan, “Aging through cascaded caches: Performance issues in the distribution of web content,” in *Computer Communication Review*, 2001.

[181] E. Cohen, E. Halperin, and H. Kaplan, “Performance aspects of distributed caches using TTL-based consistency,” in *Theoretical Computer Science*, 2005.

[182] D. P. Heyman and M. J. Sobel, *Stochastic models*. North-Holland, 1990.

[183] H. P. Schwefel, M. B. Hansen, and R. L. Olsen, “Adaptive caching strategies for context management systems,” in *IEEE International Symposium on Personal, Indoor and Mobile Radio Communications, PIMRC*, 2007.

[184] R. L. Olsen, H. P. Schwefel, and M. B. Hansen, “Quantitative analysis of access strategies to remote information in network services,” in *GLOBECOM - IEEE Global Telecommunications Conference*, 2006.

[185] M. Bøgsted, R. L. Olsen, and H. P. Schwefel, “Probabilistic models for access strategies to dynamic information elements,” *Perform. Eval.*, vol. 67, no. 1, pp. 43–60, Jan. 2010.

[186] A. Shawky, R. Olsen, J. Pedersen, and H. Schwefel, “Network aware dynamic context subscription management,” *Comput. Networks*, 2014.

[187] M. Malawski, K. Figiela, and J. Nabrzyski, “Cost minimization for computational applications on hybrid cloud infrastructures,” *Futur. Gener. Comput. Syst.*, 2013.

[188] M. Malawski, G. Juve, E. Deelman, and J. Nabrzyski, “Algorithms for cost-and deadline-constrained provisioning for scientific workflow ensembles in IaaS clouds,” *Futur. Gener. Comput. Syst.*, 2015.

[189] M. Malawski, K. Figiela, M. Bubak, E. Deelman, and J. Nabrzyski, “Cost optimization of execution of multi-level deadline-constrained scientific workflows on clouds,” in *Lecture*

Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), 2014.

[190] A. L. Turinsky, A. L. Turinsky, R. L. Grossman, and R. L. G. Ý, "A Framework for Finding Distributed Data Mining Strategies That are Intermediate Between Centralized Strategies and In-Place Strategies," 2000.

[191] I. Menache and M. Singh, "Online Caching with Convex Costs," in *Proceedings of the 27th ACM on Symposium on Parallelism in Algorithms and Architectures - SPAA '15*, 2015, pp. 46–54.

[192] A. Hassani, A. Medvedev, P. Delir Haghighi, S. Ling, A. Zaslavsky, and P. P. Jayaraman, "Context Definition and Query Language: Conceptual specification, implementation, and evaluation (Under review)," *Sensors J.*, 2019.

[193] R. Boyd, "Polyglot Persistence Case Study: Wanderu + Neo4j + MongoDB," 2015. [Online]. Available: http://neo4j.com/blog/polyglot-persistence-mongodb-wanderu-case-study/.

[194] T. Anagnostopoulos *et al.*, "A Stochastic Multi-agent System for IoT-enabled Waste Management in Smart Cities," *Waste Manag. Res.*, Jul. 2018.

[195] A. Medvedev *et al.*, "Situation Monitoring in Context-as-a-Service IoT Platform," in *Global IoT Summit 2018 (IEEE)*, 2018.

[196] J. Wang and Jia, "A survey of web caching schemes for the Internet," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 29, no. 5, p. 36, Oct. 1999.

[197] E. Aguiar *et al.*, "Cloud computing: State-of-the-art and research challenges," *Commun. ACM*, 2010.

[198] R. B. Uriarte, F. Tiezzi, and R. De Nicola, "SLAC: A formal service-level-agreement language for cloud computing," in *Proceedings - 2014 IEEE/ACM 7th International Conference on Utility and Cloud Computing, UCC 2014*, 2014.

[199] R. C. Larson and A. R. Odoni, *Urban operations research*. Prentice-Hall, 1981.

Appendices

Appendix A. List of abbreviations

1SLA - A data access policy where one SLA is defined

2SLA - A data access policy where two SLAs are defined

ACID - Atomicity, Consistency, Isolation and Durability

AMQP - Advanced Message Queuing Protocol

ANN - Artificial Neural Networks

API - Application Programming Interface

BHR - byte Hit Rate

bIoTope - building an IoT OPen innovation Ecosystem

CA - Context-Awareness

CAP - Consistency, Availability, Partition tolerance

CAR - Cost of Automatic Retrievals

CDQL - Context Definition and Query Language

CEP - Complex Event Processing

CIM - Context Information Management

CM - Cache Management

CMP - Context Management Platforms

CoaaS - Context-as-a-Service

CoC - Cost of Context

CP - Context Provider

CQ - Context Queries

CQE - Context Query Engine

CQL - Context Query Languages

CR - Context Repository

CR - Context Repository

CRE - Context Reasoning Engine

CSDR - Context Service Description Repository

CSMS - Context Storage Management System

CSO - Connected Smart Objects

CST - Context Spaces Theory

CTR - Cost of Triggered Retrievals

CU - context updates

DDG - double dependency graph

DDoS - distributed denial-of-service attack

DS - Data Stream (processing)

DS - Document store

DSL – Domain Specific Language

DSRC - Dedicated Short Range Communication

DSS – Decision Support System

ETSI - European Telecommunications Standards Institute

EU FP7 – European Union Seventh Framework Programme

GE – Generic Enabler

GEL - Graph Expression Language

GPS - Global Positioning System

GSN - Global Sensor Networks

GUI - graphical user interface

HDD – Hard Disk Drive

HMR - Hit rate, the Miss rate, and the ratio of Refreshes to requests

HR - Hit Rate

HVAC - Heating, Ventilation, and Air Conditioning

i.i.d - independently and identically distributed

IaaS - Infrastructure as a Service

ICN - Information-Centric Networking

IMDG - In-memory Data Grid

IoT - Internet-of-Things

IoTps – IoT performance metric

JSON - JavaScript Object Notation

JSON-LD - JavaScript Object Notation for Linked Data

KV - key-value

LFU - Least Frequently Used

LR - logistic regression

LRU - Least recently Used

LSM - Linked Sensor Middleware

LSR - Latency Saving rate

LVF - Least Value First

M2M – Machine to Machine (communication)

ML - Machine Learning

MM - Markov Model

MR - Miss Rate

NoD-NoR - Not only a Database, Not only a Redirector

NoSQL – Not Only SQL

nSLA - A data access policy where n SLAs are defined

NTR –Need to Refresh

OLTP - Online Transaction Processing

ORM - Object-Relational Mapping

OWL - Web Ontology Language

P2P - Peer-to-Peer

PaaS - Platform as a Service

Pmiss - cumulative probability of a cache miss

PR - Performance Repository

PRDR - Proactive Raw Data Retrieval

QE - CoaaS query engine

QoS - Quality of Service

RDBMS - Relational Database Management System

RDF - Resource Description Framework

RESTful - representational state transfer (service)

RFID - Radio Frequency IDentification

RPN - Reverse Polish notation

RR - Refresh ratio

RRB - Refresh Rate-Based (policy)

SaaS - Software as a Service

SCADA - supervisory control and data acquisition

SHR - Simulated Hit Rate

SLA - Service Level Agreement

SLA1 - Service Level Agreement 1 (First) (the most expensive)

SLAn - Service Level Agreement n-th

SM - Subscription Module

SMR - Simulated Miss Rate

SPARQL - SPARQL Protocol and RDF Query Language

SQEM - Storage Query Execution Manager

SQL – Structured Query Language

SR - Simulated Refresh ratio

SSD – Solid State Disk

SSN - Semantic Sensor Network

TCP/IP - Transmission Control Protocol/Internet Protocol

TOG - The Open Group

TPC - Transaction Processing Performance Council

TTL - Time-to-Live

VIN - Vehicle Identification Number

XML - Extensible Markup Language

Appendix B. Glossary

Complex Event Processing is the process of analysing continuous stream of events for finding the defined patterns.

Context [20] "is any information that can be used to characterise the situation of an entity. An entity is a person, place, or object that is considered relevant to the interaction between a user and an application, including the user and applications themselves."

Context attribute is the data, which can characterise an associated context entity.

Context consumer is any participant of the IoT ecosystem, which requires context about entities.

Context entity is an object (real or virtual) of the IoT ecosystem.

Context Management Platform (CMP) is an advanced middleware platforms which comprise functionalities of (i) IoT marketplace, (ii) gateways to multiple data sources, (iii) subscription mechanisms, (iv) features for performing aggregations, reasoning, analytical functions, and (v) advanced sensor data management.

Context provider is any participant of the IoT ecosystem, which is able to provide context about entities.

Context query represents a formulated request for contextual information.

Context request (in CDQL scope) is a part of a context query, which is formulated to retrieve contextual information about a particular entity type.

Context update is a message, containing information about the current value of one or many context attributes.

Context-as-a-Service (concept) refers to an IT business model, where a platform provides contextual information about external entities based on the pay-as-you-go model.

Cost of Automatic Retrievals (CAR) - the cost incurred by the total number of planned retrievals, which occurred during the planning period.

Cost of Triggered Retrievals (CTR) - the cost incurred by total number of retrievals caused by cache misses during the planning period.

Data stream – we use the term Event Stream interchangeably with Data Stream. However, in other domains (e.g. video streaming) the term data stream can have a different meaning.

Database Mode of CMP operation refers to the mode based on the full coverage strategy. All the queries are served from the cache.

Elastically scalable system is a system, which can rapidly increase the amount of used physical resources (e.g. computational, storage), as well as decrease the amount of these resources.

Event stream is an incoming continuous and potentially infinite sequence of messages, which contain data about changes in context attributes' values.

Freshness is the metric, which shows how stale is the cached value.

Full Coverage strategy is a caching strategy, where the prefetching is always performed in the end of the freshness period, thus queries can be always served from the cache.

Gap (cache) is the time Δt between the end of the expiry period and the moment of planned retrieval. If a request arrives during this period, it "falls within the gap," and CSMS has a cache miss.

Invoker (CoaaS platform) is a software component responsible for retrieving the information form context providers.

IoT ecosystem (IoT Phase 3) refers to the stage, when smart devices can communicate with each other spontaneously, without being locked into a silo of a company, which produced the device or the software. This stage is characterised by a massive number of horizontal connections; devices and vertical silos that form an ecosystem.

IoT silo is a vertically oriented IoT system where the dataflow is controlled by one organisation.

Least Fresh First (LFF) strategy is a caching strategy when the least fresh first items are evicted first.

Least Valuable First (LVF) strategy is a caching strategy when the items which are less valuable to the system are evicted first.

Need to Refresh (NTR) – is a metric, which shows how strong is the influence of refreshing the data item at current time on the cache performance.

Not only Database – Not only Redirector (NoD-NoR) mode of CMP operation refers to the mode, where a decision about the caching strategy and caching rate is made individually for

each context attribute. The queries can be served from the cache, as well as from the external providers.

Planned refresh period is the time between planned retrievals of a data item by CoaaS from a context provider.

Planning period is the period of time for which the decision about the caching strategy and the corresponding parameters is made.

Prefetching refers to the process of retrieving context from providers and caching it, before it was requested by the consumer. Prefetching is used as an enabler for the proactive caching strategy.

Proactive strategy is a caching strategy that is based on prefetching the data before it was requested by the consumer. A proactive strategy allows cache gaps, which makes it different from the full coverage strategy.

PULL-based query (CDQL) is a query, which is executed once at the moment of arrival. It can be seen as a SELECT query in databases.

PUSH-based query (CDQL) represents a subscription for situation monitoring. The resulting context is "pushed" back to the consumer, when the platform detects the situation by analysing incoming event streams.

Reactive strategy is a caching strategy, which is based on reusing cached data for serving queries, but never uses prefetching.

Real refresh period is the average time between retrievals.

Redirector Mode of CMP operation refers to the mode, when all the queries are served based on the data, which is retrieved from the context providers on the fly. Cache is not used.

Service Level Agreement (SLA) is a formally defined contract between the consumer and the provider of IT services. SLA define such aspects as latencies, reliability, accuracy, freshness, prices and penalties.