### SEVENTH FRAMEWORK PROGRAMME
Specific Targeted Research Project

<table>
<thead>
<tr>
<th>Call Identifier:</th>
<th>FP7–ICT–2011–7</th>
</tr>
</thead>
<tbody>
<tr>
<td>Project Number:</td>
<td>287305</td>
</tr>
<tr>
<td>Project Acronym:</td>
<td>OpenIoT</td>
</tr>
<tr>
<td>Project Title:</td>
<td>Open source blueprint for large scale self-organising cloud environments for IoT applications</td>
</tr>
</tbody>
</table>

### D4.5.2 Elastic Publish/Subscribe Processing Engine for Sensor Data Streams

<table>
<thead>
<tr>
<th>Document Id:</th>
<th>OpenIoT-D452-140805-Draft</th>
</tr>
</thead>
<tbody>
<tr>
<td>File Name:</td>
<td>OpenIoT-D452-140805-Draft.pdf</td>
</tr>
<tr>
<td>Document reference:</td>
<td>Deliverable 4.5.2</td>
</tr>
<tr>
<td>Version:</td>
<td>Draft</td>
</tr>
<tr>
<td>Editor:</td>
<td>Aleksandar Antonić, Ivana Podnar Žarko</td>
</tr>
<tr>
<td>Organisation:</td>
<td>UNIZG-FER</td>
</tr>
<tr>
<td>Date:</td>
<td>2014 / 08 / 05</td>
</tr>
<tr>
<td>Document type:</td>
<td>Deliverable</td>
</tr>
<tr>
<td>Security:</td>
<td>PU (Public)</td>
</tr>
</tbody>
</table>

Copyright © 2014 OpenIoT Consortium: NUIG-National University of Ireland Galway, Ireland; EPFL - Ecole Polytechnique Fédérale de Lausanne, Switzerland; Fraunhofer Institute IOSB, Germany; AIT - Athens Information Technology, Greece; CSIRO - Commonwealth Scientific and Industrial Research Organization, Australia; SENSAP Systems S.A., Greece; AcrossLimits, Malta. UNIZG-FER University of Zagreb, Faculty of Electrical Engineering and Computing, Croatia. Project co-funded by the European Commission within FP7 Program.

---

**PROPRIETARY RIGHTS STATEMENT**

This document contains information, which is proprietary to the OpenIoT Consortium. Neither this document nor the information contained herein shall be used, duplicated or communicated by any means to any third party, in whole or in parts, except with prior written consent of the consortium.
## DOCUMENT HISTORY

<table>
<thead>
<tr>
<th>Rev.</th>
<th>Author(s)</th>
<th>Organisation(s)</th>
<th>Date</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>V200</td>
<td>Aleksandar Antonić</td>
<td>UNIZG-FER</td>
<td>2014/01/15</td>
<td>ToC based on D4.5.1</td>
</tr>
<tr>
<td>V210</td>
<td>Aleksandar Antonić</td>
<td>UNIZG-FER</td>
<td>2014/01/15</td>
<td>D4.5.1 content with some modifications.</td>
</tr>
<tr>
<td>V220</td>
<td>Aleksandar Antonić</td>
<td>UNIZG-FER</td>
<td>2014/05/10</td>
<td>Section 4 &amp; 5 update</td>
</tr>
<tr>
<td>V230</td>
<td>Ivana Podnar Žarko</td>
<td>UNIZG-FER</td>
<td>2014/05/26</td>
<td>Minor modifications to Section 2 and Section 3. Comments to Section 3 and Section 4.</td>
</tr>
<tr>
<td>V240</td>
<td>Aleksandar Antonić</td>
<td>UNIZG-FER</td>
<td>2014/05/27</td>
<td>Minor modifications</td>
</tr>
<tr>
<td>V250</td>
<td>Kresimir Pripužić</td>
<td>UNIZG-FER</td>
<td>2014/05/29</td>
<td>Section 4 update</td>
</tr>
<tr>
<td>V260</td>
<td>Aleksandar Antonić</td>
<td>UNIZG-FER</td>
<td>2014/05/30</td>
<td>Section 6 update</td>
</tr>
<tr>
<td>V270</td>
<td>Jean-Paul Calbimonte</td>
<td>EPFL</td>
<td>2014/06/18</td>
<td>Technical review</td>
</tr>
<tr>
<td>V280</td>
<td>Maja Matijašević</td>
<td>UNIZG-FER</td>
<td>2014/07/14</td>
<td>Quality review</td>
</tr>
<tr>
<td>V290</td>
<td>Ivana Podnar Žarko</td>
<td>UNIZG-FER</td>
<td>2014/07/21</td>
<td>Circulated for Approval</td>
</tr>
<tr>
<td>V291</td>
<td>Martin Serrano</td>
<td>NUIG</td>
<td>2014/08/05</td>
<td>Approved</td>
</tr>
<tr>
<td>Draft</td>
<td>Martin Serrano</td>
<td>NUIG</td>
<td>2014/08/05</td>
<td>EC Submitted</td>
</tr>
</tbody>
</table>
# TABLE OF CONTENTS

1 INTRODUCTION .............................................................................................................. 7  
  1.1 Scope .......................................................................................................................... 7  
  1.2 UNIZ-FER contribution to the OpenIoT platform .................................................... 8  
  1.3 Audience .................................................................................................................... 9  
  1.4 Summary .................................................................................................................... 9  
  1.5 Structure ..................................................................................................................... 9  

2 SELF-ORGANIZING PUBLISH/SUBSCRIBE IN THE CLOUD ................................ 10  
  2.1 Overview ..................................................................................................................... 10  
    2.1.1 Boolean and top-k/w matching in publish/subscribe systems ......................... 11  
    2.2 Publish/subscribe model ....................................................................................... 13  
    2.3 Self-organizing characteristics: scalability and elasticity ................................... 16  

3 PUBLISH/SUBSCRIBE SYSTEM ARCHITECTURE ........................................... 19  
  3.1 Sequence diagrams ................................................................................................... 21  
  3.2 Interaction with the OpenIoT platform .................................................................... 24  

4 CLOUD-BASED PUBLISH/SUBSCRIBE PROCESSING ENGINE ................... 27  
  4.1 Cloud-based architecture ......................................................................................... 27  
    4.1.1 Flat architecture implementation ..................................................................... 28  
    4.1.2 Hierarchical architecture implementation ...................................................... 29  
    4.1.3 Publisher representation within the CPSP engine .......................................... 29  
    4.1.4 Subscriber representation within the CPSP engine ........................................ 30  
  4.2 Load balancing within the CPSP engine .................................................................. 32  
    4.2.1 Splitting of matchers ....................................................................................... 32  
    4.2.2 Merging of matchers ....................................................................................... 34  
  4.3 Subscription language .............................................................................................. 35  
  4.4 Matcher implementation ........................................................................................... 37  
    4.4.1 Subscription forest ............................................................................................ 37  
    4.4.2 Top-k/w processing ......................................................................................... 38  

5 API SPECIFICATION AND EXAMPLES ................................................................. 39  
  5.1 Cloud Broker ............................................................................................................ 39  
    5.1.1 Main Released Functionalities & Services ....................................................... 39  
    5.1.2 Download, Deploy & Run ................................................................................. 40  
      5.1.2.1 Developer ................................................................................................. 40  
      5.1.2.1.1 System requirements ......................................................................... 40  
      5.1.2.1.2 Download ........................................................................................... 40  
      5.1.2.1.3 Deploy from the source code ............................................................... 40
5.2 Publisher .................................................................................. 42
  5.2.1 Main Released Functionalities & Services .................................. 42
  5.2.2 Download, Deploy & Run ......................................................... 43
    5.2.2.1 Developer .................................................................. 43
      5.2.2.1.1 System requirements ........................................... 43
      5.2.2.1.2 Download .......................................................... 43
      5.2.2.1.3 Deploy from the source code .................................. 43
      5.2.2.1.4 Run in Eclipse ...................................................... 44
        5.2.2.1.4.1 Deploying Publisher .......................................... 44
      5.2.2.2 User .................................................................... 44
        5.2.2.2.1 System requirements ........................................... 44
        5.2.2.2.2 Deployment/Undeployment ................................. 44
  5.3 Subscriber ............................................................................... 44
    5.3.1 Main Released Functionalities & Services .............................. 45
    5.3.2 Download, Deploy & Run ..................................................... 46
      5.3.2.1 Developer ................................................................ 46
        5.3.2.1.1 System requirements ........................................... 46
        5.3.2.1.2 Download .......................................................... 46
        5.3.2.1.3 Deploy from the source code ................................. 46
        5.3.2.1.4 Run in Eclipse ...................................................... 47
          5.3.2.1.4.1 Deploying Subscriber ......................................... 47
        5.3.2.2 User ................................................................... 47
          5.3.2.2.1 System requirements ........................................... 47
          5.3.2.2.2 Deployment/Undeployment ................................. 47
  5.4 Source code examples .............................................................. 48
    5.4.1 CloudBroker .................................................................. 48
    5.4.2 Subscriber and subscribing process .................................... 48
    5.4.3 Publisher and publishing process ........................................ 50

6 EXPERIMENTAL RESULTS ................................................................ 51

7 RELATED DATA STREAM PROCESSING PLATFORMS .......................... 56

8 CONCLUSIONS ........................................................................... 59

9 REFERENCES ................................................................................ 60
LIST OF FIGURES

FIGURE 1 MATCHING IN BOOLEAN PUBLISH/SUBSCRIBE SYSTEMS ........................................ 12
FIGURE 2 MATCHING IN TOP-K/W PUBLISH/SUBSCRIBE SYSTEMS .................................... 13
FIGURE 3 FLAT CLOUD BROKER ARCHITECTURE .................................................................. 17
FIGURE 4 HIERARCHICAL CLOUD BROKER ARCHITECTURE ............................................... 17
FIGURE 5 ARCHITECTURE OF THE PUBLISH/SUBSCRIBE SYSTEM ................................... 20
FIGURE 6 DELIVERY OF A NEW PUBLICATION .................................................................. 22
FIGURE 7 ANNOUNCING A NEW PUBLISHER ..................................................................... 22
FIGURE 8 REVOKING AN ANNOUNCEMENT ....................................................................... 23
FIGURE 9 ACTIVATING A NEW SUBSCRIPTION .................................................................. 23
FIGURE 10 CANCELLING A SUBSCRIPTION ....................................................................... 24
FIGURE 11 FORWARDING SENSOR READINGS TO THE OPENIoT PLATFORM ..................... 25
FIGURE 12 ACTIVATING A NEW SUBSCRIPTION FROM THE OPENIoT PLATFORM ............ 26
FIGURE 13 SYSTEM COMPONENTS AND THEIR COMMUNICATION PATHWAYS .................... 28
FIGURE 14 COMMUNICATION OF A PUBLISHER WITH THE CPSP ENGINE ..................... 30
FIGURE 15 COMMUNICATION OF A SUBSCRIBER WITH THE CPSP ENGINE (THE MESSAGE RECEIVER PART) ................................................................. 31
FIGURE 16 COMMUNICATION OF A SUBSCRIBER WITH THE CPSP ENGINE (THE DELIVERY SERVICE PART) ................................................................. 31
FIGURE 17 PROCESSING OF A RECEIVED MESSAGE FOLLOWED BY A MATCHER SPLITTING TRIGGER ................................................................. 33
FIGURE 18 PROCESSING OF RECEIVED MESSAGE FOLLOWED BY A MATCHER MERGING TRIGGER ................................................................. 34
FIGURE 19 AN EXAMPLE OF A BOOLEAN SUBSCRIPTION FOREST ................................... 37
FIGURE 20. ADDING AN INCOMING PUBLICATION INTO A STRICT K-SKYBAND ................. 38
FIGURE 21 DATA PROPAGATION DELAY WHEN INCREASING THE NUMBER OF SUBSCRIPTIONS UNDER CONSTANT PUBLICATION RATE .................................................. 52
FIGURE 22 DATA PROPAGATION DELAY WHEN PROCESSING INCREASING PUBLICATION DATASETS ................................................................. 53
FIGURE 23 THE DATA PROPAGATION DELAY DEPENDING ON THE INCOMING PUBLICATION FREQUENCY WITH DYNAMIC CUPUS CONFIGURATION ........................................ 54
FIGURE 24 CPSP ENGINE CONFIGURATION DURING AN EXPERIMENT WITH VARYING PUBLISHING LOAD .... 55
LIST OF TABLES

TABLE 1 SUPPORTED OPERATORS FOR BOOLEAN SUBSCRIPTIONS .................................................. 35
TABLE 2 CLOUD BROKER’S PUBLIC METHODS AND CONSTRUCTOR ........................................... 39
TABLE 3 IMPLEMENTED CLOUD BROKER API DEFINITION ...................................................... 39
TABLE 4 PUBLISHER PUBLIC METHODS AND CONSTRUCTOR .................................................. 42
TABLE 5 IMPLEMENTED PUBLISHER API DEFINITION ............................................................... 42
TABLE 6 SUBSCRIBER PUBLIC METHODS AND CONSTRUCTOR .................................................. 45
TABLE 7 IMPLEMENTED SUBSCRIBER API DEFINITION .............................................................. 45
TABLE 8 CLOUD BROKER SOURCE CODE SNIPPET ................................................................. 48
TABLE 9 SUBSCRIBING PROCESS SOURCE CODE SNIPPET ....................................................... 49
TABLE 10 PUBLISHING PROCESS SOURCE CODE SNIPPET ....................................................... 50
TERMS AND ACRONYMS

CO  carbon monoxide
CPSP  Cloud-based Publish/Subscribe Processing
CUPUS  CloUd-based PUblish/Subscribe for the Internet of Things
GCM  Google Cloud Messaging
GSN  Global Sensor Networks
ICO  Internet-Connected Object
IoT  Internet of Things
LSM  Linked Sensor Middleware
MIO  Mobile Internet-connected Object
MSC  Mobile Sensor Controller
NO2  nitrogen dioxide
TCP  Transmission Control Protocol
_top-k/w  Continuous top-k processing over sliding windows
UDP  User Datagram Protocol
VOC  Volatile Organic Compound
1 INTRODUCTION

1.1 Scope

The main goal of the OpenIoT project is to provide an open source platform for building and deploying on-demand utility-based IoT applications, i.e., applications that promote and realise the convergence of cloud-computing with the Internet-of-Things. The core of this infrastructure comprises a middleware framework, which facilitates service providers to deploy and monitor IoT applications in the cloud, while also enabling service integrators and end-users to access and orchestrate internet-connected objects (ICOs) and seamlessly acquire their data.

During the first year of the OpenIoT project, additional requirements regarding mobility and quality of service have been identified to enable the OpenIoT platform to adapt to the continuous evolution of the design and implementation processes of IoT data systems and particularly to adapt the OpenIoT Architecture to mobile sensors. Additional OpenIoT functionality related to mobile sensor data collection and quality of service provisioning implies the implementation and development of publish/subscribe mechanisms for efficient continuous processing and filtering of sensor data streams in cloud environments. This deliverable focuses on designing and implementing those additional features in the OpenIoT architecture.

Since the streaming rate of Internet of Things (IoT) data streams falls in the Big Data domain, in this deliverable we propose solutions for efficient continuous processing of sensor data streams based on the publish/subscribe principles to identify relevant data objects and deliver them in near real-time to largely distributed data consumers, e.g., user mobile phones. The algorithms are tailored to the cloud environment and optimized for processing of streams characterized by high streaming rates.

OpenIoT deliverable D4.5.2 presents a publish/subscribe cloud implementation tailored for the Internet of Things entitled CUPUS: CloUd-based PUblish/Subscripte for the Internet of Things which is a new OpenIoT platform component/module. It includes the design of the open source prototype available at the Github infrastructure of the OpenIoT project (https://github.com/OpenIotOrg/openiot).

The current CUPUS prototype supports content-based publish/subscribe, i.e., stateless Boolean subscriptions, and top-k subscriptions over sliding windows (top-k/w subscriptions) that are stateful. CUPUS is designed to enable pre-filtering of sensor data streams close to data sources, e.g., on mobile devices, so that only data of interest and value to various subscribers is pushed into the publish/subscribe cloud. Note that the filtering process is not guided locally on mobile devices, but from the cloud, based on global data requirements. Moreover, the CUPUS cloud distributes push-based notifications from the cloud to largely distributed destinations, e.g., mobile devices, in near real-time. More information on data acquisition from mobile ICOs and data delivery to mobile devices is available in deliverable D3.4.2: Publish/subscribe middleware for mobile internet-connected objects.
The CUPUS architecture is designed to be elastic, so that it can dynamically allocate more resources within the cloud or release underused resources in accordance with the processing load. It supports a dynamic flat architecture with concurrent processes that share the processing load, as instructed by a central coordinator.

CUPUS release is fully integrated with the rest of the OpenIoT platform by use of the QoS Manager component which is presented in deliverable D4.6. Sensor data acquired by mobile devices is published and annotated by use of X-GSN virtual sensors, and is furthermore stored in the OpenIoT RDF store. Moreover, OpenIoT components can search for mobile sensors within a certain geographical area and define requests to receive the data from mobile sensors located in geographical areas of interest by issuing requests to the QoS Manager Web service interface.

1.2 UNIZ-FER contribution to the OpenIoT platform

CUPUS is a novel component of the OpenIoT platform, which primarily focuses on cloud-based publish/subscribe processing as well as near real-time and push-based delivery of sensor-generated information to end-user devices, typically mobile devices. Since mobile users can frequently change their context, it is vital that the time period from data acquisition until data delivery is appropriate for specific usage scenarios when users are, for example, walking, running or cycling. Therefore, the end-to-end data lifecycle in mobile environments needs to be carefully optimized: We need to minimize the time needed for data acquisition, processing (matching of publications existing subscriptions), and data delivery. In this deliverable we focus on efficient cloud-based data processing, while Deliverable 3.4.2 focuses on data acquisition and delivery in a mobile environment.

In the second version of the OpenIoT platform release, CUPUS is fully integrated with the rest of the OpenIoT platform. It can be regarded as a special "high-speed" information bus for mobile environments which integrates mobile ICOs with the rest of the OpenIoT platform. In other words, CUPUS enables propagation of OpenIoT requests to mobile ICOs which triggers data acquisition on mobile devices, and stores the generated data within the OpenIoT cloud database rather than using its own permanent storage. Moreover, other OpenIoT components that require content-based and top-k/w publish/subscribe services can use the CUPUS infrastructure for efficient publish/subscribe processing.
1.3 Audience

The target audience for this deliverable includes:

- **OpenIoT project members**, in particular, those to be engaged in the deployment and/or use of the CUPUS middleware. For these members, the deliverable could serve as a valuable guide for the installation, deployment, integration and use of CUPUS cloud that comprises the OpenIoT open source software.

- **The IoT open source community**, which should view the present deliverable as a vital source of information for understanding publish/subscribe middleware built within the OpenIoT project. Note also that members of the open source community might be also willing to contribute to the OpenIoT project. For these members, the deliverable can serve as a basis for understanding the technical implementation of the CUPUS components.

- **IoT researchers at large**, who need a practical guide on the main elements/components for cloud-based publish/subscribe services that support large-scale IoT solutions in mobile environments with near-time processing delay.

All the above groups could benefit from reading the report, but also from using the released prototype implementation.

1.4 Summary

In this deliverable we present **CUPUS: Cloud-based Publish/Subscribe for the Internet of Things that** supports publish/subscribe processing of large data volumes within the cloud. This deliverable also presents a prototype implementation and accompanying report of an elastic and self-organizing publish/subscribe engine for continuous processing of sensor data streams.

1.5 Structure

In this deliverable we focus on publish/subscribe processing of large data volumes within the cloud. We present the conceptual publish/subscribe model and the underlying data model in Section 2. Section 3 presents the details our cloud-based architecture and explains the interaction of the publish/subscribe engine with the environment. In Section 4 we provide the implementation details of the CUPUS cloud, while Section 5 includes the API specification with code examples. We give initial experimental results which investigate CUPUS performance in Section 6. Section 7 provides an overview of related stream processing engines and Section 8 concludes the report.
2 SELF-ORGANIZING PUBLISH/SUBSCRIBE IN THE CLOUD

2.1 Overview

Since the Internet of Things continuously generates data streams which fall in the Big Data domain, it is vital to efficiently process sensor data streams and deliver data items of interest to users in near real-time. For example, mobile IoT applications running on user mobile phones and tablets are greatly affected by frequent changes of user context which results in varying user needs for sensor-based information. Therefore all notifications delivered to mobile devices need to be relevant, both in time and space. In view of an urban crowdsourcing application, in which both pedestrians and cyclists use an environmental or traffic monitoring application, which provides them with notifications affecting their movement throughout the city, the time period from the time of the sensor data acquisition until the time of the notification delivery to mobile devices needs to be short (in the range of a few seconds). This is indeed not a trivial task, since during that period of time the acquired data needs to be enriched, filtered and processed so that in the end of this process only the information of interest is served to the end users.

Publish/subscribe middleware offers the mechanisms to deal with some of the previously identified challenges: It enables selective real-time acquisition and filtering of sensor data on mobile devices, efficient continuous processing of large data volumes within the cloud, and near real-time delivery of notifications to mobile devices. Publish/subscribe is a well-established continuous processing and communication infrastructure for efficient and flexible data filtering, as well as push-based data dissemination from data sources, called publishers, to data destinations, called subscribers. Data filtering and distribution are flexible because it takes into account user preferences expressed as subscriptions (or continuous queries) [Muhl2006]. It is often used for efficient messaging in real-time because it has the ability to process large amounts of data due to simple subscription languages and efficient matching algorithms.

In a typical publish/subscribe system, the publishers publish content (i.e. data objects), which they want to distribute among the subscribers in the form of messages called publications. Subscribers subscribe to the content in which they are interested by issuing messages called subscriptions. A publish/subscribe service is responsible for delivering publications in real-time, i.e. immediately after their publication, to those subscribers whose subscriptions match selected publications. Publications are delivered in messages called notifications. A publication matches a subscription when it satisfies all constraints defined by the subscription. A common processing entity forming a publish/subscribe systems is a broker, an intermediary to which publishers post publications and subscribers register subscriptions. The main broker task is to perform publication matching, i.e. comparison of subscriptions to publications to determine whether a publication satisfies constraints defined by a subscription. Moreover, brokers perform the task of notification delivery to subscribers, e.g., processes running on mobile devices, and maintain a list of publications for temporarily disconnected subscribers.
Brokers are the main building blocks of a publish/subscribe system. As they can be replicated and distributed to divide the processing load associated with the matching process, they ensure the overall system scalability and elasticity. In Section 2.3 we present potential cloud-based publish/subscribe architectures to achieve elastic real-time computation under varying processing load.

Publish/subscribe middleware is adequate for mobile IoT environments where Mobile Internet-connected Objects (MIOs) generate sensor data which are relayed through mobile devices into a data cloud. In addition, mobile devices can receive and serve cloud-processed data of interest to users in a timely fashion, while the communication between publishers and subscribers is persistent and asynchronous.

Since matching has already been identified as the main broker task, hereafter we need to analyse subscription models supported by our publish/subscribe implementation. In addition to the state-of-the-art Boolean subscription which is defined as a conjunction of constraints on attribute values, we also support top-k subscriptions over sliding windows (or top-k/w subscriptions). Top-k/w publish/subscribe over sliding windows identifies k best-ranked objects with respect to a given scoring function over a sliding window of size w, and is based on the publish/subscribe communication paradigm augmented by algorithms coming from the field of data stream processing [Pripužić2014a]. For example, using an environmental monitoring application with top-k/w subscriptions, environmental scientists can identify and monitor up to k sites with the largest pollution readings over the course of a single day, or identify a predefined number of sensors closest to a particular location measuring the largest pollution levels over time (e.g., top-10 readings per hour). The main difference of such processing, compared to one-time top-k query over past data stream objects, is that with top-k/w processing an object is identified as a top-k object over time, i.e., it is delivered and reported as a top-k object as soon as it becomes the top-k object in a sliding window.

### 2.1.1 Boolean and top-k/w matching in publish/subscribe systems

In current publish/subscribe systems, subscription is mainly defined as a stateless Boolean function [Mühl2002]. A decision whether to deliver a publication to a subscriber is made based on the result of the matching process, by comparing a publication to subscriber's subscription, as shown in Figure 1. The matching process depends exclusively on publication and subscription content, and does not take into account any additional information present in the system. This approach has the following drawbacks:

1. a subscriber may be either overloaded with publications or receive too few publications over time,
2. a subscriber may be overloaded with publications,
3. it is impossible to compare different publications matching a given subscription, as ranking functions are not defined, and
4. partial matching between subscriptions and publications is not supported.
As the publication content is generally unknown in advance, it is impossible to predict the number of future publications matching a subscription. If a subscription is too general, a subscriber may receive too many publications over time. On the other hand, in case of an over specified subscription, the subscriber may receive too few publications (or none, in the worst case). Thus, a subscriber has to specify a "perfect" subscription to receive an optimal number of matching publications. It is a sort of guessing, where even a slight change in subscription may result in a drastically different number of matching publications. In general, a user perceives the entire system through both the quantity and quality of received publications. Therefore, a large quantity of received publications may end up being considered spam, while a system that delivers too few publications might be recognized as non-working. The number of received publications is crucial for the acceptance of an actual system by users even more if, for example, subscribers need to pay for each delivered publication matching subscriber information interest.

The top-k/w publish/subscribe model enables a subscriber to control the number of publications it receives per subscription within a predefined time period. Each subscription defines 1) a time-independent scoring function, and 2) the parameters $k \in \mathbb{N}$ and $w \in \mathbb{R}^+$. Unlike the Boolean publish/subscribe model, it is time-dependent because at each point in time $t$, the parameter $k$ limits the number of matching publications restricting it to the $k$ best matched, or “top-k” publications published between points in time $t-w$ and $t$. This interval is called the time-based window of size $w$ and it is constantly sliding in time. Note that the size of subscription window may also be defined as the number of most recent publications (count-based window) [Mouratidis07].

The number of received publications in the top-k/w publish/subscribe model is independent of the publishing frequency, and depends on parameters $k$ and $w$. This means that the matching process which compares a newly published publication to a subscription depends both on the score of the new publication and scores of previous publications calculated using the same scoring function. In other words, each publication is competing with other publications from the sliding window for a position among the top-k publications as depicted in Figure 2. Obviously, the quality of received publications depends on calculated scores, and is statistically proportional to the number of published publications.
The implementation of top-k/w processing is not trivial for big data streams because a publication within a window of a top-k/w subscription becomes a top-k/w publication when there are less than \( k \) higher ranked publications within the window. This can happen only if a new publication enters the system when there are fewer than \( k \) higher ranked publications within that subscription window, or, if at a later point in time, when one of top-k publications is dropped from the subscription window, a candidate publication has the highest rank among all other candidate publications. Therefore, the set of candidate publications needs to be maintained in main memory for top-k/w processing, while this set changes with each new publication.

### 2.2 Publish/subscribe model

In this section we give formal specifications of two publish/subscribe systems used in our prototype: a) the Boolean publish/subscribe system, and b) the top-k/w publish/subscribe system, by using the set theory.

The **Boolean publish/subscribe system** is based on the **Boolean matching model**, which is a generalization of the three most common matching models found in literature: topic-based, content-based, and type-based models [Eugster2003]. The matching process uses a given Boolean matching function defined by each subscription.

The **top-k/w publish/subscribe system** relies on a novel matching model, named the **top-k/w matching model** [Pripužić2014b]. The top-k/w matching model is stateful and time-dependent, because at each point in time \( t \), the parameter \( k \) limits the number of matching publications restricting it to the \( k \) “best”, or “top-k” publications published within a subscription window. Each publication is ranked according to a time-independent scoring function, and we assume **time-based windows** in our model.

#### 2.2.1 Boolean Publish/Subscribe System

Let \( PS=(B,P,S,M,N) \) be a quintuple, where \( B \) is a finite set of brokers, \( P \) is a finite set of publishers, \( S \) is a finite set of subscribers, \( M \) is a finite set of publications, and \( N \) is a finite set of Boolean subscriptions in a Boolean publish/subscribe system. \( PS \) gives the **structural view** of a Boolean publish/subscribe system and determines the boundaries of its state space. A publisher \( p \in P \) may publish or cancel publications from \( M \), while a subscriber \( s \in S \) may activate or cancel subscriptions from \( N \). A broker
$b \in B$ performs matching between active subscriptions and publications defined by a subset of publishers from $P$ and subset of subscribers from $S$ that are currently connected to $b$.

**Publication.** A publication $m \in M$ is defined as some content (e.g., a sensor reading, or, a tweet), which is published by publisher $p \in P$ at a point of time $t$, while a broker $b \in B$ responsible for processing $m$ receives it at a point in time $t^b$. We say that the time of appearance $t^b$ denotes a point in time when publication $m$ appears in the system, i.e., when it is received by broker $b$. Broker $b$ is responsible for processing publication $m$ because publisher $p$ is connected to broker $b$ at a point in time $t$. Furthermore, we assume publication content is certain, and its content and time of appearance do not change after being received by broker $b$. We further assume publications are assigned unique timestamps when entering the system, so that all publications can be ordered by their time of appearance. This is indeed true for a centralized system with a single node assigning unique timestamps to incoming publications, while this assumption does not hold in a distributed setting. However, the assumption can be further relaxed in a distributed setting, so that each node assigns unique timestamps to its incoming publications, and thus the model does not require time synchronization between the nodes in the entire system.

**Boolean Subscription.** A Boolean subscription $n \in N$ is defined as a time independent Boolean matching function over $M$ which is activated by subscriber $s \in S$ at a point of time $t$, while a broker $b \in B$ responsible for processing $n$ receives it at a point in time $t^b$. We say that the time of appearance $t^b$ denotes a point in time when subscription $n$ appears in the system, i.e., when it is received by broker $b$. Broker $b$ is responsible to perform matching of incoming publications to subscription $n$ because subscriber $s$ is connected to broker $b$ at a point in time $t$. The Boolean matching function assigns a Boolean value, either true or false, to all publications in $M$. It is important to stress that the Boolean matching function is time-independent and depends exclusively on the publication content. Thus, since both the matching function and publication content are static, a Boolean subscription is a *stateless* publication filter which filters out all publications which are assigned the value “false” and passes the publications marked “true”.

**Matching Publications.** The set of publications matching a Boolean subscription $n \in N$ is comprised of publications which are assigned the Boolean value *true* by the $n$’s matching function. The set of matching publications is obviously a subset of $M$.

**Boolean publish/subscribe system.** A Boolean publish/subscribe system is a quintuple $PS=\langle B,P,S,M,N \rangle$ if for each Boolean subscription $n \in N$, the system delivers all matching publications to the subscriber $s$, which issued the subscription $n$. Each matching publication should be delivered to the subscriber $s$ exactly once, and immediately after being matched by a broker that is responsible for processing it, while the system does not deliver any non-matching publications to the subscriber $s$.

It is important to stress that a broker is responsible for processing all publications and subscriptions which it receives from connected publishers and subscribers. The set of connected publishers for broker $b$ is a subset of $P$ and varies over time. Analogously, the set of connected subscribers for broker $b$ is a subset of $B$ and also varies over time. In case the system is composed of a single broker, this broker is responsible for processing all publications and subscriptions defined by all publishers in $P$ and all subscribers in $S$. We discuss possible broker organisations within the
cloud and strategies for dividing the load generated by publishers and subscribers in Section 2.3.

Note that a publisher $p$ which has defined publication $m$ with time of appearance $t^p$ may decide to cancel $m$ by sending a message to its broker $b$ that receives such a message at a point in time $t^b > t^p$. Broker $b$ can from the point in time $t^b$ declare publication $m$ as a cancelled publication and broker $b$ will no longer match this publication to newly arriving matching subscriptions and reconnecting subscribers that have interest in the previously cancelled publication. Analogously, a subscriber $s$ which has defined subscription $n$ with time of appearance $t^n$ may decide to cancel the active subscription $n$ by sending a message to its broker $b$ that receives such a message at a point in time $t^b > t^n$. All incoming publications after $t^n$ are no longer matched to subscription $n$ by broker $b$.

### 2.2.2 Top-k/w Publish/Subscribe System

A top-k/w publish/subscribe system (top-k/w system) is defined, analogously to the Boolean system, as a quintuple $\text{PS}=(B,P,S,M,N)$, where $B$ is a finite set of brokers, $P$ is a finite set of publishers, $S$ is a finite set subscribers, $M$ is a finite set of publications, and $N$ is a finite set of top-k/w subscriptions. Similarly to the Boolean system, a publisher $p \in P$ may publish or cancel publications from $M$, while a subscriber $s \in S$ may activate or cancel subscriptions from $N$. A broker $b \in B$ performs the matching between active subscriptions and publications defined by a subset of publishers from $P$ and subset of subscribers from $S$ that are currently connected to $b$.

**Top-k/w Subscription.** A top-k/w subscription $n \in N$ is also activated by subscriber $s \in S$ at a point of time $t$, while a broker $b \in B$ responsible for processing $n$ receives it at a point in time $t^b$. However, in contrast to a Boolean subscription which requires a Boolean matching function over the set of publications $M$, a top-k/w subscription $n$ requires the following:

1. a time-independent scoring function $u_s: M \rightarrow \mathbb{R}$,
2. parameter $w \in \mathbb{R}^+$ which defines the size of subscription window (time-based window) and
3. parameter $k \in \mathbb{N}$ denoting the number of top publications from the window that have to be delivered to subscriber $s$.

Furthermore, as top-k/w subscriptions are continuous, they have a predefined time of activation and cancellation, which are implicit timestamps assigned by responsible brokers. A point in time $t^b$ represents the *time of subscription activation*, while $t^c$ is the *time of subscription cancellation*. Analogous to the Boolean subscription, we say that a top-k/w subscription is active within the time interval $(t^a, t^c]$.

Scoring functions, in general, are application specific, meaning that they depend on the application scenario in which the top-k/w system is used. The scoring functions are defined over the set of publications, they are time-independent, and depend exclusively on publication content. Note that this model supports generic scoring functions, because we do not impose any specific constraints on scoring function definition, such as monotonicity, which is frequently assumed by many top-k processing techniques [Ilyas2008]. Our assumption is that a scoring function produces a real number based on publication content, so that the publications can be ranked consistently, because a tie-breaking criterion can give preference to a more recent object. Scoring functions may assign ranks to publications in descending or
ascending order, based on their scores. Without loss of generality, in this report we assume the applied scoring function is such that lower scores are preferable to higher scores, and therefore assign ranks to publications scores in ascending order. Notice that although data objects are static and their scores do not change over time, their rankings may change over time as new publications appear in the system, while previously appeared publications are dropped from the subscription window.

A publication is considered to be within the subscription window at a point in time $t$, if the publication appears in the system after subscription activation, and less than $w$ time units have passed since its appearance. Only publications within a subscription window are of interest to a top-$k/w$ subscription, and thus the model supports ad-hoc subscriptions referencing future publications.

**Top-k publications in a subscription window.** A publication $m$ is a top-$k$ publication for a subscription $n$ at time $t$ if $m$ is within the current window of $n$, and there are less than $k$ higher ranked publications than $m$ in this window. Note that the set of top-$k/w$ publications is continuously being updated and changes over time, and that $m$ is delivered to the subscriber issuing $n$ when $m$ becomes an element of the top-$k/w$ set for the first time. Since a set of top-$k/w$ publications depends on other publications in the subscription window, top-$k/w$ subscriptions are stateful publications filters.

**Top-k/w publish/subscribe system.** A top-$k/w$ publish/subscribe system is a quintuple $PS=(B,P,S,M,N)$ if for each top-$k/w$ subscription $n \in N$, the system delivers all top-$k/w$ publications to the subscriber of $n$, and each of these publications is delivered exactly once and immediately after it becomes an element of the top-$k/w$ set for the first time, while the system does not deliver non-top-$k/w$ publications to the subscriber of $n$.

It can be shown that a top-$k/w$ system is a generalization of a Boolean system, i.e. that every Boolean system is a special case of the corresponding top-$k/w$ system. We can easily achieve that a top-$k/w$ system behaves as a Boolean system by selecting specific values of top-$k/w$ subscription parameters and scoring function, and thus every top-$k/w$ system implementation also supports Boolean subscriptions without further modifications. This allows us to have Boolean and top-$k/w$ subscriptions simultaneously active in the same publish/subscribe system, as it is the case for our system presented in Section 3. A detailed discussion of top-$k/w$ properties and its comparison with Boolean subscriptions is available in [Pripužić2014a].

### 2.3 Self-organizing characteristics: scalability and elasticity

The main strengths of the publish/subscribe communication paradigm are: a) loose coupling of communicating components, and b) system scalability. Loose coupling of system components is achieved by anonymous communication, since the processes, i.e., the publishers and the subscribers, do not have to be aware of each other to communicate. Brokers can be regarded as trusted intermediaries that enable such communication and integration to take place. This property allows the system topology to be dynamic, as information sources and destinations are independent of each other. Moreover, publishers and subscribers can appear, connect to the system, disconnect, and then also seamlessly reconnect. The loose coupling
property enables publish/subscribe systems to be scalable, because they can be seamlessly distributed over large, and possibly heterogeneous, networks. In addition, in such architectures parallelization of operations can be implemented, and message caching is natively supported. Distributed publish/subscribe solutions have so far been extensively researched, but efficient solutions targeting cloud-based environments have so far not been adequately addressed.

System scalability may deteriorate when publish/subscribe is used in a tightly-coupled high-volume environment when, for example, thousands of servers within a cloud are sharing the same publish/subscribe infrastructure. In such cases, problems like instability of the throughput, possible processing slowdowns, and IP broadcast storms may appear. The use of publish/subscribe solutions in such environments requires specific cloud-based architectures that can offer elastic real-time computation under varying load. The load in this case is generated by a varying number of publishers and subscribers: It depends on the number of concurrent active subscriptions and the joint publication rate of all publishers. However, one can argue that the main driver for resource allocation within the cloud needs to be tightly coupled with the publication rate since publications generate the processing load. In other words, if there are no new publications, there is no need to perform the matching process, even in the case of a huge number of active subscriptions.

Brokers are the main processing components of all publish/subscribe systems since they are responsible for receiving publications from publishers, matching of active subscriptions to publications, and delivery of matching publications to subscribers. A cloud-based broker needs to offer scalable and efficient matching service: The service has to be elastic, so as to adapt well to the current workload. In other words, it should be able to process many subscriptions in parallel to minimize the matching overhead per publication. Additionally, a cloud-based implementation should be able to increase or reduce the number used resources within the cloud according to the matching load which primarily depends on the frequency of incoming publications.

There are two potential parallel architectures for cloud-based publish/subscribe:

1. a flat architecture composed of independent brokers and a central coordinator (Figure 3), and,
2. a hierarchical architecture with a central coordinator managing a number of broker trees (Figure 4).

![Figure 3 Flat cloud broker architecture](image1.png)

![Figure 4 Hierarchical cloud broker architecture](image2.png)
A flat architecture with independent brokers relies on a central coordinator to divide the processing load among the brokers. There are two possible scenarios for dividing the load: a) each broker is responsible for a subset of subscriptions, while each incoming publication is replicated and forwarded to all brokers, and b) all brokers are responsible for all subscriptions, but each incoming publication is submitted for matching to a single broker in, for example, a round robin fashion. In this architecture the coordinator can take care of resource allocation in case of the varying processing load. It can initiate and assign more broker instances in case of a high rate of incoming publication when the cloud matching performance deteriorates, or replicate one broker in case it becomes overloaded. It can also stop and release broker instances that become idle over time, and reassign their responsibilities among other brokers. The flat architecture may be applied for processing both Boolean and top-k/w subscriptions.

A hierarchical broker architecture organises broker matchers into a hierarchy. This is possible due to the inherent property of Boolean subscriptions, subscription coverage. A Boolean subscription $s_1$ is said to be covered by another Boolean subscription $s_2$ if every publication which matches $s_1$ also matches $s_2$, while $s_1$ is more specific than $s_2$. The subscription coverage property can be used to organize subscriptions into a subscription forest. (The term forest comes from graph theory, where a forest is defined as a disjoint union of trees.) Subscription forest is used to improve the broker matching performance since a matching algorithm based on subscription covering organizes subscriptions into trees such that a leaf node subscription is covered by a parent node subscription. Therefore, when a publication matches a subscription from a tree, all its leaf subscriptions are also matching, and the algorithm does not need to check the matching relationship between the publication and all subsequent leaf node subscriptions. Since, in practice, two subscriptions can be such that they do not cover each other, a single tree is not sufficient to represent all subscriptions. They are therefore represented by a subscription forest. More details on subscription forest are given in Section 4.4.

Subscription forest can be mapped onto a hierarchical broker architecture where each broker is responsible for different parts of the subscription forest. At this point it is not obvious how to map top-k/w subscriptions to the hierarchical broker architecture.

Both the flat and hierarchical architectures can be used in our cloud-based publish/subscribe implementation, as further discussed in Section 4.1.
3 PUBLISH/SUBSCRIBE SYSTEM ARCHITECTURE

CUPUS is a cloud-based publish/subscribe middleware designed for the (mobile) IoT environments which consists two types of processing engines: a) a cloud-based solution which is designed to process big data streams stemming from IoT environments and b) processing engine tailored for mobile devices (Mobile Broker) which is presented in deliverable D3.4.2. The developed Cloud-based Publish/Subscribe Processing Engine (CPSP Engine) described within this deliverable supports both Boolean and top-k/w subscriptions as defined in Section 2.2. For Boolean subscriptions we support a rich predicate language similar to [Sadoghi2011] with an expressive set of operators for the most common data types: relational operators, set operators, prefix and suffix operators on strings, and the SQL BETWEEN operator. Specification of the supported subscription language is available in Section 4.3. Moreover, the CPSP engine is designed to be elastic and to support dynamic allocation of cloud resources in accordance with the cloud processing load, as discussed in Section 2.3.

The main tasks of the CPSP engine are the following:

1. to receive and manage received publications and subscriptions from publishers and subscribers,
2. to perform matching of active subscriptions to publications, and
3. to deliver matching publications to subscribers.

In view of these tasks, in this section we present the interaction of the CPSP engine with its environment (publishers and subscribers), and the rest of the OpenIoT platform. All components comprising the resulting publish/subscribe system and their interactions are depicted in Figure 5. The left-hand side of this figure shows typical publish/subscribe components: publisher and subscriber interacting with the CPSP engine. Publishers and subscribers are processes that can connect to and disconnect from the CPSP engine (methods connect and disconnect). Publishers can publish a publication and revoke a previously published publication (methods publish and unpublish). Subscribers can define a new subscription and cancel an existing subscription (methods subscribe and unsubscribe). The CPSP engine notifies a subscriber with a publication matching at least one of subscriber’s active subscriptions (method notify).

In Figure 5, we can see that the CPSP engine running within the cloud interacts with a mobile broker which is not used in typical publish/subscribe systems. Mobile broker is a component running on mobile devices that is used in our architecture to enable sensor discovery in mobile environments as well as energy-efficient and selective data acquisition from sensors attached to mobile devices. Mobile brokers serve as intermediaries between the CPSP engine and their “local” publishers and subscribers. By local publishers we refer to local sensor data sources, e.g., wearable or built-in sensors that relay their data through mobile devices to the CPSP engine. Local subscribers represent mobile device users and their subscriptions, i.e., their information needs.
Mobile brokers are a special type of brokers that can perform the matching of publications generated by local publishers before they are sent to the CPSP engine. This enables the filtering of sensor data close to data sources and can suppress the redundant sensing process and data delivery to the CPSP engine. The main idea behind this approach is to push the filtering of data from the cloud to the end-user devices to save the battery power on both sensors and end-user devices, which would otherwise be drained by unnecessary sensing and communication tasks. To enable data filtering on mobile brokers, a special mechanism is needed to maintain an appropriate and minimal set of subscriptions on mobile brokers to save resources on mobile devices. An appropriate set of subscriptions is such that it comprises all active CPSP subscriptions that can potentially match publications generated by local publishers. Hereafter we describe the mechanism and required methods to activate appropriate subscriptions on mobile devices.

Since a mobile broker serves as a proxy for local publishers and subscribers, it supports all of the previously defined methods specified for the CPSP engine which are invoked by publishers and subscribers. It handles connections and disconnections from the CPSP engine, and thus provides mobility transparency to local publishers and subscribers. It is used to publish and unpublish sensor data, to define subscriptions, and to receive notifications for local subscribers running on mobile devices. Sensor discovery is enabled by a special announce message which informs the CPSP engine about a new local publisher which is attached to a mobile broker. This message includes a description of the data types that can be produced by a local publisher.
The announce mechanism enables the CPSP engine to maintain the information about all available sensors in the system, and to turn them on when needed by sending subscriptions matching defined data types to mobile brokers. When receiving subscribe messages from the CPSP engine in cases when there is interest from other subscribers or the OpenIoT platform to receive sensor data published by its local publishers, a mobile broker can start or stop the corresponding sensor, or it can perform the filtering of sensor-generated data on the mobile device. For example, if a mobile user is interested to receive alerts from air quality sensors located within a defined geographical area when the readings from these sensors are above a certain threshold, the user can define new subscriptions specifying interest in, e.g., carbon monoxide (CO) and volatile organic compound (VOC) readings above the thresholds for the predefined area. These subscriptions are relayed through the mobile broker running on user mobile device to the CPSP engine, and further on to mobile devices attached to CO and VOC sensors which are currently residing within the predefined area. The CO and VOC sensors can be invoked to start periodical readings, if necessary, and the mobile device checks whether each received CO or VOC reading is above the predefined threshold. A detailed description of this procedure is available in Deliverable 3.4.2.

Finally, as we can see in the right-hand side of Figure 5, the CPSP engine communicates with the OpenIoT platform which is used for permanent storage of sensor data acquired through mobile devices in order to provide such readings to other OpenIoT services. In addition, the CPSP engine can receive sensor readings from the OpenIoT platform to enhance the quality of information delivered to CPSP engine subscribers.

The remainder of this section provides detailed description of interactions between the publish/subscribe system components.

3.1 Sequence diagrams

Connect and disconnect. The two methods are used by subscribers, publishers and mobile brokers when connecting to and disconnecting from the CPSP engine. Note that further on we denote either publishers or subscribers as CPSP engine clients. The method connect adds subscriber/publisher/mobile broker identifier into the list of connected components maintained by the CPSP engine, while the method disconnect removes them from the list. In case a subscriber or mobile broker reconnects to the CPSP engine after being disconnected, the engine first delivers all publications that have been matched to their active subscriptions while they were disconnected.

Publish. A new publish request defines a new publication which is subsequently matched to the list of stored active subscriptions previously received through subscribe requests (defined later on in this subsection). The matching process identifies a list of subscribers with matching subscriptions, and such subscribers are notified about this publication. Since publications can define their validity periods the CPSP engine needs to maintain a list of valid publications and thus there is no need for a special unpublish request. A sequence diagram depicting the delivery of a new publication to an interested subscriber is shown in Figure 6.
Announce. Figure 7 shows the sequence of events following a new announce event. When a mobile broker (MB) announces a new publisher, this information is forwarded to the CPSP engine in the corresponding announce message. The CPSP engine then stores the announcement in a list of stored announcements and compares it with the list of stored active subscriptions. If there are interested subscribers with subscriptions matching the announcement, the CPSP engine activates the publisher by forwarding matching subscriptions to the mobile broker in the corresponding subscribe message.

Revoke announcement. Any publisher connected to a mobile broker can revoke its previous announcement. As shown in Figure 8, when this happens, the CPSP engine just needs to delete the announcement from its list of stored announcements.
Additionally, if there are alternative publishers and matching subscriptions requesting this kind of information, the CPSP engine can activate alternative mobile brokers and their publishers.

![Diagram of CPSP engine and mobile brokers](image)

**Figure 8 Revoking an announcement**

**Subscribe.** When a new subscription is activated, the CPSP engine needs to deliver all matching and valid publications from its local storage to the new subscriber. In addition, a new *subscribe* event may activate matching publishers on mobile brokers since a new subscription can match a previous announcement. As we can see in **Figure 9**, the subscription is first added to the list of active subscriptions and then matched to stored announcements. Next, a subscribe message is sent to a mobile broker to activate a publisher whose previous announcement matches the new subscription. Finally, the subscription is matched to all valid publications, and matching publications are sent to the subscriber in a *notify* message. Note that when there are multiple matching publishers, the CPSP engine can decide which publisher to activate using one of the predefined QoS metrics such as the battery status or publisher reputation.

![Diagram of client, CPSP engine, and mobile brokers](image)

**Figure 9 Activating a new subscription**
Unsubscribe. Clients usually unsubscribe when there are no longer interested in particular publications. When the CPSP engine receives an unsubscribe message, it deletes the subscription from the list of stored subscriptions. Additionally, if the cancelled subscription was the only one remaining regarding the publication(s) of a specific publisher (connected to a mobile broker), an unsubscribe message is forwarded to its mobile broker, as shown in Figure 10, instructing this publisher to stop producing new publications.

![Figure 10 Cancelling a subscription](image)

3.2 Interaction with the OpenIoT platform

As the CPSP engine acquires sensor readings from mobile sensors by means of mobile brokers, there is a need to provide those readings to the rest of the OpenIoT platform. Moreover, as the engine does not include a permanent storage, but rather performs the matching and forwarding of mobile sensor data, all sensor data requiring permanent storage are stored and maintained by the OpenIoT platform, namely the LSM-Light component. To enable the forwarding of mobile sensor readings from the CPSP engine to the OpenIoT platform, the engine uses the Quality of Service (QoS) Manager component which is defined and described in detail deliverable D4.6. The QoS Manager manages the sensor data acquisition from various mobile data sources in order to optimize energy and bandwidth consumption while meeting application requirements. Hence, it serves as a “universal” subscriber to all publications inside the CUPUS middleware which need permanent storage, and forwards all such sensor reading to the OpenIoT platform through the X-GSN component, as shown in Figure 11. Since X-GSN does not directly interact with real physical sensors but rather with virtual sensor representations, we create one virtual sensor for each geographical area\(^1\) and map all mobile sensors located within this area to the virtual sensor. When the QoS Manager receives a new sensor reading, it first checks whether a virtual sensor exists in the observed area, and then forwards the received sensor readings to the X-GSN through a virtual sensor instance in which the mobile sensor is currently located. The sensor reading is annotated by X-GSN and subsequently stored in the OpenIoT Cloud Database (LSM-Light). Otherwise, if a

\(^1\) Geographic areas are divided in accordance with the military grid reference system (MGRS)\(^1\) where
corresponding virtual sensor does not exist for an area, the QoS Manager creates and initialises a new virtual sensor. All sensor readings for which there is current interest (i.e., an active subscription), either among subscribers of the CUPUS middleware or among the OpenIoT platform users, are forwarded to the OpenIoT platform. Otherwise, the readings from inactivated publishers are not forwarded to the OpenIoT platform. Note that a publisher on a mobile device is active if the mobile broker running on the device has received subscriptions matching publisher data from the CPSP engine, as discussed in Section 3.1, and the QoS Manager did not turn it off.

For a full integration of the publish/subscribe system with the rest of the OpenIoT platform, mobile sensors within a certain geographical area (regarded as virtual sensors generating data streams) need to be discovered by the OpenIoT platform and selected as resources for service provision. Thus, in case there is an OpenIoT service request which requires readings from mobile sensors, we need a mechanism which is initiated by the OpenIoT platform to activate adequate mobile publishers in a certain geographic area. This can be achieved by sending an explicit subscription from the OpenIoT platform via a web interface of the QoS Manager to the CPSP engine, as depicted in Figure 12. Since the OpenIoT platform does not know if these publishers are currently active or not, it has to request a new subscription covering the area of interest. After receiving such instruction, the QoS Manager will send explicit subscription on the sensor data in corresponding area to the CPSP engine. Consequently, mobile publishers will be activated and their data readings will be accessible in the OpenIoT platform.
Figure 12 Activating a new subscription from the OpenIoT platform

Error! Reference source not found. depicts our solution for integration of the OpenIoT platform and CUPUS middleware. Components extending the OpenIoT architecture include: Real-time Data Presentation, User-Specific Subscription Definition, CPSP Engine, and QoS Manager (outlined in red in the figure). In addition to the CPSP engine, QoS Manager enables the interaction of the CPSP engine with the OpenIoT platform based on the sequence diagrams depicted in Figure 11 and Figure 12. On one hand, the QoS Manager would act as a number of virtual sensors to the GSN Scheduler and enable insertion of mobile sensor data into the OpenIoT Cloud Database. On the other hand the QoS Manager would act as a subscriber to the CPSP engine, so that it acquires data which are received by the CPSP engine. In addition, as the GSN Scheduler can check requests for data for specific (virtual) sensors, this information can be used to activate mobile publishers in accordance with OpenIoT service requests and start adequate subscriptions on the QoS Manager. More details on CPSP and QoS Manager integration with the rest of OpenIoT platform are available in deliverable D4.6.
4 CLOUD-BASED PUBLISH/SUBSCRIBE PROCESSING ENGINE

4.1 Cloud-based architecture

In this section we present the architecture of the implemented CPSP engine. We refer to it in this document as the Cloud Broker. It is based on the generic data stream processing architecture defined in [Golab2003] where continuous queries, i.e., subscriptions, are maintained in processor memory for efficient processing of incoming data objects, i.e., publications, while publications needed for efficient processing are also stored in main memory. The processor accepts subscriptions and publications from distributed sources, and outputs multiple data streams, where each data stream is pushed to distributed subscribers. Publications from a single data source can be regarded as an incoming data stream. Each incoming publication is seen only when entering the processor unless explicitly stored in memory. In case of Boolean subscriptions, the processor needs to keep a history of valid publications until their expiration to perform the matching operation between valid publications and new incoming subscriptions. For top-k/w subscriptions the processor memory maintains only a subset of publications from the incoming data streams which are needed for efficient real-time top-k/w processing, and does not consider publication validity during the matching process since top-k/w subscriptions reference only future publications (as defined in Section 2.2.2).

Since the matching of publications to subscriptions is the most demanding task performed by the CPSP engine, we need to replicate the matcher processes and dynamically allocate cloud resources for matching in accordance with the processing load. This reflects the CPSP engine architecture given in Figure 13. There are logically two parts of the CPSP engine, a fixed and dynamic part. The fixed part enables engine communication with remote publishers and subscribers, and consists of three components: the MessageReceiver, DeliveryService and Coordinator. They are created on CPSP engine start-up, and have to exist during the entire lifetime of the engine. Those three components manage the communication to and from external entities, i.e., publishers, subscribers and mobile brokers running on mobile phones. The Coordinator is an intermediary component which enables the flow of client requests and internal messages between MessageReceiver, DeliveryService and the dynamic part of the engine. MessageReceiver accepts publications from publishers and subscriptions from subscribers via TCP. DeliveryService communicates with subscribers over TCP or Google Cloud Messaging (GCM).

The dynamic part of the CPSP engine consists of components called matchers that are created dynamically in accordance with the current engine load. Various types of matchers exist, e.g., Boolean and top-k/w matchers. Matcher processes within the cloud communicate with the DeliveryService via UDP to submit matching publications for subsequent delivery to subscribers.

Hereafter we explain the organisation of Boolean matchers that form a dynamic structure which is either flat or hierarchical.
4.1.1 Flat architecture implementation

The “flat” cloud version uses a fixed number of matcher components in parallel. Those matchers are the same as the RootMatcher component and are created and initialized on demand by the Coordinator. The Coordinator starts all the matcher components at startup, distributes the processing load among them, and has a direct connection with all of them. Note that, in this setup, a RootMatcher cannot have child matchers and cannot initiate the creation of a new matcher in case of overload.

This system has a simple way of processing subscriptions and publications. The Coordinator distributes received SubscribeMessage objects in a round-robin fashion to the matcher components. This results in all of the $N$ matchers having a subscription data structure containing $1/N$ of the total subscriptions being processed by the CPSP engine. Incoming publications (PublishMessage objects) are replicated by the Coordinator and sent to all the matchers. This can potentially slow down the
Coordinator since it has to make $N$ communication operations in a row, and the other components have to wait for it to finish, which makes the Coordinator a potential bottleneck of the system.

### 4.1.2 Hierarchical architecture implementation

The hierarchical architecture uses two kinds of matcher components: a RootMatcher and ChildMatchers. Each matcher component is responsible for a part of the subscription data structure that spans over all active subscriptions received by the CPSP engine. The RootMatcher is the top matcher and its parent is the Coordinator component. There are only two relevant differences between the two types of matchers. One is that the RootMatcher holds the root node of the whole data structure, and a ChildMatcher holds the root of the subset of the structure dedicated to it. The other difference is that ChildMatchers propagate messages up the matcher tree, while the RootMatcher is usually their “last stop”.

A parent matcher communicates asynchronously with its child using a special node called the proxy node. When a proxy node is created, it creates a new matcher component as a separate process, and is the only one that has the ability to communicate with it over the standard OS input/output streams. The newly created proxy node replaces a part of the structure located on the parent matcher, and sends it to the newly created ChildMatcher component.

The existence of child matchers is transparent even to the parent matcher itself, because they are hidden behind a seemingly regular node in the subscription data structure. Only proxy nodes are aware of the ChildMatcher components, and each of them only of its own.

This, coupled with the fact that each matcher can send UDP messages to the DeliveryService independently of any other component, enables the system to be loosely coupled, which means that only local information is required by each component. None of the components has an overview of the whole system, nor can it control it directly. This greatly reduces the architectural overhead in processing and communication.

The Coordinator is the central component of the system in its current design, although its only function is to be a relay between the MessageReceiver, DeliveryService and RootMatcher components, and can basically be omitted in a slightly different design. The Coordinator component starts the other three basic components as separate processes, and is the only component that has direct access to their input/output streams. Because of that, the only way those components can communicate with each other is through the Coordinator component, which reads messages from their input streams and forwards them to the appropriate output streams.

### 4.1.3 Publisher representation within the CPSP engine

The purpose of the MessageReceiver component is to accept connections and process requests from publishers and subscribers. MessageReceiver accepts TCP connections from mobile brokers, publishers and subscribers, and creates a handler object for each of them. The handler for a subscriber is called a SubscriberForBroker, a handler for a publisher is called a PublisherForBroker, and the handler for a mobile broker is called a MobileBrokerForBroker. A publisher can communicate with the
CPSP engine only through the MessageReceiver component, and it can send three different messages, as depicted in the sequence diagram in Figure 14.

![Sequence Diagram](image)

Figure 14 Communication of a publisher with the CPSP engine

When MessageReceiver receives a `PublisherRegister` message, it creates a PublisherForBroker handler object for that publisher, which processes all subsequent requests from that publisher. A `PublisherDisconnect` message triggers the destruction of the publisher's handle, its removal from the MessageReceiver and the teardown of the TCP connection. The only other message a publisher handler can process is the `Publish` message, which is used for both publishing and unpublishing a publication. Upon receiving a `Publish` message, the publication is saved on the MessageReceiver, and the `Publish` message forwarded to the Coordinator component, which in turn just forwards it to the RootMatcher component, after which the matching process begins. A subscriber's communication with the engine is somewhat more complex, as can be seen in Figure 15 and Figure 16. The main reason is that the engine has to be able to notify a subscriber about a publication at any given time and from a different engine component, and also because a subscriber can remain registered while not connected (this is not necessary for a publisher).

### 4.1.4 Subscriber representation within the CPSP engine

Figure 15 depicts a sequence diagram which shows the interaction between CPSP engine components with a subscriber when the subscriber is the one initiating all communication. Upon receiving the `SubscriberRegister` message, the MessageReceiver component creates a SubscriberForBroker handler object for the subscriber, which processes all subsequent requests from that subscriber. After that, the MessageReceiver forwards the `SubscriberRegister` message to the Coordinator, which forwards it to the DeliveryService. A `SubscriberDisconnect` message simply puts the subscriber's handler object in an inactive state, not removing it from the MessageReceiver, and is also forwarded to the DeliveryService via the Coordinator. A `SubscriberUnregister` message triggers the destruction of the subscriber's handler, its removal from the MessageReceiver, and the teardown of the TCP connection, and is also forwarded to the Coordinator. The Coordinator duplicates the message, sending one copy to the DeliveryService and the other to the RootMatcher, because all the subscriptions of the unregistering subscriber need to be removed from
subscription data structures upon its deregistration. A *Subscribe* message, used for both subscribing and unsubscribing a subscription, is processed by the subscriber’s handler in two steps. The first step is to simply forward the message to the RootMatcher via the Coordinator, and the second step is to match the new subscription against all active publications that have entered the engine, and send an *InitialMatches* message containing all the matched publications to the DeliveryService via the Coordinator.

![Diagram of communication between Broker, Coordinator, SubscriptionReceiver, and DeliveryService](image)

**Figure 15** Communication of a subscriber with the CPSP engine (the Message Receiver part)

![Diagram of communication between Coordinator, DeliveryService, and Subscriber](image)

**Figure 16** Communication of a subscriber with the CPSP engine (the Delivery Service part)
Figure 16 depicts a sequence diagram which shows the interaction between CPSP engine components with a subscriber when the engine is the one initiating the communication, i.e., when the engine needs to deliver matching publications to the subscriber. The purpose of the DeliveryService component is to receive matched publications for registered subscribers from other engine components (mostly matchers), and to send them to the appropriate subscribers if it is able to do so, or else to hold them in a queue until being able to send them. The DeliveryService has a TCP connection to every connected subscriber, and a queue object for every registered subscriber. When a SubscriberRegister message is received from the Coordinator, the DeliveryService initiates a TCP connection with the subscriber and creates a queue object (if the subscriber has previously not been registered with the engine), as can be seen in Figure 16. In the TCP connection the role of the "server" is with the subscriber, which expects connection requests from the engine: Messages are only sent from the DeliveryService to the subscriber without any response. A SubscriberDisconnect message from the Coordinator causes a teardown of the TCP connection to the disconnecting subscriber, but not the removal of the queue object, which afterward accumulates publications for the subscriber until the subscriber reconnects to the Coordinator. When the subscriber reconnects, the accumulated publications are sent to it by the DeliveryService from the corresponding matcher's queue. A SubscriberUnregister message destroys both the TCP connection and the queue object of an unregistering subscriber and removes it completely from the DeliveryService.

Finally, the DeliveryService can receive the InitialMatches message from the Coordinator component. This message carries a set of publications which need to be sent to a subscriber and the DeliveryService inserts those publications into the subscriber's queue object where they wait to be delivered to the subscriber. In addition, the DeliveryService also acts a UDP server which receives UDP packets from matcher components. Each UDP packet from a matcher contains a publication and a set of subscriber ID's and the DeliveryService inserts each publication into the corresponding subscriber queues.

4.2 Load balancing within the CPSP engine

The elasticity and scalability of the CPSP engine are achieved through the splitting and merging of matcher components. The current implementation of the CPSP engine supports the flat cloud broker architecture where both the splitting and merging processes are triggered by the idle times measured by a single matching process. The idle time is measured for the last $N$ received and processed messages, where $N$, splitting threshold and merging threshold are parameters of the CPSP engine. Thresholds are defined in the configuration file as the percentage of idle time in the observed window.

4.2.1 Splitting of matchers

The splitting of a matcher is initiated when the idle time of a matcher process in the window of $N$ last received messages is smaller than a splitting threshold, i.e., when the matching process is active most of the time during the window time span,
expressed as percentage of the total window time. The splitting criterion defined in this way is robust with regard to sudden spikes in processing load.

Figure 17 Processing of a received message followed by a matcher splitting trigger

The process of matcher splitting is depicted in Figure 17. After a message is received by the Coordinator, and processed by the RootMatcher, and if necessary forwarded to the DeliveryService, the RootMatcher checks whether it was overloaded during the observed time window. If the received message is last one within the observation window, the matcher process checks the percentage of its idle time during the time window. If the percentage of idle window time is smaller than the given splitting threshold, a splitting trigger event is fired. Since the matcher is under high load, i.e., it takes too much time to process incoming messages, its subscription structure needs to be reduced. To reduce the load on the matcher, approximately one half of its subscriptions will be forwarded to a newly created matcher. The Coordinator is in charge of managing this procedure, since it has the knowledge about all components within the CPSP engine, while the ParentMatcher decides which subscriptions it wants to transfer to the new matcher. The Coordinator creates a new matcher instance (i.e., either a Boolean or top-k/w matcher) and forwards subscriptions to it that are sent by the RootMatcher. The RootMatcher is notified about the success of the splitting process such that it can remove transferred subscriptions from its structure. If the splitting process does not finish successfully, the setup of the CUPUS middleware will remain unmodified. During the splitting process, double matching can occur for the subset of subscriptions, both on the RootMatcher and new matcher. However, this is not a serious problem for the CPSP engine as the DeliveryService can remove duplicate notifications for the same subscriber. In the end of the matcher splitting procedure, the new matcher becomes a part of the CPSP engine with the same status as all other running matchers and is used in the round robin strategy of subscription forwarding to acquire new subscriptions.
4.2.2 Merging of matchers

The merging of a matcher process is initiated when the idle time of a given matcher process in the observed time window is higher than a given merging threshold, i.e., when the matching process is not active most of the time during the observed time window, expressed as the percentage of the total window time. Similar to the splitting criterion, this criterion is chosen for its robustness with respect to sudden spikes in processing times. It is set before the engine is started and cannot be changed during runtime.

![Diagram of matcher merging process](image)

**Figure 18** Processing of received message followed by a matcher merging trigger

Similar to the process of matcher splitting, a matcher checks whether it has reached the end of its observation window and checks the percentage of its idle time to identify a merging trigger. If the idle time is greater than the given merging threshold, a merging event is fired as depicted in Figure 18. Since the observed matcher is underloaded, i.e., the process is idle most of the time, the matcher can be merged with other matchers in the system which will take over its subscriptions. The Coordinator is in charge of the matcher merging process, since it has the knowledge about all other components within the CPSP engine. The coordinator receives all subscriptions from the merging matcher and forwards them to other matchers in the engine using the round robin strategy. The merging matcher is notified about the success of the merging process such that it can gracefully disconnect from the engine and destroy itself. If the merging process does not finish successfully, the setup of the CUPUS middleware will remain unmodified. This process is required because the Coordinator needs to assign all subscriptions to other matchers who will process user publications in future. A potentially double matching process on the ParentMatcher and other matchers during the merging procedure is not a serious problem for the CPSP engine as the DeliveryService can remove duplicate notifications for the same subscriber.
4.3 Subscription language

Boolean subscriptions. A Boolean subscription is defined as a set of triplets. A triplet consists of an attribute value, a constraint value, and an operator that puts the given attribute value and constraint value in a relation. Each triplet represents a logical predicate, therefore, a subscription is a conjunction of predicates that all have to be true in order for a publication to match a subscription.

A list of supported operators is given in Table 1. The table provides operator descriptions with examples, where \( a \) is an attribute value and \( c \) is a given constraint.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LESS_THAN</td>
<td>a numeric operator ((a &lt; c))</td>
</tr>
<tr>
<td>LESS_OR_EQUAL</td>
<td>a numeric operator ((a \leq c))</td>
</tr>
<tr>
<td>EQUALS</td>
<td>both a numeric and a string operator ((a == c))</td>
</tr>
<tr>
<td>GREATER_THAN</td>
<td>a numeric operator ((a &gt; c))</td>
</tr>
<tr>
<td>GREATER_OR_EQUAL</td>
<td>a numeric operator ((a \geq c))</td>
</tr>
<tr>
<td>BETWEEN</td>
<td>a numeric operator that takes two values, a lower and an upper bound ((c_1 \leq a \leq c_2))</td>
</tr>
<tr>
<td>CONTAINS_STRING</td>
<td>a string operator that matches all strings that contain a given string</td>
</tr>
<tr>
<td>STARTS_WITH_STRING</td>
<td>a string operator that matches all strings that start with a given string</td>
</tr>
<tr>
<td>ENDS_WITH_STRING</td>
<td>a string operator that matches all strings that end with a given string</td>
</tr>
</tbody>
</table>

Numeric operators support the following primitive data types: int, long, float, double, String, boolean. The string operators (except the EQUALS operator) are internally translated to the \( \in \) (element of) operator. The system performs such translation on all triplets defined over a string attribute for every subscription that enters a CPSP engine, and finds all possible values of that attribute that could satisfy the predicate defined by the triplet. All such values comprise the set of values of the \( \in \) operator (if the set is empty the subscription is dismissed because it can never be satisfied).

Hereafter we provide an example subscription \( s_i \) and example publication \( p_j \) supported by the CPSP engine.

\[
s_i = [\text{NO}_2 > 40 \mu \text{g} m^{-3} \text{ AND } 45.81 \leq \text{lat} \leq 45.82 \text{ AND } 15.96 \leq \text{long} \leq 15.98]
\]

\[
p_j = [\text{NO}_2 == 45 \mu \text{g} m^{-3} \text{ AND } \text{lat} == 45.81543 \text{ AND } \text{long} == 15.97433]
\]

The matching process needs to determine that the example publication \( p_j \) matches subscription \( s_i \), and to deliver \( p_j \) to a subscriber that has defined \( s_i \).
**Top-k/w subscriptions.** Similarly to a Boolean subscription, a top-k/w subscription is defined as a set of triplets extended by parameters $k$ and $w$, and a scoring function. A triplet consists of an attribute, a constraint value, and an operator that puts the given attribute and constraint value in a relation. Each triplet represents a logical predicate, therefore, a subscription is a conjunction of predicates that represents the “starting point” for the evaluation of an incoming publication. Parameter $k$ defines the number of notification that a user wants to receive during the time window $w$. The scoring function is essential because it assigns scores to incoming publications and enables the matching algorithm to define which publications should be delivered to a user (i.e., to select the best $k$ publications).

An example top-k/w subscription we present here is a top-k/w subscription using weighted sum as its scoring function. The weighted sum scoring function computes the distance for each sensor data object attributed to the set of triplets defined by a top-k/w subscription and produces a weighted sum of the distances. When the top-k/w subscription defines a triplet with the GREATER_THAN operator then publications with higher values are given better scores. In the case of a LESS_THAN operator, publications with lower values receive better scores. For example, a user can define a subscription such that over time he/she receives up to 10 air quality publications in one day that are the closest to his/her home. The corresponding top-k/w subscription $s_i$ would include the following triples that define user home location, e.g. $[\text{lat}=45.815 \text{ AND long}=15.977]$, parameters $k=10$ and $w=24$ hours, while the scoring function is the distance for latitude and longitude triplets. The publication in this case remains the same as for Boolean subscriptions, e.g. $p_j=[\text{NO}_2 == 45 \text{µg/m}^3 \text{ AND lat}=45.81543 \text{ AND long}=15.97433]$.

The matching process calculates the distance of measurement $p_j$ to home location expressed in $s_i$ to evaluate its utility (i.e. “goodness”), and will deliver $p_j$ to the subscriber if and when $p_j$ becomes a member of the top-$10$ measurements for the user within a single day (which means the readings which are nearest to the home location).

The implementation of top-k/w subscriptions in the CUPUS middleware is available in the TripletTopKWSubscription class. Since various utility functions for top-k/w subscriptions can be defined in accordance with user preferences, the CUPUS middleware can be extended with additional types of top-k/w subscriptions by providing various utility functions and rules for their matching with incoming publications. We have chosen top-k/w subscriptions with weighted sum scoring function as representative subscriptions for the CUPUS middleware since they can be used for in scenarios in which users want to monitor over time the worst or best observations with respect to a specific location. Top-k/w subscriptions are also applicable in mobile crowd-sensing applications, as e.g., in crowd-sensed air quality monitoring when users want to monitor and receive over time the most unfavourable air quality readings within a particular geographical area. Users can receive, for example, top-$10$ such readings during a day and shortly after such readings are sensed and delivered to the CPSP engine.
4.4 Matcher implementation

4.4.1 Subscription forest

A procedure for comparing publications with a set of subscriptions is the most important task of a matcher. The objective of the matching algorithm is to efficiently find a set of subscriptions that match a publication, to identify subscribers of the matching subscriptions, and then to send the publication to all interested subscribers.

The first prerequisite for the implementation of the matching algorithm comparing a publication with a set of subscription is to add a function that checks coverage relationship between a subscription and publication parameters. This is achieved by introducing triplets, i.e., each parameter is defined by a triplet: object, value and operator. By this structure, each parameter can be compared to any other, and it can easily be determined whether a subscription is covering a publication, i.e. a publication is a point in a subscription subspace.

A subscription data structure is a forest (i.e. a set of trees), where each node is a single subscription (i.e. request for information from a user). For simplicity we depict a simple subscription forest comprising a single tree in Figure 19. A tree subscription is a partially ordered set, where each node covers all of their children nodes (i.e. a subscription of a parent node is more general than a child’s subscription), and may have more children, but only one parent. The process of building the forest is incremental and localised, so whenever a node is added or removed from the forest, all relationships within the rest of the forest are not affected.

The processing of incoming publications is efficient since the algorithm can very quickly identify parts of the forest that cover incoming publications. For matching purposes, if a subscription node within a tree (forest) is not covering a publication, all child nodes of the subscription nodes are also not covering the publication, and are not considered at all for delivery of the publication.

Figure 19 An example of a Boolean subscription forest

A subscription data structure is a forest (i.e. a set of trees), where each node is a single subscription (i.e. request for information from a user). For simplicity we depict a simple subscription forest comprising a single tree in Figure 19. A tree subscription is a partially ordered set, where each node covers all of their children nodes (i.e. a subscription of a parent node is more general than a child’s subscription), and may have more children, but only one parent. The process of building the forest is incremental and localised, so whenever a node is added or removed from the forest, all relationships within the rest of the forest are not affected.

The processing of incoming publications is efficient since the algorithm can very quickly identify parts of the forest that cover incoming publications. For matching purposes, if a subscription node within a tree (forest) is not covering a publication, all child nodes of the subscription nodes are also not covering the publication, and are not considered at all for delivery of the publication.
4.4.2 Top-k/w processing

To efficiently process incoming publications and match them with stored top-k/w subscriptions, we employ the Strict candidate pruning Algorithm (SA) which continuously maintains a k-skyband associated with each top-k/w subscription as explained in detail in [Pripužić2014b].

Each publication is associated with three attributes: 1) its score for the subscription, 2) its dominance counter (i.e. the number of publications in the k-skyband that are younger and scored better), and 3) exact time when it will be dropped from the subscription window. We represent the k-skyband in memory using two singly-linked lists sorted by descending object scores (i.e. ascending ranks) as shown in Figure 20, where each publication score is written in the square that represents it. The top list stores k best ranked publications, while the candidate list stores non-dominated publications for the subscription. A dominated publications has at least k younger and better scored publications in the current subscription windows. Publications in the candidate list can become top list publications when some of the top list objects become dropped from the subscription window.

![Figure 20. Adding an incoming publication into a strict k-skyband.](image)

The processing of an incoming publication is done in three steps. We traverse elements in both lists starting from the head (highest ranked) of the top list to the tail (lowest ranked) of the candidate list. In the first step we find correct position of the publication in one of the lists, according to its score. After that, in the second step we insert the publication to its position. Finally, in the third step, we traverse publications with lower ranks and remove publications which are dominated. Additionally, from these lists we also remove publications which are dropped from the query window.
5 API SPECIFICATION AND EXAMPLES

This section is devoted to the presentation of the CPSP engine components that enable a user/developer to download, install and use the entities of the developed publish/subscribe engine. Since the engine will keep evolving over time, an updated version of the information provided in this section will be provided regularly at the OpenIoT Wiki\(^2\) space under the Documentation\(^3\) section.

5.1 Cloud Broker

As already stated, the OpenIoT Cloud Broker, i.e., the CPSP engine, accepts all messages from subscriber, publisher and mobile broker entities, based on user or ICO inputs. After processing the message, the cloud broker if necessary sends a notification or subscription to the end user entity (a subscriber, publisher or mobile broker).

5.1.1 Main Released Functionalities & Services

The current release of the OpenIoT Cloud Broker implements the functionalities/capabilities that are reflected in the Table 2.

<table>
<thead>
<tr>
<th>Service Name</th>
<th>Input</th>
<th>Output</th>
<th>Info</th>
</tr>
</thead>
<tbody>
<tr>
<td>Broker (constructor)</td>
<td>String name, int brokerPort</td>
<td>Broker</td>
<td>Used to create a Broker entity. Requires as input an entity name in String format and port number. The output is a Broker object which is used for processing of all incoming messages on a specified port.</td>
</tr>
</tbody>
</table>

---

Service descriptions as well as their inputs and outputs are listed in Table 3.

---

\(^2\) https://github.com/OpenIoTOrg/openiot/wiki

\(^3\) https://github.com/OpenIoTOrg/openiot/wiki/Documentation
### 5.1.2 Download, Deploy & Run

#### 5.1.2.1 Developer

**5.1.2.1.1 System requirements**

To build this project you need Java 7.0 (Java SDK 1.7) or later, and Maven 3.0 or later. The service of this project is designed to be run on a cloud server.

**5.1.2.1.2 Download**

To download Cloud Broker’s source code use your favourite git client and retrieve the code from one of the following URLs:

- HTTPS: https://github.com/OpenIoTOrg/openiot.git
- SSH: git@github.com:OpenIoTOrg/openiot.git

The cloud broker is available under the folder "openiot/modules/CUPUS".

**5.1.2.1.3 Deploy from the source code**

If you have not yet done so, you must Configure Maven before testing the Cloud Broker deployment. After that:

- Build and Deploy the Cloud Broker
  - NOTE: The following build command assumes you have configured your Maven user settings. If you have not, you must include Maven setting arguments on the command line.
    1. Open a command line and navigate to the root directory of the CUPUS project.
    2. Type this command to build and deploy the archive:
       - mvn package
3. This will build the service in target folder and now service is ready to be started by command in terminal:
   \[ \text{java } -jar \text{target\StartBroker.jar } \langle\text{path_to_config_file}\rangle<\text{classpath}> \]

   - Undeploy the Cloud Broker service
     1. Stop the running instance of the service by typing “QUIT”

5.1.2.1.4 Run in Eclipse

5.1.2.1.4.1 Deploying Cloud Broker

To integrate and deploy the Cloud Broker in Eclipse one should follow steps below:

1. Import Existing maven project “File>Import>Maven>Existing Maven Projects”
2. Click the “Browse” button and navigate to the CUPUS source code directory that has been previously downloaded.
3. Click the Finish button.
4. Open the org.openiot.cupus.examples package
5. Right click on the “BrokerExample.java” choose “Run As>Java Application”

To undeploy the Cloud Broker follow the steps below:

1. Type “QUIT” or stop the system in the Console window

5.1.2.2 User

5.1.2.2.1 System requirements

All you need to run this project is Java 7.0 (Java SDK 1.7) or later. You can download the binaries through the OpenIoT Wiki under the Users>Downloads section.

5.1.2.2.2 Deployment/Undeployment

**Deploy**: To deploy the Cloud Broker service, copy the “CloudBroker.java” to the cloud instance. Run the terminal command: \[ \text{java } -jar \text{StartBroker.jar } \langle\text{path_to_config_file}\rangle<\text{classpath}>. \]

**Undeploy**: Stop the running instance of the service by typing “QUIT”.

---

4 An example of the config file is given in the source/main/resource folder
5 https://github.com/OpenIoTOrg/openiot/wiki
6 https://github.com/OpenIoTOrg/openiot/wiki/Downloads
5.2 Publisher

As already stated, the OpenIoT Publisher sends a publication to the system, based on user input or ICO reading. It sends all defined publications to the system, without any pre-filtering.

5.2.1 Main Released Functionalities & Services

The current release of the OpenIoT Publisher implements the functionalities/capabilities that are reflected in Table 4.

Table 4 Publisher public methods and constructor

<table>
<thead>
<tr>
<th>Publisher (name:String, brokerIP:String, brokerPort: int): Publisher</th>
</tr>
</thead>
<tbody>
<tr>
<td>connect (): void</td>
</tr>
<tr>
<td>disconnectFromBroker(): void</td>
</tr>
<tr>
<td>reconnect(): void</td>
</tr>
<tr>
<td>reconnect(brokerIP:String, brokerPort: int): void</td>
</tr>
<tr>
<td>publish(publication: Publication): void</td>
</tr>
<tr>
<td>unpublish(publication: Publication): void</td>
</tr>
</tbody>
</table>

Service descriptions as well as their inputs and outputs are listed in Table 5.

Table 5 Implemented Publisher API definition

<table>
<thead>
<tr>
<th>Service Name</th>
<th>Input</th>
<th>Output</th>
<th>Info</th>
</tr>
</thead>
<tbody>
<tr>
<td>Publisher (constructor)</td>
<td>String name, String brokerIP, int brokerPort</td>
<td>Publisher</td>
<td>Used to create a Publisher entity. Requires as input the name of the entity in String format, IP address and port number of the broker. The output is the Publisher object which is used for publishing new data.</td>
</tr>
<tr>
<td>connect</td>
<td></td>
<td>void</td>
<td>A method used to connect to previously defined cloud broker.</td>
</tr>
<tr>
<td>disconnectFromBroker</td>
<td></td>
<td>void</td>
<td>A method used to disconnect from the cloud broker. All data (i.e. valid publications) remain in memory until the publisher object is destroyed or until their validity expires.</td>
</tr>
<tr>
<td>connect</td>
<td>String brokerIP, int brokerPort</td>
<td>void</td>
<td>A method used to connect to a specified cloud broker.</td>
</tr>
<tr>
<td>publish</td>
<td>Publication</td>
<td>void</td>
<td>Used to publish new data. The input parameter is a valid publication.</td>
</tr>
<tr>
<td>Unpublish</td>
<td>Publication publication</td>
<td>void</td>
<td></td>
</tr>
<tr>
<td>-----------</td>
<td>-------------------------</td>
<td>------</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Used to delete a previously published data item (publication). The input parameter is a publication which is removed from the cloud broker memory, but not from the subscribers that have already received the notification with specified data.</td>
<td></td>
</tr>
</tbody>
</table>

## 5.2.2 Download, Deploy & Run

### 5.2.2.1 Developer

#### 5.2.2.1.1 System requirements

All you need to build this project is Java 7.0 (Java SDK 1.7) or later, Maven 3.0 or later. The service of this project is designed to be run on an end-user device.

#### 5.2.2.1.2 Download

To download Publisher’s source code use your favourite git client and retrieve the code from one of the following URLs:

- HTTPS: https://github.com/OpenIotOrg/openiot.git
- SSH: git@github.com:OpenIotOrg/openiot.git

The publisher is available under the “openiot/modules/CUPUS” folder.

#### 5.2.2.1.3 Deploy from the source code

If you have not yet done so, you must Configure Maven before testing the publisher deployment. After that:

- **Build and Deploy the Publisher**
  
  o NOTE: The following build command assumes you have configured your Maven user settings. If you have not, you must include Maven setting arguments on the command line.

  1. Open a command line and navigate to the root directory of the CUPUS project.
  2. Type this command to build and deploy the archive:
     
        - mvn package

  3. This will build the service in the target folder and now the service is ready to be started by using the following command in terminal:

        java -jar target\StartPublisher.jar

        <path_to_config_file7> <path_to_publications_folder>

---

7 An example of the config file is given in the source/main/resource folder.
• Undeploy the Cloud Broker service
  1. Stop the running instance of the service by typing “QUIT”

5.2.2.1.4 Run in Eclipse

5.2.2.1.4.1 Deploying Publisher

To integrate and deploy the Publisher in Eclipse one should follow steps below:
  1. Import Existing maven project “File>Import>Maven>Existing Maven Projects”
  2. Click the “Browse” button and navigate to the CUPUS source code directory that has been previously downloaded.
  3. Click the Finish button.
  4. Open the org.openiot.cupus.examples package
  5. Right click on the “PublisherExample.java” choose “Run As>Java Application”

To undeploy the Publisher follow the steps below:
  1. Type “QUIT” or stop the system in the Console window

5.2.2.2 User

5.2.2.2.1 System requirements

All you need to run this project is Java 7.0 (Java SDK 1.7) or later. You can download the binaries through the OpenIoT Wiki\footnote{https://github.com/OpenIotOrg/openiot/wiki} under the Users>Downloads\footnote{https://github.com/OpenIotOrg/openiot/wiki/Downloads} section.

5.2.2.2.2 Deployment/Undeployment

\textbf{Deploy}: To deploy the Publisher service, copy the “Publisher.java” to the user device and run the terminal command: \texttt{java –jar StartPublisher.jar <path_to_config_file> <path_to_publications_folder>}. 

\textbf{Undeploy}: Stop the running instance of the service by typing “QUIT”.

5.3 Subscriber

As already stated, the OpenIoT Subscriber sends a subscription to the system, based on user interest. The Subscriber entity after the subscription process listens for incoming notifications that are of interest to the end-user.
5.3.1 Main Released Functionalities & Services

The current release of the OpenIoT Subscriber implements the functionalities/capabilities that are reflected in the Table 6.

Table 6 Subscriber public methods and constructor

<table>
<thead>
<tr>
<th>Method</th>
<th>Parameters</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Subscriber</td>
<td>name(String), brokerIP(String), brokerPort(int)</td>
<td>Constructor is used to create a Subscriber entity.</td>
</tr>
<tr>
<td>connect()</td>
<td></td>
<td>Method used to connect to a previously defined cloud broker.</td>
</tr>
<tr>
<td>disconnectFromBroker()</td>
<td></td>
<td>Method used to disconnect from the cloud broker. All data remain in memory</td>
</tr>
<tr>
<td>connect(brokerIP, brokerPort)</td>
<td></td>
<td>until the subscriber object is destroyed or until data validity expires.</td>
</tr>
<tr>
<td>subscribe(Subscription)</td>
<td></td>
<td>Method used to define a new subscription.</td>
</tr>
<tr>
<td>unsubscribe(Subscription)</td>
<td></td>
<td>Method used to remove a subscription.</td>
</tr>
<tr>
<td>setNotificationListener(notificationListener)</td>
<td></td>
<td>Method used to set the notification listener.</td>
</tr>
</tbody>
</table>

Service descriptions as well as their inputs and outputs are listed in Table 7.

Table 7 Implemented Subscriber API definition

<table>
<thead>
<tr>
<th>Service Name</th>
<th>Input</th>
<th>Output</th>
<th>Info</th>
</tr>
</thead>
<tbody>
<tr>
<td>Subscriber (constructor)</td>
<td>String name, String brokerIP, int brokerPort</td>
<td>Subscriber</td>
<td>Used to create a Subscriber entity. Requires as input the entity name in String format, IP address and port number of the broker. The output is the Subscriber object which is used for defining subscriptions and receiving incoming notifications.</td>
</tr>
<tr>
<td>connect</td>
<td></td>
<td>void</td>
<td>A method used to connect to a previously defined cloud broker.</td>
</tr>
<tr>
<td>disconnectFromBroker</td>
<td></td>
<td>void</td>
<td>A method used to disconnect from the cloud broker. All data (i.e. valid subscriptions and notifications) remain in memory until the subscriber object is destroyed or until data validity expires.</td>
</tr>
<tr>
<td>connect</td>
<td>String brokerIP, int brokerPort</td>
<td>void</td>
<td>A method used to connect to a specified cloud broker. Input parameters are broker IP address and port number.</td>
</tr>
<tr>
<td>subscribe</td>
<td>Subscription subscription</td>
<td>void</td>
<td>Used to define a new subscription. The input parameter is a valid subscription object.</td>
</tr>
<tr>
<td>Method</td>
<td>Description</td>
<td></td>
<td></td>
</tr>
<tr>
<td>---------------------</td>
<td>-----------------------------------------------------------------------------</td>
<td></td>
<td></td>
</tr>
<tr>
<td>unsubscribe</td>
<td>Used to delete a previously defined subscription. The input parameter is a valid subscription object.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>setNotificationListener</td>
<td>Used to set a processing class for notifications. Notifications are received as a response to user’s subscriptions.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

5.3.2 Download, Deploy & Run

5.3.2.1 Developer

5.3.2.1.1 System requirements

All you need to build this project is Java 7.0 (Java SDK 1.7) or later, Maven 3.0 or later. The service of this project is designed to be run on an end-user device.

5.3.2.1.2 Download

To download Subscriber’s source code use your favourite git client and retrieve the code from one of the following URLs:

- HTTPS: https://github.com/OpenIoTOrg/openiot.git
- SSH: git@github.com:OpenIoTOrg/openiot.git

The subscriber is available under the "openiot/modules/CUPUS" folder.

5.3.2.1.3 Deploy from the source code

If you have not yet done so, you must Configure Maven before testing the subscriber deployment. After that:

- Build and Deploy the Subscriber
  - NOTE: The following build command assumes you have configured your Maven user settings. If you have not, you must include Maven setting arguments on the command line.
  1. Open a command line and navigate to the root directory of the Subscriber project.
  2. Type this command to build and deploy the archive:
     - mvn package
  3. This will build the service in target folder and now service is ready to be started by command in terminal: java -jar
target\StartSubscriber.jar <path_to_config_file> <path_to_subscriptions_folder>

- Undeploy the Subscriber service
  1. Stop the running instance of the service by typing "QUIT"

5.3.2.1.4 Run in Eclipse

5.3.2.1.4.1 Deploying Subscriber

To integrate and deploy the Cloud Broker in Eclipse one should follow steps below:

1. Import Existing maven project “File>Import>Maven>Existing Maven Projects”
2. Click the “Browse” button and navigate to the CUPUS source code directory that has been previously downloaded.
3. Click the Finish button.
4. Open the org.openiot.cupus.examples package
5. Right click on the “SubscriberExample.java” choose “Run As>Java Application”

To undeploy the Subscriber follow the steps below:

1. Type “QUIT” or stop the system in the Console window

5.3.2.2 User

5.3.2.2.1 System requirements

All you need to run this project is Java 7.0 (Java SDK 1.7) or later. You can download the binaries through the OpenIoT Wiki under the Users>Downloads section.

5.3.2.2.2 Deployment/Undeployment

**Deploy**: To deploy the Subscriber service, copy the “Subscriber.java” to the user device. Run the terminal command: `java -jar StartSubscriber.jar <path_to_config_file> <path_to_subscriptions_folder>`.

**Undeploy**: Stop the running instance of the service by typing "QUIT".

---

10 An example of the config file is given in the source/main/resource folder.
11 https://github.com/OpenIoTOrg/openiot/wiki
12 https://github.com/OpenIoTOrg/openiot/wiki/Downloads
5.4 Source code examples

The following simple example specifies source code snippets which start a Cloud Broker, subscriber and publisher processes, and define subscriptions processed by the Cloud Broker as well as publications published by the created publisher process. The example does not assume that there are mobile broker processes interacting with the Cloud Broker.

5.4.1 CloudBroker

Table 8 Cloud broker source code snippet

```java
//define CloudBroker parameters
CloudBroker broker = new CloudBroker("Broker_example", 12345);

//start the broker
broker.start();
```

Table 8 shows a code snippet that creates the Cloud Broker object with a name Broker_example and the broker is listening for incoming messages on port number 12345. After starting the broker, you may create subscribers and define subscriptions. Next, publishers can connect to the broker and start publishing. The cloud broker will deliver matching publications to subscribers in near real-time.

5.4.2 Subscriber and subscribing process

Table 9 shows a code snippet that creates a Subscriber object whose parameters are specified in a configuration file. The configuration file contains the name of the subscriber object and an IP address and port of the broker to which it connects to. After the subscriber object is created, it is necessary to create a notification listener, i.e. a piece of the code that will process incoming notifications arriving from the broker. In this example the notification listener only prints out incoming notifications, but it can be extended to perform more complex processing tasks (a notification listener is defined in a separate class). After that the subscriber connects to the broker, defines three new subscriptions and subscribes. All three subscriptions have infinite validity. After subscriptions are defined, the process should loop to receive notifications until a specific condition is met (e.g., when a subscriber running flag is set to false). This is not shown in the snippet as it is application specific. In the end of the snippet the subscriber disconnects from a broker.
Table 9 Subscribing process source code snippet

```java
//create the Subscriber entity
Subscriber subscriber = new Subscriber(new File("./config/sub1.config"));

//create notification listener
subscriber.setNotificationListener(new NotificationListener()
{
@Override
public void notify(UUID subscriberId, String subscriberName, Publication publication)
    {
        HashtablePublication notification = (HashtablePublication) publication;
        HashMap<String, Object> receivedData = notification.getProperties();
        System.out.println("Received publication:");
        System.out.println(publication);
        System.out.println();
    }

public void notify(UUID subscriberId, String subscriberName, Subscription subscription)
    {
        //not used for the subscriber, just for the mobile broker
    }
});

//connect to the broker
subscriber.connect();

//define subscriptions
TripletTopKWSubscription ts1 = new TripletTopKWSubscription(3, 1000*60, -1,
        System.currentTimeMillis());
    ts1.addTriplet(new Triplet("ID", 5, Operator.EQUAL));

TripletSubscription ts2 = new TripletSubscription(-1,
        System.currentTimeMillis());
    ts2.addPredicate(new Triplet("num2", 7.297, Operator.LESS_THAN));

TripletSubscription ts3 = new TripletSubscription(-1,
        System.currentTimeMillis());
    ts3.addPredicate(new Triplet("str", "cAt", Operator.CONTAINS_STRING));

//send subscriptions to broker;
subscriber.subscribe(ts1);
subscriber.subscribe(ts2);
subscriber.subscribe(ts3);

//disconnect from broker
subscriber.disconnectFromBroker();
```
5.4.3 Publisher and publishing process

Table 10 shows a code snippet that creates the Publisher object by using a configuration file. The configuration file contains the name of the publisher, and an IP address and port of the broker to which the publisher connects to. After the publisher is created, a new publication is defined. The created publication can either have infinite or limited validity time. Next, two properties are added to the publication. Finally the publisher sends the publication to the broker and disconnects from it.

Table 10 Publishing process source code snippet

```java
Publisher publisher = new Publisher (new File("./config/pub1.config"));
//connect to broker
publisher.connect();
HashtablePublication hp = null;
//define new publication
hp = new HashtablePublication(-1, System.currentTimeMillis());
//alternatively validity time can be set as: "hp.getStartTime() + [offset in ms]"
//hp.setValidity(hp.getStartTime() + 1000000);
//set publication properties as name-value pairs;
//each publication can contain an arbitrary number of properties
hp.setProperty("num1", num1);
hp.setProperty("num2", num2);
//publish the publication
publisher.publish(hp);
//disconnect from broker
publisher.disconnectFromBroker();
```

In case publication parameters are defined so as to match subscriber subscriptions, they should be printed out on the subscriber process console. In the end when both the publisher and subscriber processes disconnect from the broker, only the Cloud Broker process remains running, while both the subscriber and publisher processes finish.
6 EXPERIMENTAL RESULTS

In this section we evaluate the processing performance of the CPSP engine in a cloud environment 1) to evaluate its scalability when processing an increasing number of subscriptions and 2) to investigate implementation elasticity when varying the processing load.

To measure the processing performance of the CPSP engine we have used the real world environmental data set collected within the OpenSense project\(^\text{13}\). The data set contains 4,000 sensor readings for the following gases: carbon-monoxide, ozone, nitrogen dioxide and ultrafine-particles, collected from at least 30 different sensors in the city of Zürich. Publications generated during the experiment are generated by selecting random data measurement from the OpenSense dataset. As we do not have real user queries, subscriptions are generated by choosing a random measurement from the OpenSense set and associating it with a random numerical operator (excluding BETWEEN). Additionally, we generate subscriptions related to specific time periods regardless of sensor type such that we define constraints on timestamps. Such timestamp subscriptions are more general and do not overlap each other, which creates more independent trees within a subscription forest and thus increases the processing time at the cloud broker. We assume that the generated subscription set is a good approximation of real subscriptions because users typically have similar interests: e.g., a sensor reading for a specific gas on a busy location is of interest to many users, while many subscriptions may be of interest to just a few users, for example all sensor readings for a specific time period.

Experiments investigating the processing performance of the CPSP engine are done using two different virtual machine instances running on a Dell PowerEdge R720 rack server with the Citrix XenServer virtualization software. The first instance uses 2 vCPU cores with 4 GB of RAM (VM-2), while the second instance has 4 vCPU cores with 8 GB of RAM (VM-4). Each virtual instance uses Debian as the operating system with Java7 installed.

Two different matcher configurations are tested on VM-2 and VM-4. The first configuration starts a single matcher component such that all subscriptions are stored in a single forest data structure \((\text{config}_1)\). The second configuration consists of five matcher components which evenly divide all subscriptions \((\text{config}_5)\), each forming its own forest structure. The experiment starts by defining the subscriptions and building the forest structure, and afterwards the CPSP engine starts receiving publications. Incoming publications are forwarded in a round robin fashion to all running matchers in case of \(\text{config}_5\).

We investigate the data propagation delay in the following experiments since it comprises the time needed to send a publication to the CPSP engine, process the publication by the CPSP engine and deliver it to a subscriber. Note that publishers and subscribers in this set of experiments are running on another virtual machine of the rack server to generate processing load for the CPSP engine. Publications are sent sequentially without any delays causing a constant load on the CPSP engine in experiments 1 and 2, while for experiment number 3 we generate changing processing load to investigate elasticity of our solution.

\(^{13}\) http://opensense.epfl.ch
Figure 21 Data propagation delay when increasing the number of subscriptions under constant publication rate

**Experiment 1: Scalability.** To test the scalability of the CPSP engine in the cloud, we varied the number of subscriptions from 1 to 20,000.

Figure 21 depicts the average data propagation delay incurred by the CUPUS middleware with different matcher configurations (config1 and config5). Dashed lines represent the performance of the CUPUS middleware on VM-2, while the solid line represents results obtained on VM-4. In our tests the best performance is obtained with config5 running on VM-4. It is interesting to notice that an increase in the available resources did not result in significant improvement of the average data propagation delay when we use config1, but we obtain significantly improved results with config5 running on a VM-4. As expected, the worst results are obtained when config5 runs on VM-2: We notice performance degradation of the CPSP engine when a large number of spawned processes are running in parallel on only 2 vCPU cores. It can be concluded that CUPUS scales well when adequate resources are assigned to specific configurations, with the best-performing average propagation delay up to 1 second, even when 20,000 subscriptions are processed simultaneously.
Experiment 2: Stability under constant load. Figure 22 depicts the average propagation delay when increasing the number of publications generated during a single test run. It investigates the stability of our solution as the size of the dataset should not affect the processing performance and propagation delay. As expected, the propagation delay does not depend on the number of publications sent to the CPSP engine during a single test run under constant processing load.

Experiment 3: Elasticity. This experiment evaluates the CPSP engine performance with dynamic configurations to inspect solution elasticity, i.e., whether it adapts well to increasing/decreasing processing load. The CPSP engine is running on VM-4 machine while the subscriber and publisher are located on VM-2. We use a setup with 5,000 subscriptions and 50,000 publications in this experiment and investigate the data propagation delay while varying the publication frequency from 95 publications per second to 1000 publications per second. We also monitor matcher configurations which are dynamically being changed by the CPSP engine (i.e., the number of active matchers and requests for splitting or merging of matchers). Splitting events are triggered by high processing load on a matcher when a new matcher is created to take over part of subscriptions from an overloaded matcher. Merging events are triggered on under-loaded matchers which are to be released while other matchers take over their subscriptions.

Similar to previous configurations all subscriptions are static and defined previous to the experimental run.
Figure 23 The data propagation delay depending on the incoming publication frequency with dynamic CUPUS configuration

Figure 23 depicts the data propagation delay depending on different publishing frequencies. Our tests show that under high publishing frequency the CPSP engine performance with dynamic configuration has a higher propagation delay compared to the fixed configuration. This occurs mostly because of the maintenance job to spawn or destroy matchers, but possible benefits from that approach outweigh the shortcomings. With a dynamic configuration, a minimal number of matcher processes is used, thus allowing significant resource savings during the period of low load. We can notice that the data propagation delay drops from a high level at the beginning of the experiment to lower levels later on during the experiment because the engine autonomously reconfigures to use a minimal number of matcher processes with respect to the processing load. Sudden increases of the propagation delay are mostly connected to the elasticity requests (i.e., splitting or merging requests from a matcher) which is clearly visible in the experiment with the rate of 420 publications per second.

Figure 24 shows the CPSP engine configurations during our experiment with varying publishing load. Figure 24(a) depicts the number of matchers running in the case of very high load which changes frequently during the experiment since the engine constantly attempts to balance the number of matchers and matcher processing load. This is the main reasons for the observed significant increase of the data propagation delay. Figure 24(b) presents the engine configuration for two medium publishing loads. We can observe strong correlation between propagation delays and configuration changes of the CUPUS middleware, which is especially visible in the experiment with publication rate of 420 publications per second. After the initial reconfiguration of the engine, due to high load on a single matcher, the splitting event is triggered causing sudden high propagation delay. It is interesting to observe the configuration changes in case of 292 publications per second although the number of used matcher processes is constantly one. One can notice that this graph does not contain any changes which represent the splitting or merging event, concluding that the processing load of the matcher is below the threshold for creating a new matcher.
The experiment with low publishing load depicted in Figure 24(c) started with two matchers running in parallel, but the middleware rearranged its configuration to use only one matcher since it was sufficient to process all incoming messages. In the experiment with very low load depicted in Figure 24(d) we can see a lot of changes. In this case these are unsuccessful merge requests occurring because the matcher process is constantly under-loaded.

To conclude, the experiments show that the CPSP engine tries to use optimally the available computing resources and creates additional matchers under high load, or releases existing matchers when they are under-loaded while reconfigurations may cause higher data propagation delays.
7 RELATED DATA STREAM PROCESSING PLATFORMS

There are number of data stream processing platforms and publish/subscribe middleware solutions comparable to CUPUS. Note that publish/subscribe solutions are in general seen as distributed messaging systems supporting simple topic-based subscriptions, while CUPUS supports a rich subscription language including top-k/w processing, enables elastic processing of data streams in the cloud and on mobile devices, and also pushes matching data objects from the cloud to mobile devices. In comparison to publish/subscribe middleware, data stream processors support a rich set of operators (for example Map, Union, Aggregate, Join) and are optimized to process high-frequency data streams. However, we are not aware of data stream processing solutions that allow for flexible and aggregate stream pre-processing on mobile devices, and they require additional software components for pushing data objects to mobile devices. Hereafter we review a selection of prominent and recent solutions related to CUPUS.

Aurora

Aurora [Aurora02] is one of the first platforms that allowed parallel processing of data streams. It was originally designed as an application monitoring system. Continuous querying is enabled through a graphical interface by defining a data stream diagram between the nodes that form a query.

Input data processing is done at two tiers, the first tier determines which persistent query is affected by the input data, and on the second tier input data is assigned to the corresponding nodes which later on perform processing in accordance with the defined operator.

The discarding of input data is supported in case the system capacity is lower than the intensity of the incoming data. Random and semantic data dropping is also supported. Aurora supports seven types of queries which include the input stream data filtering, creation of new data stream from one or more input streams, sorting part of the data stream, or aggregation of the input data stream.

StreamCloud

StreamCloud [StreamCloud10] is a cloud-based data stream processing platform which implements the filtering operators, unions and aggregate functions over data streams.

StreamCloud is an intermediate layer that parallelizes user query into subqueries that are executed on different nodes. Currently, a distributed version of the data processing system Borealis is used for processing parallel queries. Parallelization of user queries is independent of the data stream processing system, so Borealis can be replaced with any other system.

BlueDove

BlueDove [BlueDove11] is an elastic and scalable content-based publish-subscribe service running in the cloud. It is based on a two-tier architecture of dispatcher and matcher components, and uses a special subscription space partitioning technique that enables it to have smart redundancy for server fault tolerance, and to exploit the skewness in the data distribution.
The dispatchers have public interfaces so that publishers and subscribers can connect to them, and send them publications and subscriptions. Every dispatcher is connected to all the matchers, and its only job is to forward received publication and subscription requests to the appropriate matchers. The appropriate matchers are found using a simple one-hop look-up, which enables dispatchers to be very lightweight and have very high throughput.

**Kinesis**
Kinesis [Kinesis] is an Amazon product and a part of the Amazon Web Services computing platform. It is a fully managed service for real-time processing of streaming data at any scale. It can accept any amount of data, from any number of sources, scaling up and down as needed. Unlike other real-time stream processing solution, the Amazon’s model is more convenient for those who would rather invest in development work than in any infrastructure. Kinesis requires that a user creates at least two applications, a Producer and a Worker. The Producer takes the data from a data source and converts it into a Kinesis Stream, a continuous flow of data pieces sent in the form of HTTP PUTs. The Worker takes the data from the Kinesis Stream and does the required data processing.

**InfoSphere**
IBM InfoSphere [InfoSphere] Streams is an advanced computing platform that allows customers to quickly ingest, analyse and correlate massive volumes of continuous data streams. It can help customers turn burgeoning data volumes into actionable information and business insights. It delivers a highly scalable, agile software infrastructure that enables businesses to perform in-motion analytics on a wide variety of structured and unstructured data types at massive volumes and speeds, which enables real-time analytic processing.

**Storm**
Storm [Storm] is a free and open source distributed fault-tolerant and real-time computational system. It abstracts the inherent complexities of Queue/Workers system which allows users to write real-time topologies without needing to worry about scaling, implementing fail-over or inter-process communication. A Storm topology is analogous to a MapReduce job, with key difference that MapReduce job eventually finishes, while a topology runs forever.

A topology is a graph of spouts and bolts that are connected with stream groupings. Stream is an unbounded sequence of tuples that is processed and created in parallel in a distributed fashion. A spout is a source of streams, and it will read tuples from an external source and emit them into the topology. Bolts are responsible for all processing in topologies, while stream grouping define how that stream should be partitioned between the bolt’s tasks. Each spout or bolt executes as many tasks across the cluster, and each task is equivalent to one thread of execution. Worker is a process that runs the topology, and there is one or more worker processes.
Apache S4
Apache S4 [ApacheS4] is a general-purpose, near real-time platform for processing continuous unbounded streams of data. It provides a runtime distributed platform that handles communication, scheduling and distribution across containers. It fills the gap between complex proprietary systems and batch-oriented open source computing platforms, and it hides the complexity inherent in parallel processing system. S4 applications developed by users are deployed on S4 clusters, and those applications are built as a graph containing processing elements (PEs) and streams that interconnect them.

Processing elements communicate asynchronously by sending events on streams, and events are dispatched to nodes, which represent distributed containers. Dynamic and loose coupling of S4 applications is achieved through a pub-sub mechanism.

STREAM
STREAM [STREAM06] is a relational Data Stream Management System that supports continuous queries specified in a rich declarative language called CQL. Each CQL text query generates a physical query plan that runs continuously. Query plan merging can occur often, so a single query plan may compute multiple continuous queries.

STREAM query operator can be CQL operator or a system operator, and every CQL operator has one of the three types: stream-to-relation, relation-to-relation, or relation-to-stream. STREAM supports an interactive graphical interface for visualizing run-time plan and system behaviour. Using this interface, users can visualize and control plan adaptively.
8 CONCLUSIONS

Deliverable D4.5.2 corresponds to the final official release of the open source implementation of the OpenIoT publish/subscribe solution for efficient processing of sensor data streams within the cloud environment to identify relevant data objects and deliver them in near real-time to largely distributed data consumers, e.g., user mobile phones. This middleware is named CloUd-based PUblish/Subcribe for the Internet of Things (CUPUS). The release comprises a prototype implementation of the CUPUS processing engine named CPSP engine tailored to cloud environments, along with the accompanying documentation which is provided as part of this document. Note that the open source implementation of the CPSP engine is available within the Github infrastructure of the project (https://github.com/OpenIoTOrg/openiot).

This deliverable presents the underlying publish/subscribe model and cloud-based architecture of CUPUS. Moreover, it includes a thorough technical description of the CPSP engine which supports both Boolean and top-k/w subscriptions and includes an explanation of CUPUS interaction with the rest of the OpenIoT platform. We also include an API specification and code examples, and provide experimental results which investigate CPSP engine performance under high and varying processing load.
9 REFERENCES


[Storm] http://storm-project.net/


Deliverable 4.5.2 Elastic Publish/Subscribe Processing Engine for Sensor Data Streams


OpenIoT 2014