Ref. Ares(2014)44358 - 10/01/2014

SEVENTH FRAMEWORK PROGRAMME

Specific Targeted Research Project

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

D4.5.1 Elastic Publish/Subscribe Processing Engine for Sensor Data Streams a

Document Id:	OpenIoT-D451-131220-Draft
File Name:	OpenIoT-D451-131220-Draft.pdf
Document reference:	Deliverable 4.5.1
Version:	Draft
Editor:	Aleksandar Antonić, Ivana Podnar Žarko
Organisation:	UNIZG-FER
Date:	2013 / 12 / 20
Document type:	Deliverable
Security:	PU (Public)

Copyright © 2013 OpenIoT Consortium: NUIG-National University of Ireland Galway, Ireland; EPFL - Ecole Polythechnique 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; UniZ-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

Rev.	Author(s)	Organisation(s)	Date	Comments
V01	Aleksandar Antonić	UNIZG-FER	2013/06/28	Initial ToC
V02	Ivana Podnar Žarko	UNIZG-FER	2013/11/18	Added initial version of Section 2
V03	Aleksandar Antonić	UNIZG-FER	2013/11/20	Input to Section 3
V04	Ivana Podnar Žarko	UNIZG-FER	2013/11/20	Modified Section 2.2
V05	Kristijan Rožanković	UNIZG-FER	2013/11/21	Input to Section 6
V06	Aleksandar Antonić	UNIZG-FER	2013/11/21	Input to Section 5
V07	Krešimir Pripužić	UNIZG-FER	2013/11/21	Input to Section 3 & 4
V08	Kristijan Rožanković	UNIZG-FER	2013/11/22	Modified Section 6
V09	Ivana Podnar Žarko	UNIZG-FER	2013/11/22	Modified Section 2.3
V10	Aleksandar Antonić	UNIZG-FER	2013/11/24	Modified section 3
V11	Ivana Podnar Žarko	UNIZG-FER	2013/11/25	Added new Section 3 on architecture and rearranged section 4 Final modifications to Section 2
V12	Krešimir Pripužić	UNIZG-FER	2013/11/26	Final modifications to Section 3
V13	Aleksandar Antonić	UNIZG-FER	2013/11/26	Modified Section 4 & 5
V14	Ivana Podnar Žarko	UNIZG-FER	2013/11/25	Added Section 1, final modifications to section 3 to 5, added Section 6
V15	Ivana Podnar Žarko	UNIZG-FER	2013/11/27	Modifications to Section 1 and conclusion
V16	Nikos Kefalakis	AIT	2013/12/09	Technical Review with Comments
V17	Johannes Mulder	FHG-IOSB	2013/12/15	Quality Reviews
	Ivana Podnar Žarko	UNIZG-FER	2013/12/16	Answers to TR and QR comments
V18	Martin Serrano	DERI	2013/12/17	Circulated for Approval
V19	Martin Serrano	DERI	2013/12/20	Approved
Draft	Martin Serrano	DERI	2013/12/23	EC Submitted

Copyright © 2013 OpenIoT Consortium

TABLE OF CONTENTS

Copyright © 2013 OpenIoT Consortium

Copyright © 2013 OpenIoT Consortium

LIST OF FIGURES

LIST OF TABLES

Copyright © 2013 OpenIoT Consortium

TERMS AND ACRONYMS

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

Copyright © 2013 OpenIoT Consortium

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 it also enables 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 be adaptive to the continuous evolution of the design- and implementation-process of the IoT data systems and particularly to adapt the OpenIoT Architecture to mobile sensors. OpenIoT additional functionality related with 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.** Thus this deliverable focuses on designing and planning the implementation for those identified additional features in the OpenIoT architecture.

Since the streaming rate of 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 D4.5.1 presents a publish/subscribe cloud for the Internet of Things entitled **CUPUS: CloUd-based PUblish/Subscribe for the Internet of Things** which is a new OpenIoT platform component/module. This is the first version of this deliverable, which includes the design of the open source prototype and implementation of CUPUS available at the Github infrastructure of the OpenIoT project (https://github.com/OpenIotOrg/openiot). The current prototype supports *content-based publish/subscribe*, i.e., stateless Boolean subscriptions. It is designed to enable pre-filtering of sensor data streams close to data sources, e.g., on mobile devices, such 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 D3.4.1: Publish/subscribe middleware for mobile internet-connected objects. The CUPUS architecture is designed to be elastic such that it can dynamically allocate more resources within the cloud or release underused resources in accordance with the processing load. The first version of the prototype supports a static flat architecture with concurrent processes that share the processing load, as instructed by a central coordinator.

Copyright © 2013 OpenIoT Consortium

The second release of this deliverable will include an extended prototype supporting top-k subscriptions over sliding windows that are stateful. Moreover, it will support dynamic allocation of processing resources for elastic publish/subscribe processing. The second CUPUS release will be fully integrated with the rest of the OpenIoT platform.

1.2 UNIZ-FER contribution to the OpenIoT platform

CUPUS is a novel component comprising 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 adequate 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 3.4.1 focuses on data acquisition and delivery in mobile environment.

In the second version of the OpenIoT platform release CUPUS will be fully integrated with the rest of the OpenIoT platform. It can be regarded as a special “high-speed” information bus for mobile environments which also integrates mobile ICOs with the rest of the OpenIoT platform. In other words, CUPUS will enable scheduling of OpenIoT requests on mobile ICOs and will store 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**, notably members of the project that intend to engage 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 middleware for integrating IoT applications, notably applications that adopt a cloud/utility-based model. 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.

Copyright © 2013 OpenIoT Consortium

- **IoT researchers at large,** who could find in this deliverable 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 is 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 preliminary 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.

Copyright © 2013 OpenIoT Consortium

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 proc*ess 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 where either pedestrians or cyclists use an environmental or traffic monitoring application which provides them with notifications affecting their movement throughout the city, one can conclude that the time period from sensor data acquisition until notification delivery to mobile devices needs to be extremely short. This is indeed not a trivial task since during that period of time the acquired data needs to be enriched, filtered and processed such that in the end of this process only information of interest is served to 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, i.e. *publishers*, to data destinations, i.e. *subscribers*. Data filtering and distribution is 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 publishers *publish* content (i.e. data objects) which they want to distribute among subscribers in 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., processed running on mobile devices, and maintain a list of publications for temporarily disconnected subscribers.

Copyright © 2013 OpenIoT Consortium

Brokers are the main building blocks of a publish/subscribe system. As they can be replicated and distributed to divide the matching load, they enable 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. The communication between publishers and subscribers is persistent and asynchronous. This enables a subscriber to be disconnected while remaining registered, which means that messages matching the subscriber's subscriptions will be saved and delivered to its mobile device as soon as it reconnects to the system.

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 [Pripuzic2013]. For example, in an environmental monitoring application, environmental scientists can with top-k/w subscriptions 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 in the fact 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 a 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 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. subscriber may be overloaded with publications,
3. it is impossible to compare different matching publications with respect to a subscription as ranking functions are not defined, and
4. partial matching between subscriptions and publications is not supported.

Copyright © 2013 OpenIoT Consortium

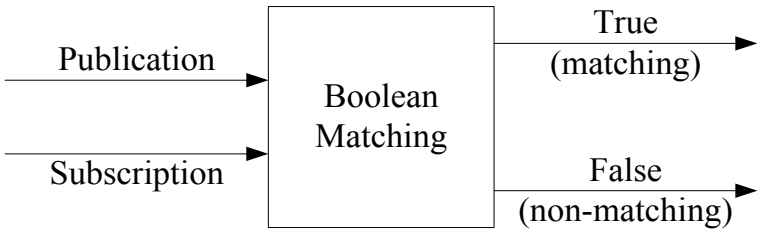

Figure 1 Matching in Boolean publish/subscribe systems

As 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 contrary, in case of an *overspecified* subscription, the subscriber may receive too few publications or none in the worst case. Thus, a subscriber has to specify an "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 will be considered as a sort of 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 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 $k \in \mathbf{N}$ and $w \in \mathbf{R}^+$. 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 top-k publications that are 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*) [MouratidisP07].

The quantity 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 in practice means that the matching process comparing a newly published publication to a subscription depends both on publication score 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.

Copyright © 2013 OpenIoT Consortium

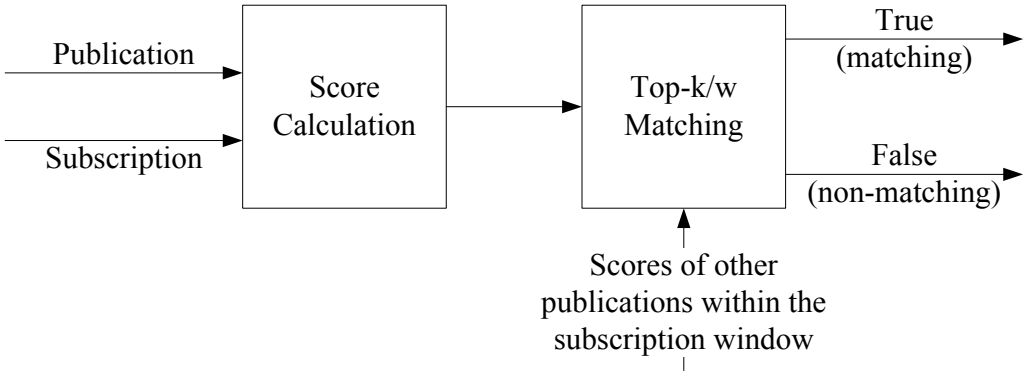

Figure 2 Matching in top-k/w publish/subscribe systems

The reason why the implementation of top-k/w processing is not trivial for big data streams is the following: 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 either when a new publication enters the system when there are less than *k* higher ranked publications within the subscription window, or at a later point in time when one of top-k publications is dropped from the subscription window, and 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.

2.2 Publish/subscribe model

In this section we give formal specifications of two different publish/subscribe systems using 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. On the contrary, the *top-k/w publish/subscribe system* relies on a novel matching model, named the *top-k/w matching model* which 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 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 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 $p \in \mathbf{P}$ may publish or cancel publications from **M**, while a subscriber $s \in \mathbf{S}$ may activate or cancel subscriptions from **N**. A broker $b \in \mathbf{B}$ 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*.

Copyright © 2013 OpenIoT Consortium

Publication. A publication $m \in \mathbf{M}$ is defined as some content, e.g. a sensor reading or a tweet, which is published by publisher $p \in \mathbf{P}$ at a point of time t, while a broker $b \in \mathbf{B}$ responsible for processing m receives it at a point in time t^A. We say that the *time of appearance* t^A denotes a point in time when publication p 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 such 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 such 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 $n \in \mathbf{N}$ is defined as a time independent *Boolean matching function* over **M** which is activated by subscriber $s \in \mathbf{S}$ at a point of time t, while a broker $b \in \mathbf{B}$ responsible for processing n receives it at a point in time t^A. We say that the *time of appearance* t^A 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 it 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 lets out publications marked with true.

Matching Publications. The set of publications matching a Boolean subscription $n \in \mathbf{N}$ 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=**(**B**,**P**,**S**,**M**,**N**) if for each Boolean subscription $n \in \mathbf{N}$, the system delivers all matching publications to subscriber s issuing subscription n. Each matching publication should be delivered to subscriber s exactly once and immediately after being matched by a broker that is responsible for processing it, while the system does not deliver non-matching publications to 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.

Copyright © 2013 OpenIoT Consortium

Note that a publisher *p* which has defined publication *m* with time of appearance t^A may decide to cancel *m* by sending a message to its broker *b* that receives such a message at a point in time $t^C > t^A$. Broker *b* can from the point in time t^C 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^A 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 $t^C > t^A$. All incoming publications after t^A 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 similarly to the Boolean system as a quintuple **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 $p \in \mathbf{P}$ may publish or cancel publications from **M**, while a subscriber $s \in \mathbf{S}$ may activate or cancel subscriptions from **N**. A broker $b \in \mathbf{B}$ 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*. However, top-k/w subscriptions from **N** are quite different from Boolean subscriptions.

Top-k/w Subscription. A top-k/w subscription $n \in \mathbf{N}$ is also activated by subscriber $s \in \mathbf{S}$ at a point of time *t*, while a broker $b \in \mathbf{B}$ responsible for processing *n* receives it at a point in time t^A. 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 $u_s: \mathbf{M} \rightarrow \mathbf{R}$,

2) parameter $w \in \mathbf{R}^+$ which defines the size of subscription window (time-based window) and

3) parameter $k \in \mathbf{N}$ 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^A 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 period $(t^A, t^C]$.

Scoring functions are application specific and their explicit definition depends completely on the application scenario in which the top-k/w system is used. Similarly to Boolean matching functions, we assume that 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 such that publications can be ranked consistently because a tie-breaking criterion can give preference to a more recent object. Scoring functions

Copyright © 2013 OpenIoT Consortium

may assign ranks to publications either in descending or ascending order of 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 ranks may change over time as new publications appear in the system, while previously appeared publications are dropped from a subscription window.

A publication is 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 *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 $n \in \mathbf{N}$, 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 [Pripuzic2013].

2.3 Self-organizing characteristics: scalability and elasticity

The main strengths of the publish/subscribe communication paradigm are loose coupling of communicating components and system scalability. Loose coupling of system components is achieved by anonymous communication since the processes, i.e., publishers and 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

Copyright © 2013 OpenIoT Consortium

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 different kinds of large 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 we lack efficient solutions targeting cloud-based environments.

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 in the throughput, slowdowns, and IP broadcast storms may appear. The use of the publish/subscribe architecture 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 coupled with the publication rate since in case there are no publications to be processed, there is no need to perform the matching process, even in 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 such that it adapts well to current workload. In other words, it should be able to process many subscriptions in parallel such that the matching overhead per publication is minimized. 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 parallelisation architectures for cloud-based publish/subscribe:

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

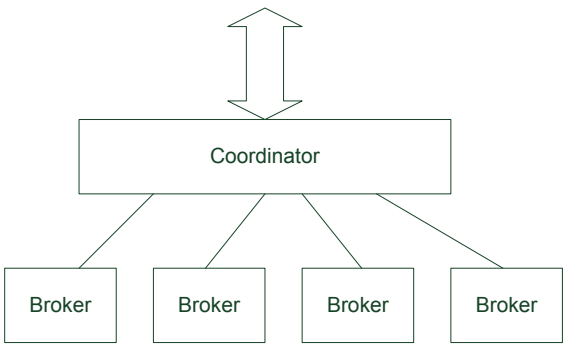

Figure 3 Flat cloud broker architecture

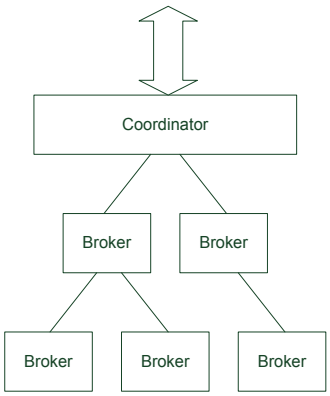

Figure 4 Hierarchical cloud broker architecture

Copyright © 2013 OpenIoT Consortium

The flat architecture with independent brokers relies on a central coordinator to divide the processing 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 easily be used for both Boolean and top-k/w architecture.

Hierarchical broker architecture organises broker matchers into a hierarchy. This is possible due to the inherent property of Boolean subscriptions, *subscription coverage*, which can be used to organize subscriptions into a forest. A Boolean subscription s_1 is *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. Subscription forest is used to improve the broker matching performance since a matching algorithm based on *subscription covering forest* 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 may easily be mapped onto a hierarchical broker architecture where each broker is responsible for different parts of the subscription forest. It is at this point not obvious how to map top-k/w subscriptions to a hierarchical broker architecture.

Both the flat and hierarchical architectures can be used in our cloud-based publish/subscribe implementation as it is further discussed in Section 4.1.

Copyright © 2013 OpenIoT Consortium

3 PUBLISH/SUBSCRIBE SYSTEM ARCHITECTURE

The *Cloud-based Publish/Subscribe Processing Engine* (CPSP Engine) developed within this deliverable will support both Boolean and top-k/w subscriptions as defined in Section 2.2. Moreover, the CPSP engine is designed to be elastic, as discussed in Section 2.3. The CPSP engine that is delivered and published as open source in the first version of deliverable D4.5, i.e., in D4.5.1, supports content-based Boolean matching and currently runs a single broker instance. Similar to [Sadoghi2011], we support a rich predicate language with an expressive set of operators for the most common data types: relational operators, set operators, prefix and suffix operators on strings, and the SQL BETWEEN operator.

The final version of deliverable D4.5 (D4.5.2) will include support for top-k/w subscriptions and will support dynamic allocation of cloud resources in accordance with the cloud processing load.

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 or revoke a previously published publication (methods *publish* and *unpublish*). Subscribers can define a new subscription or cancel an existing subscription (methods *subscribe* and *unsubscribe*). The CPSP engine notifies a subscriber with a publication matching at least one of its active subscriptions (method *notify*).

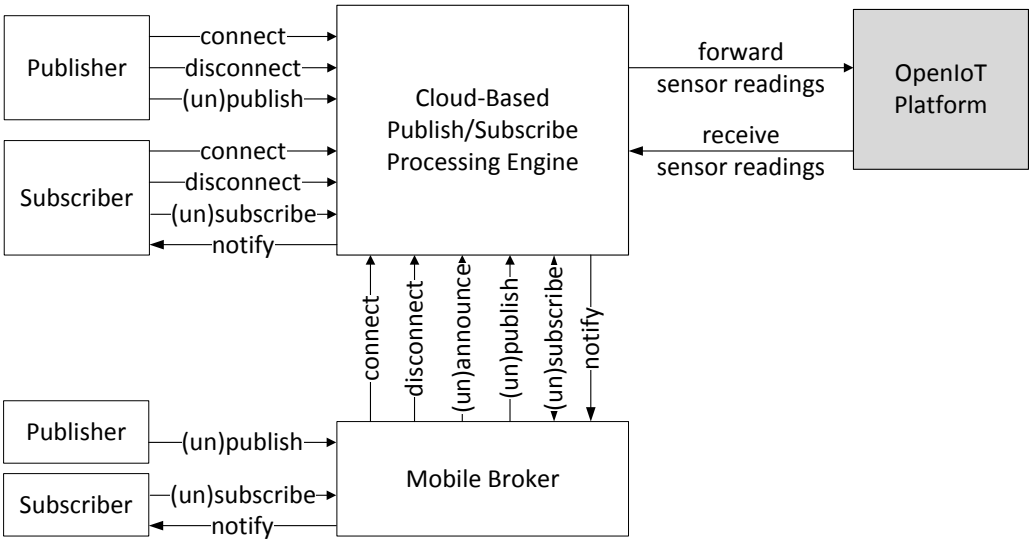

Figure 5 Architecture of the publish/subscribe system

Copyright © 2013 OpenIoT Consortium

In Figure **5**, we can see that the CPSP engine 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 batteries of both sensors and end-user devices, which would otherwise be drained by completely 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 subscribers. Hereafter we describe the mechanism and required methods to activate appropriate subscriptions on mobile devices.

Since a mobile broker serves as a proxy for 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 their local subscribers. Sensor discovery is enabled by a special announce message which is used to inform 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 the local publisher. This way, the CPSP engine knows about all available sensors in the system and can 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 it 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 an OpenIoT platform user searches for air quality sensors within a defined geographical area and wants to review when the readings from such sensors are above a certain threshold, the OpenIoT platform can define new subscriptions specifying interest in e.g. CO and VOC readings above certain thresholds. These subscriptions are relayed through the CPSP engine to mobile devices attached to CO and VOC sensors which are residing within the predefined area. The CO and VOC sensors can be invoked to start periodic readings, if necessary, and the mobile device checks whether each received CO or VOC readings is above the predefined threshold. A detailed description of this procedure is available in Deliverable 3.4.1.

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

Copyright © 2013 OpenIoT Consortium

sensor data acquired through mobile devices in order to provide such readings for 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.

In the rest of this section we provide detailed description of interactions between publish/subscribe system components.

3.1 Sequence diagrams

Connect and disconnect. The two methods are used by subscribers, publishers and mobile brokers. 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 reconnect to the CPSP engine, the engine first delivers all publications that have been matched to their active subscriptions while they were disconnected.

Publish. The CPSP engine stores all received subscriptions through *subscribe* requests to a list of active subscriptions. A new *publish* request defines a new publication which is subsequently matched to the list of stored active subscriptions. The matching process identifies the list of subscribers with matching subscriptions, and such subscribers are notified about this publication. Since publications can define their validity periods there is no need for a special unpublish request. The CPSP engine also needs maintains a list of valid publications. A sequence diagram depicting the delivery of a new publication to an interested subscriber is shown in Figure **6**.

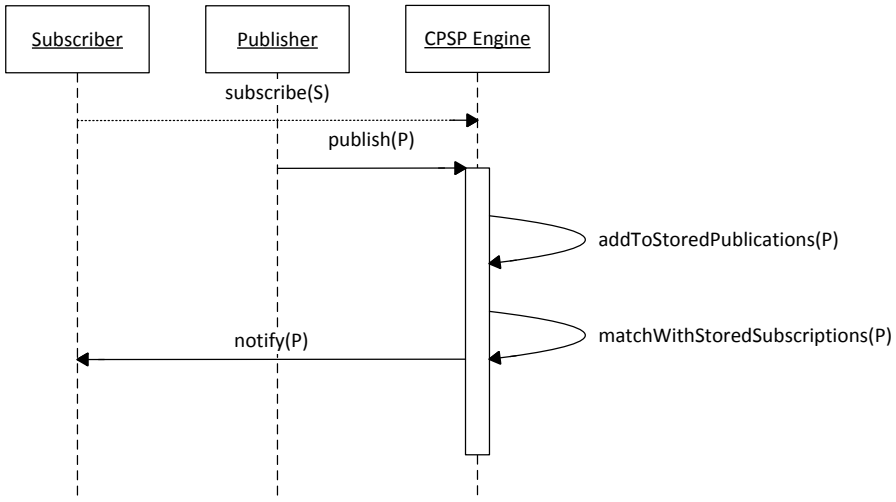

Figure 6 Delivery of a new publication

Announce. Figure **7** shows the sequence of events following a new *announce* event. When a mobile broker 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

Copyright © 2013 OpenIoT Consortium

subscriptions matching the announcement, it may activate the publisher by forwarding matching subscriptions to the mobile broker in the corresponding subscribe message.

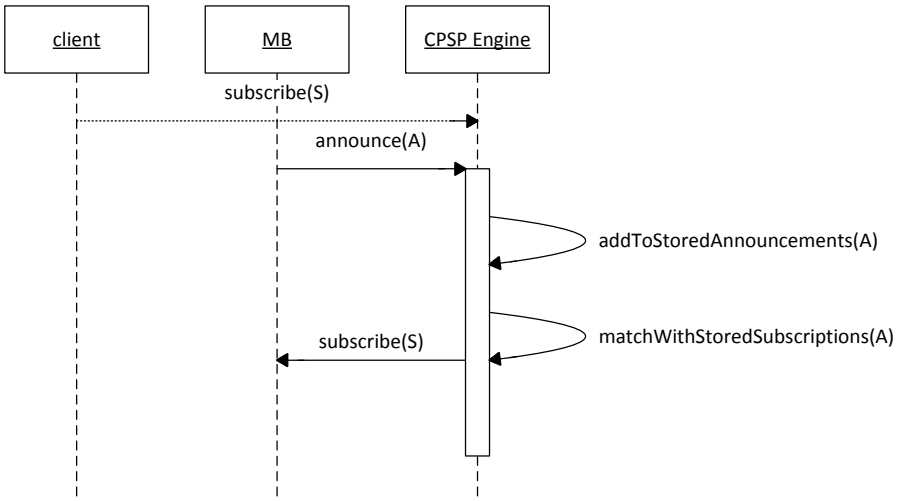

Figure 7 Announcing a new publisher

Unannounce. 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.

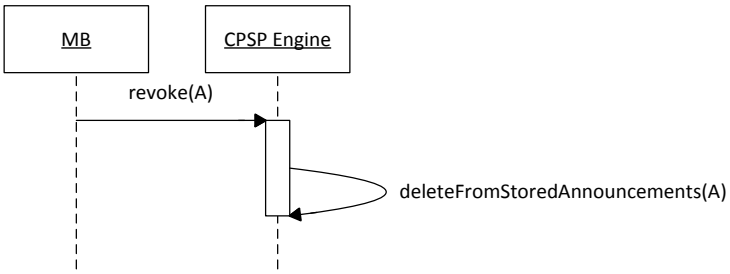

Figure 8 Revoking an announcement

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 with 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

Copyright © 2013 OpenIoT Consortium

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.

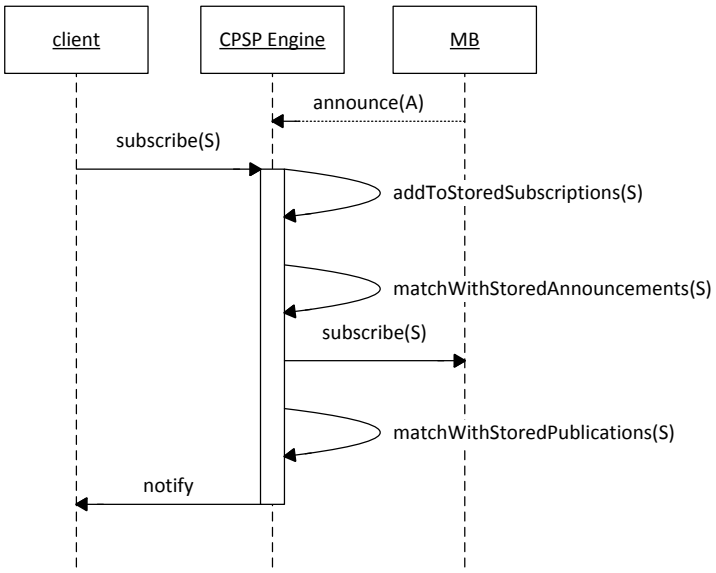

Figure 9 Activating a new subscription

Unsubscribe. Clients usually unsubscribe when there are no longer interested in particular publications. When the CPSP engine receives an *unsubscribe* message, it needs to delete the subscription from the list of stored subscriptions. Additionally, if such a cancelled subscription is the last one which is interested in publications of a specific publisher (connected to a mobile broker), it will be deactivated (i.e. instructed to stop producing new publications) by forwarding an unsubscription message, as shown in Figure **10**.

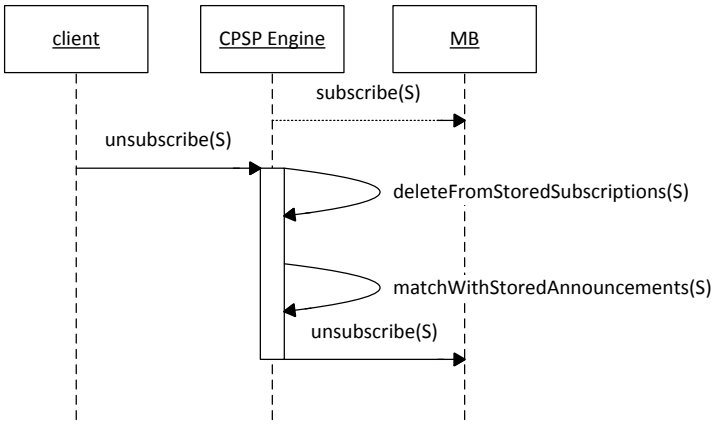

Figure 10 Cancelling a subscription

Copyright © 2013 OpenIoT Consortium

3.2 Interaction with the OpenIoT platform

The CPSP engine provides sensor readings from mobile sensors 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. To enable the forwarding of mobile sensor readings from the CPSP engine to the OpenIoT platform, the engine regards the OpenIoT platform as a general subscriber to all publications and forwards all sensor reading to the OpenIoT platform, as shown in Figure **11**. This data will subsequently be stored in the OpenIoT Cloud Data Storage.

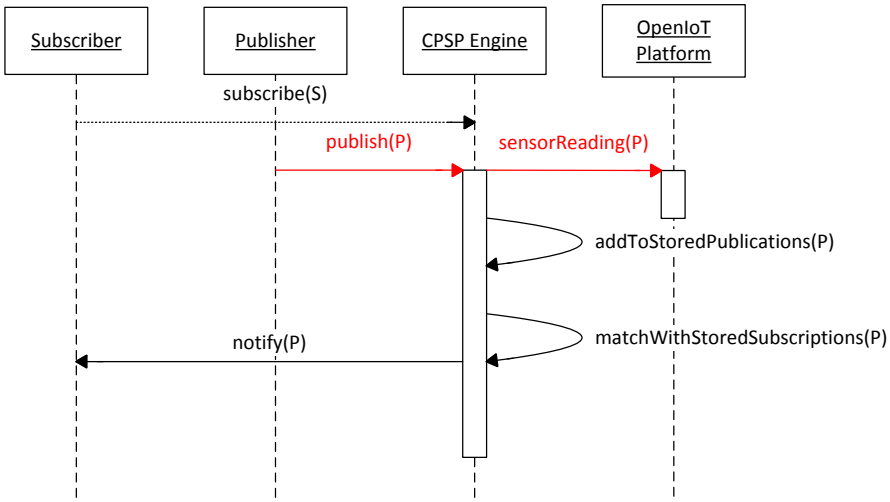

Figure 11 Forwarding sensor readings to the OpenIoT platform

In other words, all sensor readings for which there is current interest, either among subscribers of the publish/subscribe system 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 inactive unless the mobile broker running on the device has received subscriptions matching publisher data from the CPSP engine, as discussed in Section 3.1.

For a full integration of the publish/subscribe system with the existing OpenIoT platform, mobile sensors need to be regarded as virtual sensors generating data streams that can be discovered by the OpenIoT platform and selected as resources for service provision. As the OpenIoT platform supports the addition and removal of virtual sensors, we can use the publish/subscribe methods for announcing and revoking existing announcements to inform the OpenIoT platform about the characteristics of mobile sensors such that they are added into the OpenIoT Cloud Database.

Copyright © 2013 OpenIoT Consortium

Figure **12** shows a sequence of events that lead to the addition of a sensor into the OpenIoT platform, while Figure **13** shows the procedure for removing a mobile sensor from the OpenIoT platform. The former situation is initiated by an announce message generated by a mobile broker which is sent to the CPSP engine. The engine can forward the information about the new mobile sensor to the OpenIoT platform. The latter situation is initiated by a revoke message sent to the CPSP engine that informs the OpenIoT platform that the sensor can no longer produce the data, i.e., it becomes inactive. Thus, the OpenIoT platform is informed about mobile sensor changes, which is important to deliver services to end-users.

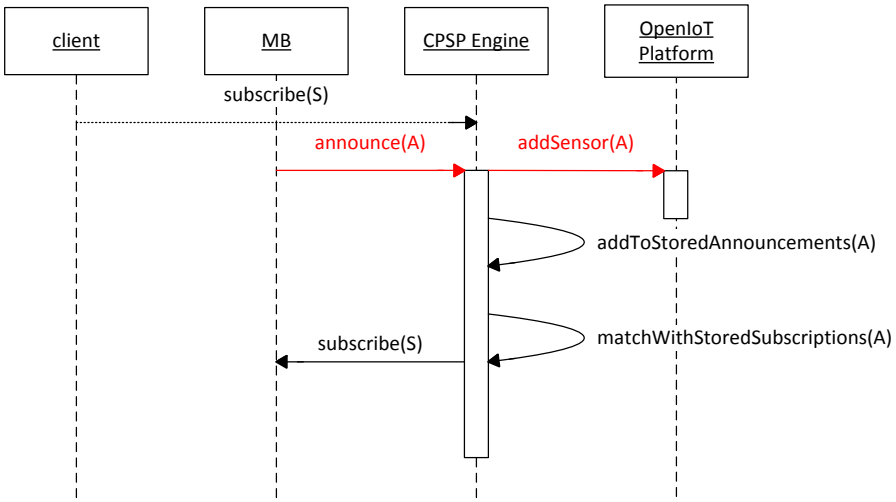

Figure 12 Announcing a new publisher (mobile sensor)

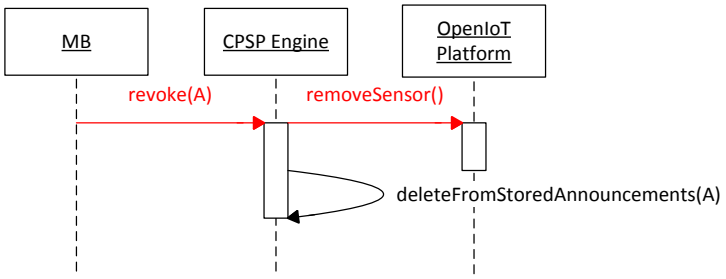

Figure 13 Revoking a publisher (mobile sensor) announcement

The addition of a mobile sensor into the OpenIoT platform does not guarantee that the sensor will subsequently be activated and start producing sensor readings. This will happen in case either

1. any of the CPSP engine subscribers is interested in the data produced by the sensor (an example for this situation is shown in Figure **12**), or
2. there is an explicit request from the OpenIoT platform for the mobile sensor data. It is possible to implement such request, i.e., definition of a new subscription from the OpenIoT platform, by reusing the mechanism for indirect

Copyright © 2013 OpenIoT Consortium

sensor control at the level of X-GSN that checks periodically whether there are active user requests for data from particular (mobile) sensors. The indirect sensor control mechanism is defined in deliverable D5.1.2, Section 3.8.

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. This can be achieved by sending an explicit subscription from the OpenIoT platform or an intermediary component which reuses the previously mentioned indirect sensor control mechanism (see D5.1.2, Section 3.8) to the CPSP engine, as depicted in Figure **14**. Since the OpenIoT platform does not know if these publishers are currently active or not, it always has to explicitly subscribe to the data of interest, and also unsubscribe when it is no longer interested in this data. Figure **15** show the sequence of events when the OpenIoT platform unsubscribes from a mobile sensor since there are no user requests that need the data generated by this sensor.

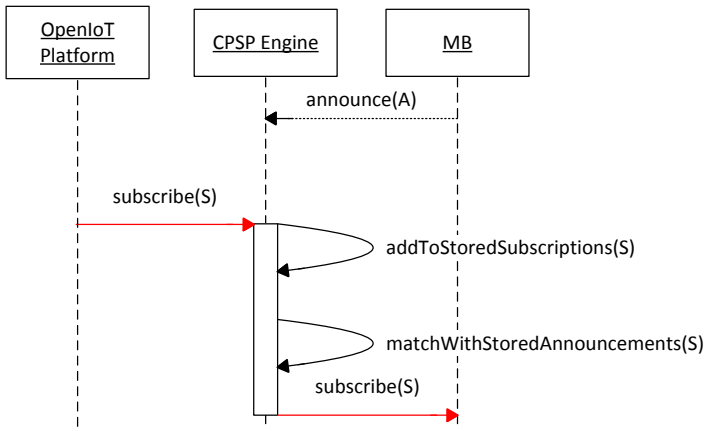

Figure 14 Activating a new subscription from the OpenIoT platform

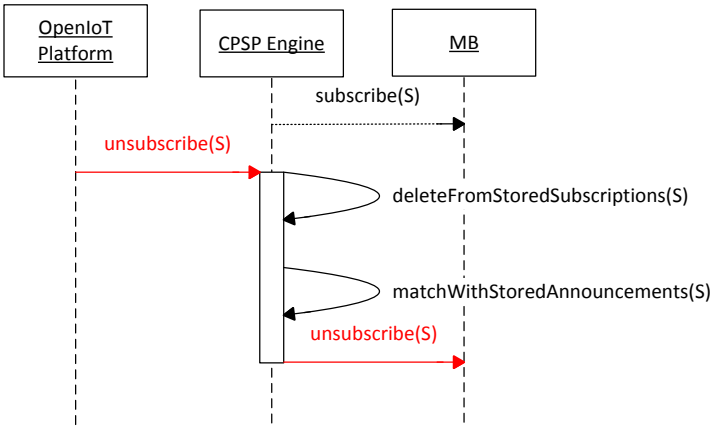

Figure 15 Cancelling a subscription from OpenIoT platform

Copyright © 2013 OpenIoT Consortium

We have so far described the conceptual interaction of the CPSP engine with the OpenIoT platform without specifying explicit OpenIoT components that interact with the CPSP engine. Figure 16 depicts our proposal for such integration: Additional components extending the OpenIoT architecture are marked with red squares. In addition to the CPSP engine, our proposal is to build another intermediary cloud component, Mobile Sensor Controller (MSC) to enable the interaction of the CPSP engine with the OpenIoT platform based on the sequence diagrams depicted in Figure **11** - Figure **15**. On one hand, the MSC would act as a virtual sensor (or 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 MSC would act as a subscriber to the CPSP engine such that is 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 MSC. More details on CPSP integration in the OpenIoT platform will be available in the second version of this deliverable.

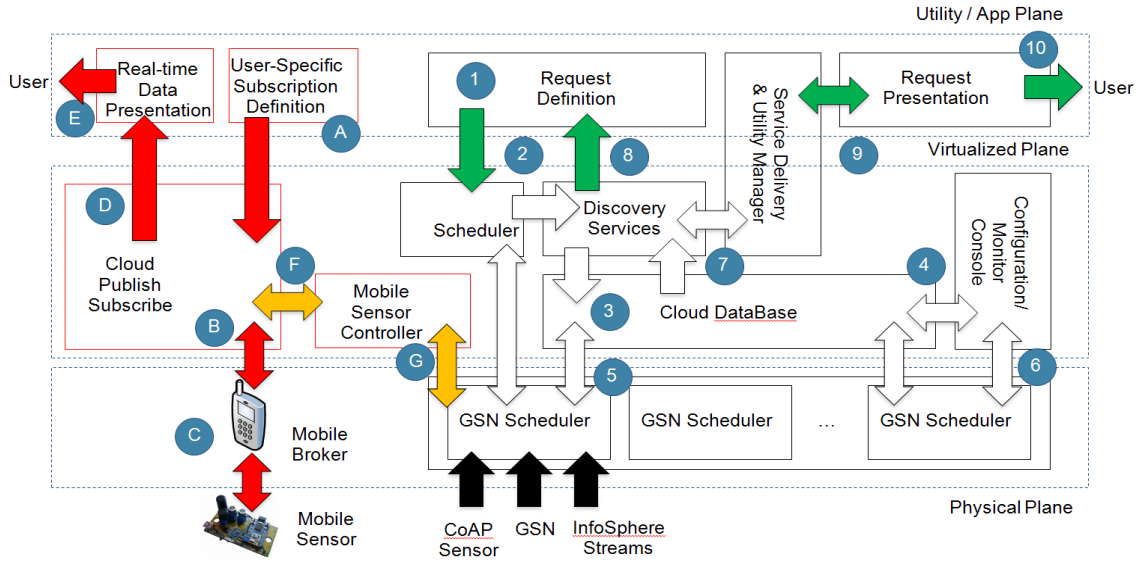

Figure 16 The OpenIoT architecture integrating the publish/subscribe system

Figure 16 depicts the general interaction within the extended OpenIoT architecture and the basic functionality offered by additional components:

A. End User Subscription: User GUI at the application layer needs to enable users to define specific subscriptions in accordance with their information needs. We assume the GUI runs on a mobile device.
B. Sensor Activation & Management: The CPSP enables the activation of mobile sensors and their publishers through interaction with the mobile broker.
C. Data Acquisition and Filtering: A mobile broker enables data acquisition from mobile sensors and filtering of acquired data on mobile devices.
D. Publish/Subscribe Data Filtering: The CPSP performs data filtering.
E. Near Real-Time Data Presentation: User GUI presents the data matching user-specific subscriptions received from the CPSP engine.
F. Mobile Sensor Data Adaptation: The MSC enables adaptation of sensor data received by the CPSP engine to GSN.
G. Mobile Sensor Data Exchange: The data is exchanged between the MSC and GSN Scheduler.

Copyright © 2013 OpenIoT Consortium

4 CLOUD-BASED PUBLISH/SUBSCRIBE PROCESSING ENGINE

4.1 Cloud-based architecture

In this section we present the architecture of the CPSP engine. We refer to it in this document also 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. 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 incoming data streams. 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 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 as defined in the top-k/w model, since top-k/w subscriptions reference only future publications.

The CPSP engine performs the following tasks:

1. receives publications from publishers and subscriptions from subscribers,
2. manages active subscriptions and valid or top/k/w publications,
3. matches active subscriptions to publications, and
4. delivers matching publications to subscribers.

Out of these four tasks, the matching of publications to subscriptions is the most demanding one and thus we need to replicate the matchers and dynamically allocated cloud resources for matching in accordance with the processing load.

An overview of the CPSP engine architecture is given in Figure **17**. There are logically two parts of the CPSP engine - a fixed part 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 also over TCP.

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/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 flat or dynamic forest structure.

Copyright © 2013 OpenIoT Consortium

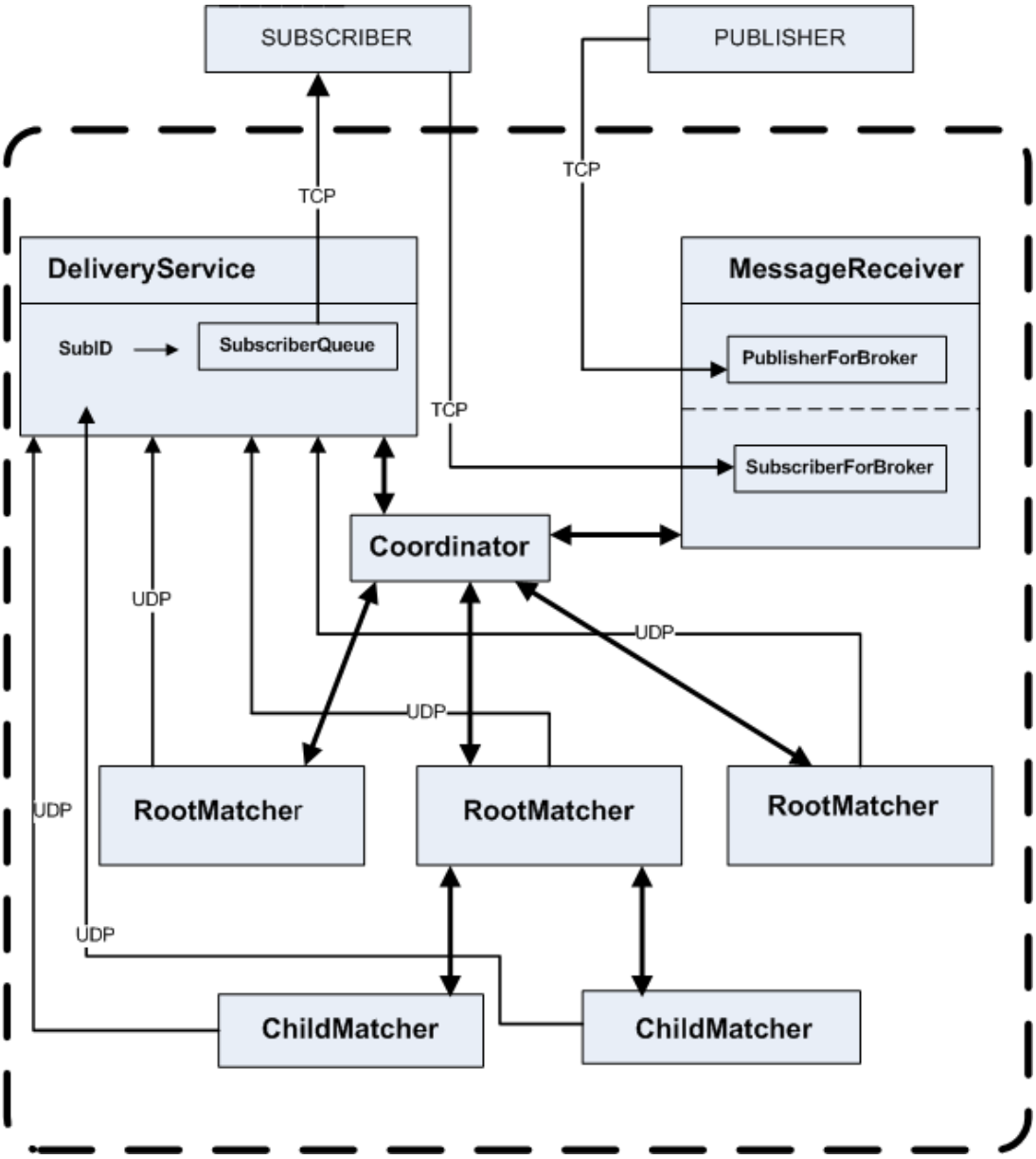

Figure 17 System components and their communication pathways

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 a RootMatcher in this setup cannot have child matchers and cannot initiate the creation of a new matcher in cases when it becomes overloaded.

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 one *N*-th of the total subscriptions being processed by the CPSP engine. Incoming publications (PublishMessage objects) are

Copyright © 2013 OpenIoT Consortium

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. A handler for a subscriber is called a *SubscriberForBroker*, a handler for a publisher is called a *PublisherForBroker* and handler for a mobile broker

Copyright © 2013 OpenIoT Consortium

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 **18**.

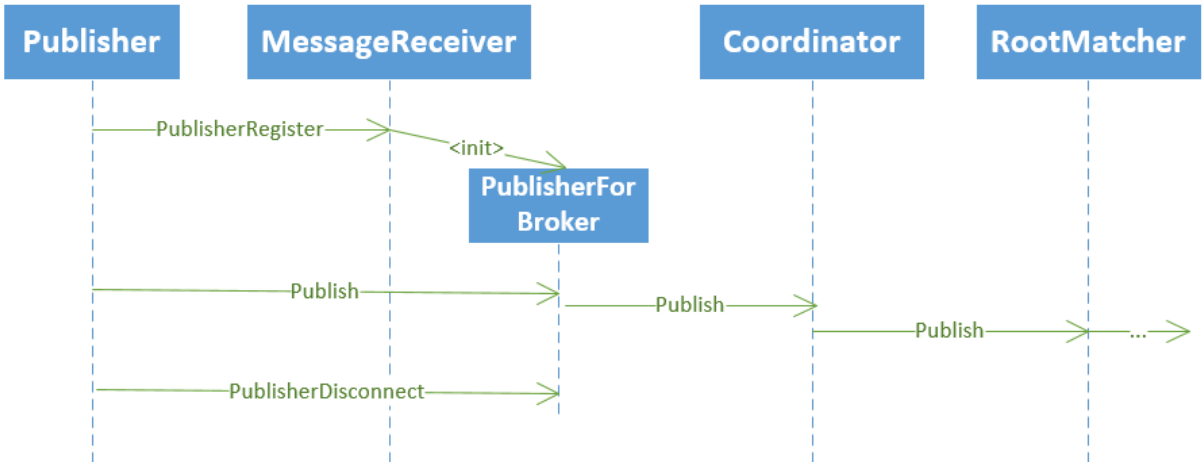

Figure 18 Communication of a publisher with the CPSP engine

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 **19** and Figure **20**. The main reason is in the fact 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 **19** 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

Copyright © 2013 OpenIoT Consortium

all the subscriptions of the unregistering subscriber need to be removed from subscription data structures upon its unregistration. 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.

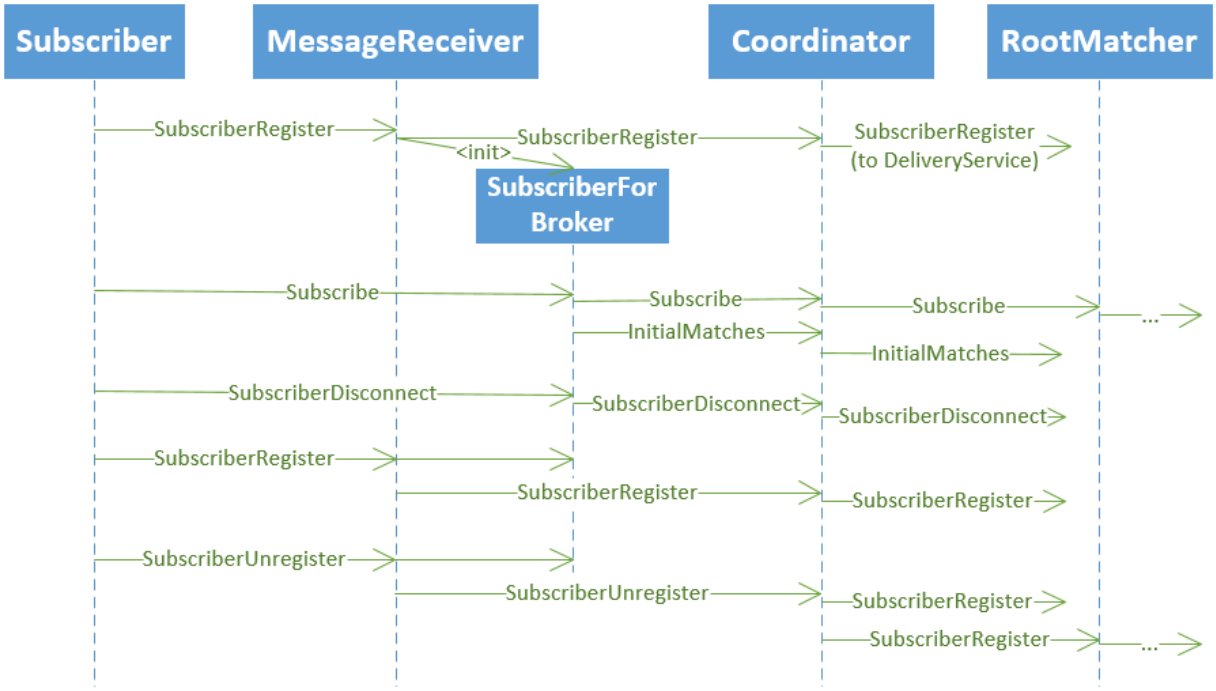

Figure 19 Communication of a subscriber with the CPSP engine (the Message Receiver part)

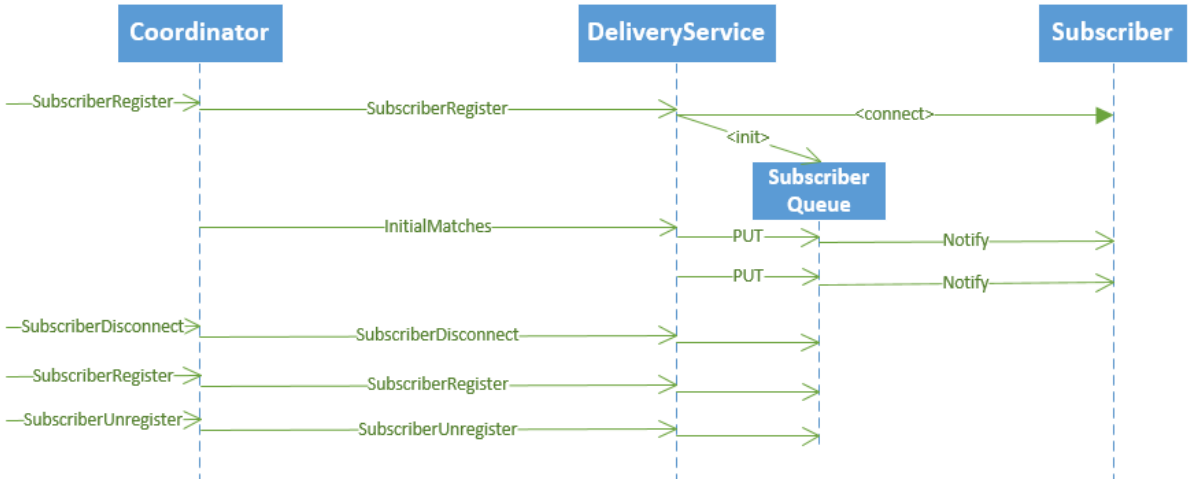

Figure 20 Communication of a subscriber with the CPSP engine (the Delivery Service part)

Copyright © 2013 OpenIoT Consortium

Figure **20** 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 **20**. 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. 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.

The only remaining message the DeliveryService can receive from the Coordinator component is the *InitialMatches* message, which carries a set of publications for sending to one subscriber and that is exactly what the DeliveryService does, it puts them all in subscriber's queue object for sending. In addition, the DeliveryService is also 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 that belong to subscribers with matching subscriptions. The publication is sent to each of the subscribers, or put in their respective queues.

4.2 Load balancing in the Cloud

The elasticity and scalability of the system is achieved through the splitting and merging of matcher components. Both the splitting and the merging are triggered by the processing times of the last *N* publications, where *N* is a parameter of the engine.

4.2.1 Splitting of matchers

The splitting of a matcher is initiated when the minimal processing time in the window of *N* last publication matching is greater than a given threshold, i.e. when the processing time for all publications in the window is larger than a given threshold. The splitting criterion defined in this way is robust with regard to sudden spikes in processing times, which do not indicate a real overload of a processor, but is also a bit unfortunate due to the manual setting of the threshold on engine start-up and the inability to change it during runtime.

Copyright © 2013 OpenIoT Consortium

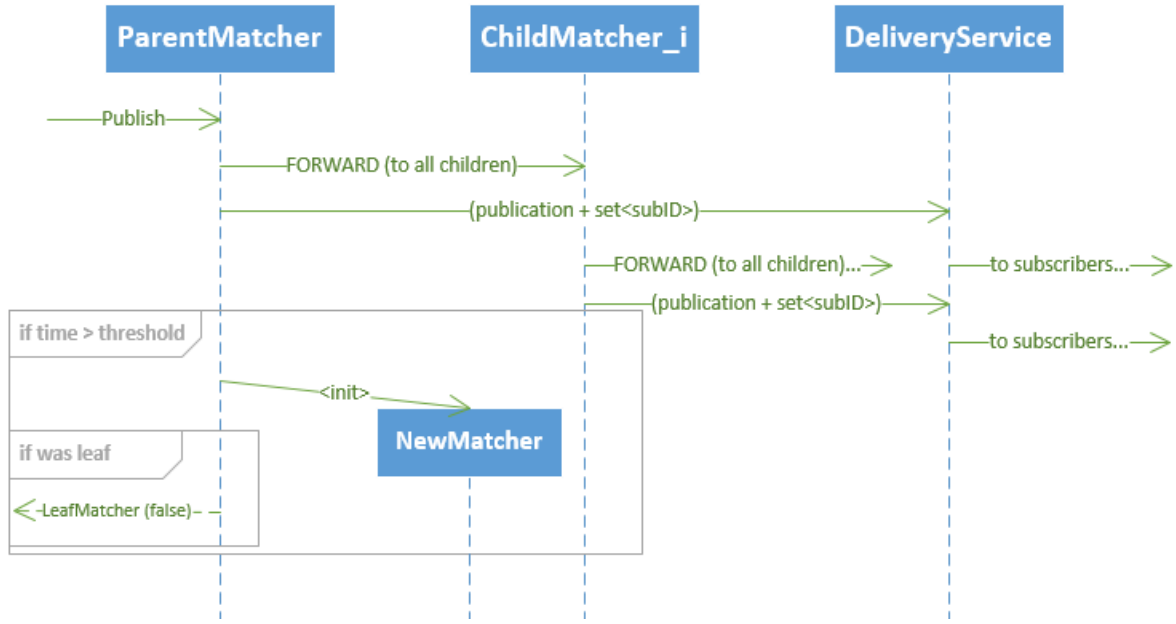

Figure 21 Processing of a *Publish* message followed by a matcher splitting

After a *Publish* request has been processed and forwarded (Figure **21**), a check is made whether the minimal processing time in the window (including the one just completed) is greater than the given splitting threshold. If it is, a search for appropriate subscription data structure node with the biggest matching time among all nodes in the local substructure of the structure is started. When such a node is found, a *proxy* node is created that substitutes that node in its old node structure, and that node, along with the entire substructure to which it is the root, is sent to the newly created Matcher. Newly created matcher can be a child matcher to old matcher or another matcher added in parallel with the old one. The final step is for the matcher to notify its matcher if it had stopped being a leaf matcher after splitting. This is done by the matcher sending a *LeafMatcher* message.

4.2.2 Merging of matchers

The merging of a matcher is initiated when the maximal processing time in the window of *N* last publication matching is less than a given threshold, i.e. when all publications in the window where processed in lower processing time than the defined threshold. This criterion is, like the splitting criterion, chosen for its robustness with regard to sudden spikes in processing times.

Copyright © 2013 OpenIoT Consortium

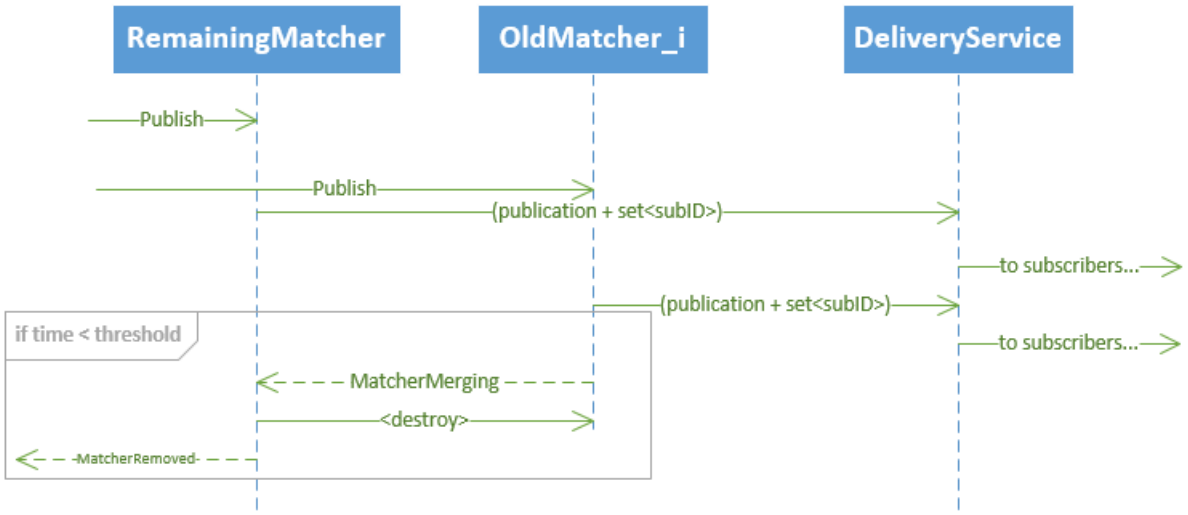

Figure 22 Processing of a *Publish* message followed by matchers merging

The same as with the splitting of matchers, checking of the previously described condition is performed after a *Publish* request has been processed and forwarded. If the condition is met, the matcher that wants to be merged (the OldMatcher in Figure **22**) sends a *MatcherMerging* message to its parent or some other matcher that is running in parallel. A *MatcherMerging* message contains the substructure of the merging matcher along with other relevant publish-subscribe information. Upon receiving a *MatcherMerging* message, the *proxy* node on the receiving matcher:

1. destroys the corresponding OldMatcher,
2. processes all requests (over the received substructure) that might have been sent to the old matcher before it was destroyed, but didn't have a chance to get processed,
3. replaces itself in the remaining matcher with the received substructure,
4. if it was a child matcher, it sends a *MatcherRemoved* message up the local substructure, and to its matcher's parent matcher.

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.

Copyright © 2013 OpenIoT Consortium

Table 1 Supported operators for Boolean subscriptions

Operator	Description
LESS_THAN	a numeric operator ($a<c$)
LESS_OR_EQUAL	a numeric operator ($a\leq c$)
EQUALS	both a numeric and a string operator ($a==c$)
GREATER_THAN	a numeric operator ($a>c$)
GREATER_OR_EQUAL	a numeric operator ($a\geq c$)
BETWEEN	a numeric operator that takes two values, a lower and an upper bound ($c_1\leq a\leq c_2$)
CONTAINS_STRING	a string operator that matches all strings that contain a given string
STARTS_WITH_STRING	a string operator that matches all strings that start with a given string
ENDS_WITH_STRING	a string operator that matches all strings that end with a given string

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 $\in$ 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=[NO2 > 40μgm^{-3} AND 45.81≤lat≤45.82 AND 15.96≤long≤15.98]

p_j=[NO2 == 45μgm^{-3} AND lat==45.81543 AND 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 can be defined as a set of triplets, but additionally it needs to include parameters *k* and *w*, and a scoring function. 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 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 *k* best publications).

Copyright © 2013 OpenIoT Consortium

If we select sliding-window k-nearest neighbours (k-NN) subscription for an example, and both subscriptions and publications are represented as points in a d-dimensional Euclidean space. The score of a publication *p* with respect to a subscription *s* can be calculated as: $u_s(p) = d(p_s,p_p) = \left[\sum_{i=1}^{d}(v_i - u_i)^2\right]^{1/2}$ where $p_p=\{v_1,v_2,\ldots,v_d\}$ and $p_s=\{u_1, u_2,\ldots u_d\}$ are points representing the publication *p* and subscription *s*, respectively. Obviously, this scoring function prefers lower scores to higher scores. With a k-NN query, a user could define a subscription such that over time he/she receive 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. [lat=45.815 AND long=15.977], parameters k=10 and w=24 hours, while the scoring function is the Euclidean distance over latitude and longitude. The publications in this case remain the same as for Boolean subscriptions, e.g. p_j=[NO2 == 45µgm^{-3} AND lat==45.81543 AND long==15.97433].

The matching process will calculate 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.

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.

Copyright © 2013 OpenIoT Consortium

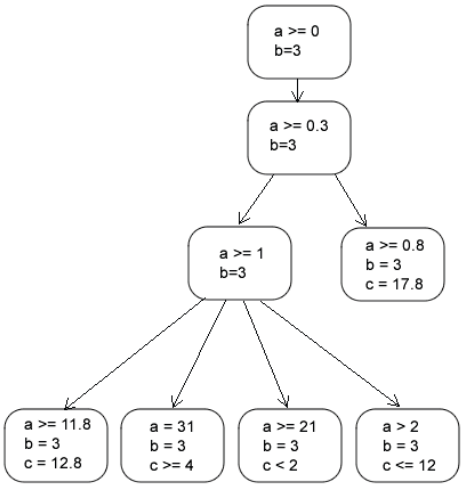

Figure 23 An example of a Boolean subscription forest

A simple example of a subscription data structure is shown in Figure **23**. 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). It is shown that the tree subscription is 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, so whenever a node is added or removed from the forest, all relationships between the rest of the forest is maintained.

The processing of incoming publications is efficient since the algorithm can very quickly identify which part of the forest is covering incoming publication. For matching purposes of the publication rest of the forest (i.e. child nodes of those parent nodes that are not covering the publication) are neglected and not considered at all for delivery of the publication.

4.4.2 Top-k/w matching

Subscription indexing reduces the number of publications that a Boolean or top-k/w subscription needs to process by identifying and neglecting those publications which are certainly not of interest for a subscription. Hereafter we sketch indexing techniques for distance and aggregation scoring functions.

Distance and aggregation (i.e. weighted sum) functions are applied to structured publications represented as points in a multidimensional attribute space. In existing approaches, a regular grid is used to index subscriptions in such space. A regular grid divides an attribute space in cells of equal size, while a subscription threshold defines the subscription subspace of interest as shown in Figure **24**. Usually, each cell contains a list of subscriptions whose subspace of interest completely or at least partially covers the cell. This makes the processing of incoming publications more efficient since only interested subscriptions are informed about a publication appearing in the cell. Please note that the subspace of interest is covered by the *cells of interest* from the regular grid that encompass a larger portion of the attribute space than the subspace of interest itself. When the subscription subspace of interest changes (i.e. expands or contracts) this can reflect as a change in the corresponding cells of interest. Since this situation happens less often than a regular threshold

Copyright © 2013 OpenIoT Consortium

change, subscription cells of interest change less often than its threshold which is important for a distributed setup where processing node need to exchange threshold updates or changes of subscription cells of interest.

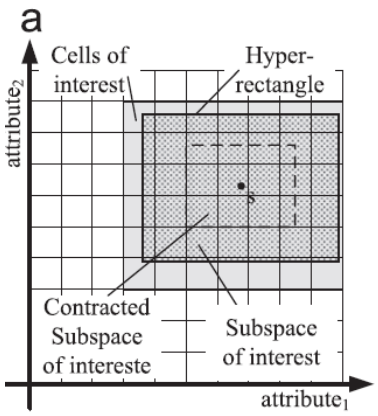

Figure 24 Top-k/w subscription indexing

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[1] space under the Documentation[2] section.

5.1 Cloud Broker

As already stated, the OpenIoT Cloud Broker, i.e., the CPSP engine, accepts all messages from subscriber, publisher or 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 2 Cloud Broker's public methods and constructor

Broker (name:String, brokerPort: int): Broker --- **run** (): void **connectToBroker**(brokerIP:String, brokerPort:int): void

[1] https://github.com/OpenIotOrg/openiot/wiki

[2] https://github.com/OpenIotOrg/openiot/wiki/Documentation

Copyright © 2013 OpenIoT Consortium

disconnectFromBroker(brokerIP:String, brokerPort:int): void **shutdown**(): void

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

Table 3 Implemented Cloud Broker API definition

Service Name	Input	Output	Info
Broker (constructor)	String name, int brokerPort	Broker	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.
run		void	A method used to start the cloud broker.
shutdown		void	A method used to stop the cloud broker. All data (i.e. valid subscriptions and publications) remain in memory until the broker object is destroyed.

5.1.2 Download, Deploy & Run

5.1.2.1 Developer

5.1.2.1.1 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 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 scheduler is available under the "openiot/modules/pub-sub/cloudbroker" folder.

Copyright © 2013 OpenIoT Consortium

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 Cloud Broker project.
  2. Type this command to build and deploy the archive:
     - mvn clean compile assembly:single
  3. This will build the service in target folder and now service is ready to be started by command in terminal: `java –jar target\CloudBroker.jar <path_to_config_file> <classpath>`

- Undeploy the Cloud Broker service
  1. Stop the running instance of the service by typing "`QUIT`"

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[3] under the Users>Downloads[4] 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: `java –jar CloudBroker.jar <path_to_config_file> <classpath>`.

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

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 publications to the system, without any filtering on end-user device

[3] https://github.com/OpenIotOrg/openiot/wiki

[4] https://github.com/OpenIotOrg/openiot/wiki/Downloads

Copyright © 2013 OpenIoT Consortium

5.2.1 Main Released Functionalities & Services

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

Table 4 Publisher public methods and constructor

Publisher (name:String, brokerIP:String, brokerPort: int): Publisher
---
connect (): void
disconnectFromBroker(): void
reconnect(): void
reconnect(brokerIP:String, brokerPort: int): void
publish(publication: Publication): void
unpublish(publication: Publication): void

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

Table 5 Implemented Publisher API definition

Service Name	Input	Output	Info
Publisher (constructor)	String name, String brokerIP, int brokerPort	Publisher	Used to create a Publisher entity. Requires as input a name of the entity in String format, IP address and port number of the broker. Output is the Publisher object which will be used for publishing new data.
connect		void	A method used to connect to previously defined cloud broker.
disconnectFromBroker		void	A method used to disconnect from cloud broker. All data (i.e. valid publications) remain in the memory until the publisher object is destroyed or until their validity expires.
connect	String brokerIP int brokerPort	void	A method used to connect to specified cloud broker
publish	Publication publication	void	Used to publish new data. Input parameter is a valid publication.

Copyright © 2013 OpenIoT Consortium

unpublish	Publication publication	void	Used to delete previously published data. Input parameter is the publication that is removed from the cloud broker but not from the subscribers that already received the notification with specified data.

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 scheduler is available under the "openiot/modules/pub-sub/publisher" 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
    - 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.
    4. Open a command line and navigate to the root directory of the Publisher project.
    5. Type this command to build and deploy the archive:
        - mvn clean compile assembly:single
    6. This will build the service in target folder and now service is ready to be started by command in terminal: `java –jar target\Publisher.jar <path_to_config_file> <path_to_publications_folder>`

- Undeploy the Cloud Broker service
    1. Stop the running instance of the service by typing "`QUIT`"

Copyright © 2013 OpenIoT Consortium

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[5] under the Users>Downloads[6] section.

5.2.2.2.2 Deployment/Undeployment

Deploy: To deploy the Publisher service, copy the "Publisher.java" to the user device Run the terminal command: `java –jar Publisher.jar <path_to_config_file> <path_to_publications_folder>`.

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

Subscriber (name:String, brokerIP:String, brokerPort: int): Subscriber --- **connect** (): void **disconnectFromBroker**(): void **connect**(brokerIP:String, brokerPort:int): void **subscribe**(subscription: Subscription): void **unsubscribe**(subscription: Subscription): void **setNotificationListener** (notificationListener: NotificationListener): void

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

[5] https://github.com/OpenIotOrg/openiot/wiki

[6] https://github.com/OpenIotOrg/openiot/wiki/Downloads

Copyright © 2013 OpenIoT Consortium

Table 7 Implemented Subscriber API definition

Service Name	Input	Output	Info
Subscriber (constructor)	String name, String brokerIP, int brokerPort	Subscriber	Used to create a Subscriber entity. Requires as input the entity name in String format, IP address and port number of the broker. Output is the Subscriber object which will be used for subscribing on behalf of a user and receiving incoming notifications.
connect		void	A method used to connect to previously defined cloud broker.
disconnectFromBroker		void	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 their validity expires.
connect	String brokerIP int brokerPort	void	A method used to connect to specified cloud broker
subscribe	Subscription subscription	void	Used to subscribe to some specific data. Input parameter is a valid subscription request.
unsubscribe	Subscription subscription	void	Used to delete previously defined subscription request. Input parameter is the subscription that is removed from the cloud broker.
setNotificationListener	NotificationListener notListener	void	Used to set a processing class for notifications. Notifications are received as a response to user's subscriptions.

Copyright © 2013 OpenIoT Consortium

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 scheduler is available under the "openiot/modules/pub-sub/subscriber" 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.
    7. Open a command line and navigate to the root directory of the Subscriber project.
    8. Type this command to build and deploy the archive:
        - mvn clean compile assembly:single

This will build the service in target folder and now service is ready to be started by command in terminal: `java -jar target\Subscriber.jar <path_to_config_file> <path_to_subscriptions_folder>`

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

5.3.2.2 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[7] under the Users>Downloads[8] section.

[7] https://github.com/OpenIotOrg/openiot/wiki

[8] https://github.com/OpenIotOrg/openiot/wiki/Downloads

Copyright © 2013 OpenIoT Consortium

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 Subscriber.jar <path_to_config_file> <path_to_subscriptions_folder>`.

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

5.4 Source code examples

5.4.1 CloudBroker

Table 8 Cloud broker source code snippet

```
//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 the port 12345.

5.4.2 Subscriber and subscribing process

Table **9** shows a code snippet that creates the Subscriber object by using a configuration file. The configuration file contains name of the subscriber object, an IP address and port of the broker. After the subscriber is created, it is necessary to create a notification listener, i.e. a piece of the code that will process incoming notifications. In the example the notification listener only prints out incoming notification, but it can be extended to make more complex processing (a notification listener can be defined in a separate class). After that the subscribers connects to the broker, defines and subscribes using three subscriptions with infinite validity. At the end of the snippet is shown how to disconnect from a broker.

Copyright © 2013 OpenIoT Consortium

Table 9 Subscribing process source code snippet

```
//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();
    }
});

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

//define subscriptions
TripletSubscription ts1 = new TripletSubscription(-1,
System.currentTimeMillis());
ts1.addPredicate(new Triplet("num1", 5.4, Operator.GREATER_OR_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 name of the publisher, an IP address and port of the broker. After the publisher is created, a new publication is defined. Publication validity is infinite, but in the snippet is also provided a way how to make a publication with finite validity. After that, two properties (i.e. data that will be published) are added to the publication. At the end is given an example how to send publication to the broker and after that how to disconnect from the broker.

Copyright © 2013 OpenIoT Consortium

Table 10 Publishing process source code snippet

```
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 propertis 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();
```

6 PRELIMINARY EXPERIMENTAL RESULTS

In this section we present the results of an initial experiment which tests the CPSP engine under constant load. The engine is tested in two setups: 1) with a single Boolean matcher which is responsible for processing all subscriptions and publications, and 2) with a flat architecture running N=15 independent matchers as separate processes in parallel. We evaluated the matching performance when increasing the number of subscriptions with a constant publication frequency.

The experiment is performed on two desktop PCs (Intel Core i3 with 2 cores, 3.3 GHz, 4 GB) within a LAN. In both experiments the CPSP engine is running on one PC, while we have started one subscriber and one publisher to generate load on the other PC.

The data used in the experiments are real world sensor data collected during the scope of the OpenSense project[9]. In particular, the data set contains up to 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 set. As we do not have real user queries, subscriptions are generated such that we choose a random measurement from the OpenSense set and associate it with a random numerical operator (excluding BETWEEN). Additionally, we generate subscriptions related to specific

[9] http://opensense.epfl.ch

Copyright © 2013 OpenIoT Consortium

time periods regardless of sensor type such that we define constraints on timestamps. Such “timestamp” subscriptions are more general and without much overlap and mutual covering with other subscriptions.

Figure 25 and Figure 26 show the average matching times on CPSP engine for a single publication, given in microseconds, while increasing the number of CPSP concurrent subscriptions. The measurements were made for 1000, 2000, 3000, 4000, 5000 and 10000 subscriptions. We varied in each experiment the number of generated publications (10000, 50000, 100000 and 150000 publications).

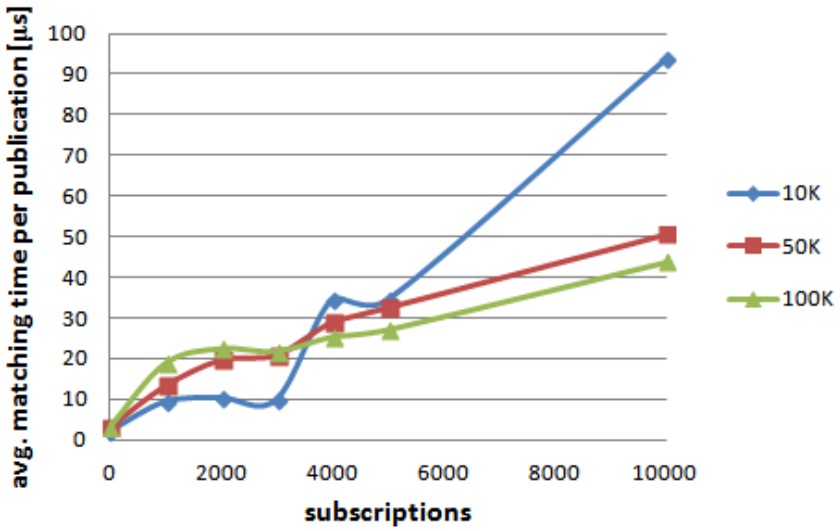

Figure 25 Average matching time per publication with one Boolean matcher

Figure **25** shows the average matching time when only one matcher process is running on a PC. We can conclude that the plot is still unstable and variable with 10000 publications and can thus be disregarded, while the plots with 50000 and 100000 publications show sub-linear growth. One can see that with 10000 concurrent subscriptions in the system the average matching time per publication is around 50µs.

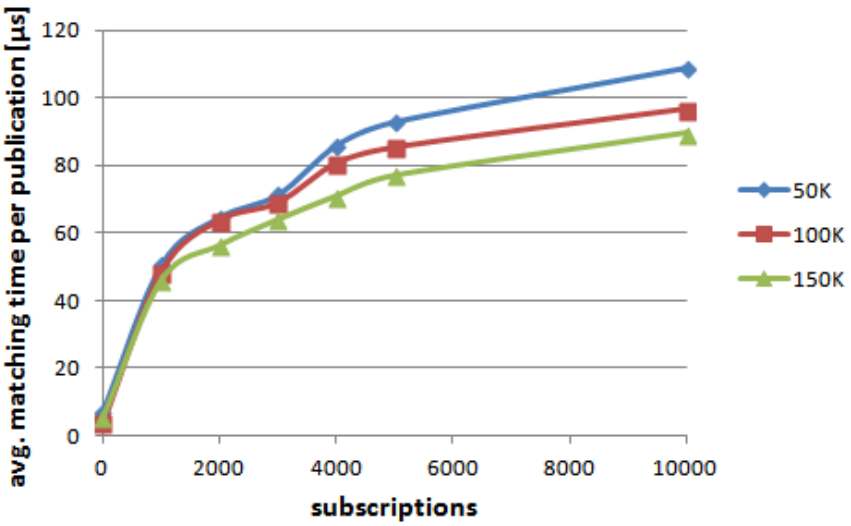

Figure 26 Average matching time per publication with
15 parallel Boolean matchers

Copyright © 2013 OpenIoT Consortium

Figure 26 shows the average matching time when there are 15 matcher processes running on a PC which is obviously not a cloud-based deployment as each process should run in a separate cloud VM, but the experiment will show us some important trends. Here the matching time per publication obviously increases compared to the previous figure with a single matcher due to hardware limitations. Note that in this setup when a single publication is submitted to 15 different processes that equally share the subscription load, we assume that the matching time equals the time it took for the slowest process to finish matching.

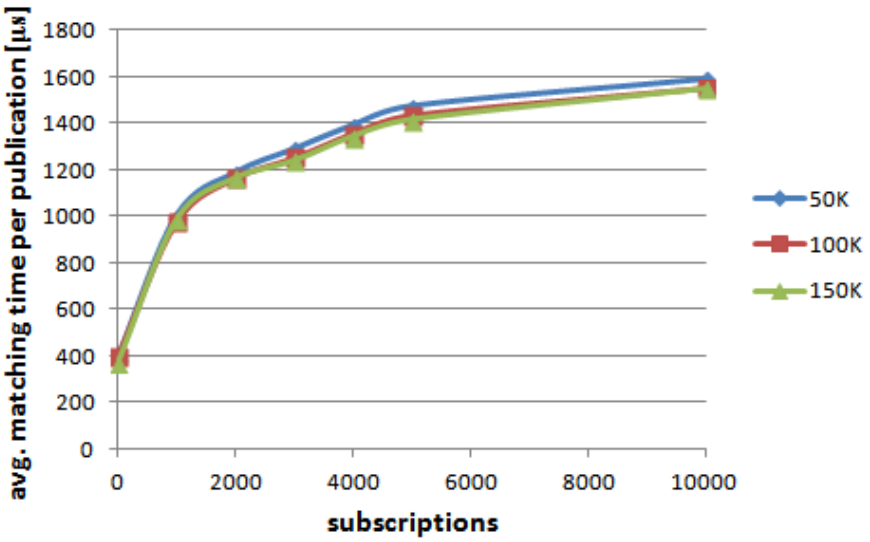

Figure 27 Overall average processing time per publication with 15 parallel Boolean matchers

Figure **27** plots the overall average processing time per publication in a deployment running 15 matchers. The plot includes the latency incurred by MessageReceiver and Coordinator that need to communicate with 15 matchers which causes significant delays. However, the graph shows the scalability trend when increasing the number of subscribers and gives indication of processing scalability since the matching operation is in this case burdened and constrained by the inter-process communication. Further experiments are needed to evaluate the CPSP engine under high and variable processing load within a cloud environment.

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 and enables flexible filtering of data streams in the cloud and on 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

Copyright © 2013 OpenIoT Consortium

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 application monitoring system. Continuous querying is enabled through a graphical interface such that a data stream diagram is created 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 and definition of arbitrary function of inputs. StreamCloud is an intermediate layer that parallelizes user query into subqueries that are executed on different nodes. Currently, distributed version of data processing system Borealis is used for processing parallel divided 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 such 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, 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.

Copyright © 2013 OpenIoT Consortium

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.

Copyright © 2013 OpenIoT Consortium

8 CONCLUSIONS

Deliverable D4.5.1 corresponds to the first 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/Subscribe for the Internet of Things (CUPUS). The release comprises a prototype implementation of the CUPUS processing engine, along with accompanying documentation which is provided as part of this document. Note that the open source implementation of the above mentioned component of the release 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 description of its technical implementation and includes a thorough explanation of CUPUS interaction with the rest of the OpenIoT platform. We also include an API specification and code examples, and provide initial preliminary experimental results which investigate CUPUS performance under high processing load.

The second version of this deliverable will include an elastic implementation of the CPSP engine supporting both Boolean and top-k/w subscriptions. In addition, the CPSP engine will be fully integrated with other OpenIoT platform components creating a general IoT platform supporting both stationary and mobile ICOs.

9 REFERENCES

[ApacheS4] http://incubator.apache.org/s4/

[Aurora02] Don Carney, Uğur Çetintemel, Mitch Cherniack, Christian Convey, Sangdon Lee, Greg Seidman, Michael Stonebraker, Nesime Tatbul, and Stan Zdonik. 2002. Monitoring streams: a new class of data management applications. In Proceedings of the 28th international conference on Very Large Data Bases (VLDB '02). VLDB Endowment 215-226.

[BlueDove11] Ming Li, Fan Ye, Minkyong Kim, Han Chen, and Hui Lei. 2011. A Scalable and Elastic Publish/Subscribe Service. In Proceedings of the 2011 IEEE International Parallel & Distributed Processing Symposium (IPDPS '11). IEEE Computer Society, Washington, DC, USA, 1254-1265.

[InfoSphere] http://www-03.ibm.com/software/products/en/infosphere-streams

[Kinesis] http://aws.amazon.com/kinesis/

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

[STREAM06] Arvind Arasu, Shivnath Babu, and Jennifer Widom. 2006. The CQL continuous query language: semantic foundations and query execution. The VLDB Journal 15, 2 (June 2006), 121-142.

Copyright © 2013 OpenIoT Consortium

[StreamCloud10] Vincenzo Gulisano, Ricardo Jimenez-Peris, Marta Patino-Martinez, and Patrick Valduriez. 2010. StreamCloud: A Large Scale Data Streaming System. In Proceedings of the 2010 IEEE 30th International Conference on Distributed Computing Systems (ICDCS '10). IEEE Computer Society, Washington, DC, USA, 126-137.

[Mühl2006] Gero Mühl, Ludger Fiege, and Peter Pietzuch. 2006. *Distributed Event-Based Systems*. Springer-Verlag New York, Inc., Secaucus, NJ, USA.

[Sadoghi2011] Mohammad Sadoghi and Hans-Arno Jacobsen. 2011. BE-tree: an index structure to efficiently match boolean expressions over high-dimensional discrete space. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data* (SIGMOD '11). ACM, New York, NY, USA, 637-648. DOI=10.1145/1989323.1989390

[Pripužić2013] Krešimir Pripužić, Ivana Podnar Žarko, Karl Aberer, Top-k/w publish/subscribe: A publish/subscribe model for continuous top-k processing over data streams, Information Systems, Volume 39, January 2014, Pages 256-276, ISSN 0306-4379, http://dx.doi.org/10.1016/j.is.2012.03.003.

[Mühl2002] Gero Mühl, Ludger Fiege, and Alejandro P. Buchmann. 2002. Filter Similarities in Content-Based Publish/Subscribe Systems. In *Proceedings of the International Conference on Architecture of Computing Systems: Trends in Network and Pervasive Computing* (ARCS '02), Hartmut Schmeck, Theo Ungerer, and Lars C. Wolf (Eds.). Springer-Verlag, London, UK, UK, 224-240.

[Mouratidis2007] Kyriakos Mouratidis and Dimitris Papadias. 2007. Continuous Nearest Neighbor Queries over Sliding Windows. *IEEE Trans. on Knowl. and Data Eng.* 19, 6 (June 2007), 789-803. DOI=10.1109/TKDE.2007.190604

[Eugster2003] Patrick Th. Eugster, Pascal A. Felber, Rachid Guerraoui, and Anne-Marie Kermarrec. 2003. The many faces of publish/subscribe. *ACM Comput. Surv.* 35, 2 (June 2003), 114-131. DOI=10.1145/857076.857078

[Ilyas2008] Ihab F. Ilyas, George Beskales, and Mohamed A. Soliman. 2008. A survey of top-*k* query processing techniques in relational database systems. *ACM Comput. Surv.* 40, 4, Article 11 (October 2008), 58 pages. DOI=10.1145/1391729.1391730

[Golab2003] Lukasz Golab and M. Tamer Özsu. 2003. Issues in data stream management. *SIGMOD Rec.* 32, 2 (June 2003), 5-14. DOI=10.1145/776985.776986

Copyright © 2013 OpenIoT Consortium