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

D3.4.1 Publish/Subscribe Middleware for Mobile Internet-Connected Objects a

Document Id:	OpenIoT-D341-131220-Draft
File Name:	OpenIoT-D341-131220-Draft.pdf
Document reference:	Deliverable 3.4.1
Version:	Draft
Editor:	Aleksandar Antonić, Krešimir Pripužić
Organisation:	UNIZG-FER
Date:	2013 / 12 / 20
Document type:	Deliverable
Security:	PU (Public)

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	Martina Marjanović	UNIZG-FER	2013/11/21	Added initial version of Section 4 and Section 5
V04	Martina Marjanović	UNIZG-FER	2013/11/24	Added initial version of Section 3
V05	Aleksandar Antonić	UNIZG-FER	2013/11/25	Added input to Section 2
V06	Ivana Podnar Žarko	UNIZG-FER	2013/11/25	Added input to Section 1 & 2
V07	Aleksandar Antonić	UNIZG-FER	2013/11/27	Modified input to Section 2
V08Y	Martina Marjanović	UNIZG-FER	2013/11/27	Modified input to Section 5
V09	Krešimir Pripužić	UNIZG-FER	2013/11/27	Added input to Sections 1 & 3
V10	Aleksandar Antonić	UNIZG-FER	2013/11/28	Modified Section 2 & 3
V12	Martina Marjanović	UNIZG-FER	2013/11/28	Modified Section 2.3
V13	Krešimir Pripužić	UNIZG-FER	2013/11/28	Modified Section 3.1
V14	Aleksandar Antonić	UNIZG-FER	2013/11/28	Final input to Section 3 & 4
V15	Krešimir Pripužić	UNIZG-FER	2013/11/28	Final modifications to Section 1, Added Section 7
V16	Krešimir Pripužić	UNIZG-FER	2013/12/02	Final corrections
V17	Armin Haller & Arkady Zaslavsky	CSIRO	2013/12/12	Technical Review
V18	Reinhard Herzog	IOSB	2013/12/12	Quality Review
V19	Ivana Podnar Žarko	UNIZG/FER	2013/12/15	Answers to TR and QR comments Corrections and Amendments
V20	Martin Serrano	DERI	2013/12/15	Circulated for Approval
V21	Martin Serrano	DERI	2013/12/20	Approved
Draft	Martin Serrano	DERI	2013/12/23	EC Submitted

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

TERMS AND ACRONYMS

CoAP	Constrained Application Protocol
CoRE	Constrained RESTful Environments
CPSP	Cloud-based Publish/Subscribe Processing
DTLS	Datagram Transport Layer Security
ETSI	European Telecommunications Standards Institute
GCM	Google Cloud Messaging
GPS	Global Positioning System
HTTP	Hypertext Transfer Protocol
IETF	Internet Engineering Task Force
IoT	Internet of Things
IP	Internet Protocol
M2M	Machine-to-Machine
MAC	Media Access Control
MB	Mobile Broker
MIO	Mobile Internet-connected Objects
MoPS	Mobile Publish/Subscribe
PEIR	Personal Environmental Impact Record
QoS	Quality of Service
REST	Representational state transfer
RFCOMM	Radio Frequency Communication
SPP	Serial Port Profile
TCP	Transmission Control Protocol
UDP	User Datagram Protocol
URI	Uniform resource identifier

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 and seamlessly acquire their data.

The ubiquity of wearable sensors and proliferation of mobile devices with a number of built-in sensors creates a vast amount of data sources that continuously generate huge amounts of data. In the context of Internet of Things (IoT), these data sources are named *Mobile Internet-connected Objects a*s they generate sensor data streams to be relayed through mobile devices into a data cloud, while mobile devices additionally can serve cloud-processed data of interest to users in a timely fashion. Such heterogeneous and complex environments comprising sensors, various mobile devices and the cloud, offer new opportunities for deploying innovative mobile crowd sensing applications, e.g., applications that fall within the domain of smart cities, real-time traffic monitoring, crowd sourced environmental monitoring, ambient assisted living, or social sensing.

Since sensors become pervasive while mobile devices are used to opportunistically transmit the sensed data into the cloud, the energy consumption per sensor reading becomes an important efficiency measure. This is especially relevant for sensors known to be energy-hungry, e.g., air quality sensors. In addition, mobile device battery life is one of its weakest points and thus the number of data transmissions from a mobile device into the cloud needs to be reduced. To that end, a context-aware orchestration of the sensing process and data transmission is necessary in large scale (possibly crowd sourced) IoT applications to enable energy-efficient data harvesting while maintaining an adequate level of sensing coverage. This deliverable will focus on mobility aspects of internet-connected objects for energy-efficient orchestration of sensor data harvesting and data transmission into the cloud.

OpenIoT deliverable D3.4.1 corresponds to the open source implementation of the mobile broker component in standard Java API, while the second version of this deliverable (i.e. D3.4.2) will include the full implementation of mobile broker component and related application for Android devices. This deliverable also gives an overview of a prototype implementation of our publish/subscribe middleware for mobile internet-connected objects. The prototype implementation of the deliverable is available at the Github infrastructure which is devoted to the project and which is available at: https://github.com/OpenIotOrg/openiot.

In contrast to existing OpenIoT components which comprise the OpenIoT platform, this deliverable brings in new functionality to the original OpenIoT Description of Work (DoW) through the addition of a new project partner (UNIZG-FER) which focuses on IoT scenarios integrating mobile users and crowd sensing since mobile ICOs was not the primary focus of the original OpenIoT DoW.

1.2 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 OpenIoT publish/subscribe middleware for mobile internet-connected objects. For these members the deliverable could serve as a valuable guide for the installation, deployment, integration and use of the middleware.
- **The IoT open source community,** which should view the present deliverable as a middleware solution for energy-efficient and mobility-aware data collection and dissemination in environments with mobile internet-connected objects. 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 components that comprise the first release of this middleware.
- **IoT researchers at large,** who could find in this deliverable a practical guide on the main elements/components that comprise a non-trivial publish/subscribe middleware for mobile internet-connected objects.

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

1.3 Summary

This deliverable is a prototype implementation and accompanying report of publish/subscribe middleware for mobile internet-connected objects which enables energy-efficient and mobility-aware data collection and dissemination in environments where mobile devices are used as gateways between mobile internet-connected objects and the cloud.

1.4 Structure

In this deliverable we focus on publish/subscribe middleware for mobile internet-connected objects. In Section 2, we present the communication model that is supported by the middleware. Section 3 explains the communication and implementation details of the mobile broker component. The mobile broker API specification is given in Section 4. Initial measurements of the energy consumption for three potential communication solutions are given in Section 5. Finally, section 6 surveys the related crowd sourcing platforms and compares them with our publish/subscribe middleware, while Section 0 concludes the document.

2 ENERGY-EFFICIENT COMMUNICATION MODEL FOR MOBILE ENVIRONMENTS

2.1 Overview

A publish/subscribe middleware offers the mechanisms to deal with the challenges related to context-aware and energy-efficient acquisition and filtering of sensor data in mobile environments. It provides the means for selective acquisition of sensor data from mobile wearable sensors as well as filtering of sensor data on mobile devices prior to its delivery into the cloud for further processing. This section presents a publish/subscribe middleware system named Mobile Publish/Subscribe (MoPS) that enables selective sensor data acquisition and filtering while being tailored to the requirements of mobile and resource-constrained IoT environments.

MoPS can be used in IoT environments where mobile devices are applied as gateways for collecting and transmitting sensor data into the cloud, while at the same time mobile devices receive the data of interest from the cloud. In contrast to existing centralized database solutions that typically send all sensed data into the cloud, MoPS supports *flexible and controllable acquisition of data* and its subsequent transmission into the cloud only in situations when the sensed data is indeed required by the application. In other words, the data should be produced and transmitted to the back-end systems only if it is valuable, e.g., there is current interest by system users to be alerted about certain events, or the data is needed for future data-mining tasks.

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

2.2 MoPS Communication Model

The MoPS model comprises a set of publishers, P_i, and a set of subscribers, S_j, that interact over a hierarchical two-tier publish/subscribe network composed of *mobile brokers*, MB_k, and cloud brokers, B_l. An example model is shown in Figure **1**. Publishers, e.g., wearable or built-in sensors, *publish* data items and send them either to mobile brokers or directly to cloud brokers. Subscribers, e.g. processes on mobile devices, can activate and dismiss subscriptions by sending messages *subscribe* and *unsubscribe* to mobile or cloud brokers, who in turn use the message *notify* for push-style delivery of matching data items, i.e., items that satisfy subscription constraints, to subscriber processes.

Brokers are responsible for efficient matching of data items to active subscriptions as well as their subsequent delivery to either subscribers, mobile brokers, or other cloud brokers, i.e., components that have defined matching subscriptions. Cloud brokers share the processing load to enable scalable system performance, and exchange subscriptions as well as matching data items. Subscribers and mobile brokers may

connect to any cloud broker, or the cloud-based implementation may be centralized such that the initial connection to the cloud is established through a single proxy broker.

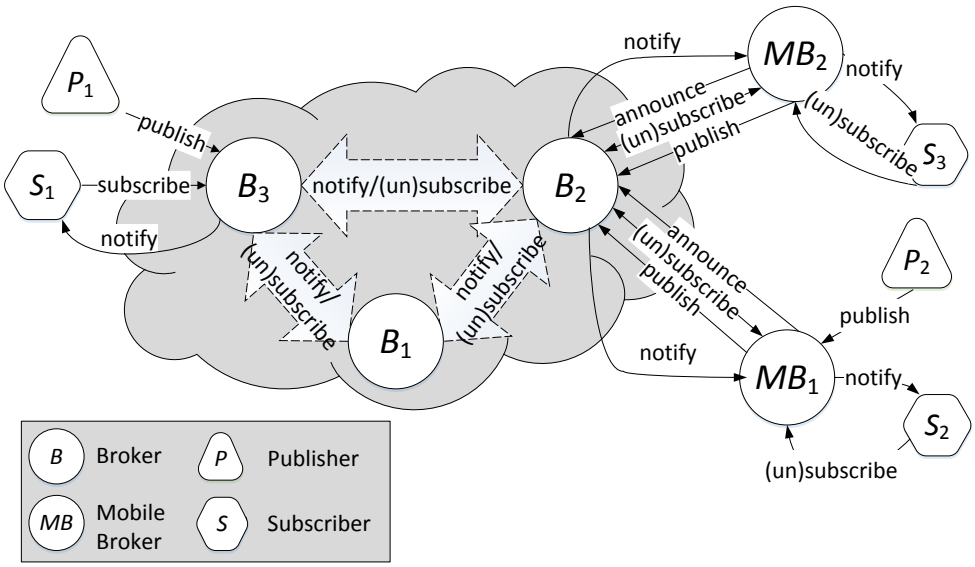

Figure 1 Publish/subscribe model and interaction

The main novelty of the MoPS model compared to existing publish/subscribe solutions is the implementation of mobile brokers running on mobile devices such as smartphones and tablets. After their initial registration with cloud brokers, mobile brokers can *announce* the type of data for publishers which they represent. For example, P_2 in Figure 1 is, e.g., a wearable gas sensor detecting levels of nitrogen dioxide (NO2) and ozone (O3). After MB_1 detects P_2 because they exchange signalling information over a Bluetooth connection, MB_1 can define the type of data items to transmit to its cloud broker B_2 in the future. MB_1 sends a message *announce* (NO2,O3,*x*,*y*), where *x*=45.81302 and *y*=15.97781 represent MB_1's current geographical latitude and longitude. The reason for creating the announce message is the following: We need to activate subscriptions from cloud brokers on MB_1, but only those that can potentially match future publications created by P_2. Obviously, as it is not desirable to activate all subscriptions from the cloud on a single mobile device, the announce message is compared to existing subscriptions on B_2. For example, B_2 identifies subscription s_i=[NO2 > 40μgm^{-3} AND 45.81<lat<45.82 AND 15.96<long<15.98] as a subscription potentially matching future publications of P_2. Thus, B_2 sends a message *subscribe* to activate s_i on MB_1. Further on, MB_1 publishes P_2's data items into the cloud, but only those that match s_i.

Selective and flexible data acquisition

By reusing the inherent features of the two-tier publish/subscribe model, we provide a flexible mechanism to control the sensing density over a predefined area covered by traces of mobile internet-connected objects (MIOs). It requires an orchestration of the sensing process with activation of adequate subscriptions on mobile brokers, as instructed by the back-end cloud system based on the integrated crowd sensed data.

If we assume the density demand is predefined for an area as required by the application logic, MIOs residing in this area during a certain time interval can be instructed either

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

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

This is the main mechanism for frequency reduction of data transmissions from MIOs into the cloud which has the potential to greatly reduce energy consumption on MIOs. Consider the following example in Figure 2. It depicts movement traces for three MIOs m_1, m_2 and m_3 within a certain area, and denotes time intervals [t_{11}, t_{12}] and [t_{21}, t_{22}] within which the two MIOs perform data transmissions into the cloud (they are marked by the symbol ×), while m_3 does not perform any transmissions. MIOs perform transmissions at marked places because during the two time intervals subscriptions matching the data acquired by m_1 and m_2 are active on those MIOs. This does not impose any constraints on the sensing process as it largely depends on MIO interaction with sensors in its vicinity. For example, if the sensing process is pull-based, an MIO can invoke it periodically during the subscription activity periods. If sensors are configured to perform periodic sensing, mobile brokers residing on MIOs will ignore the sensed data while it does not match any of the active subscriptions.

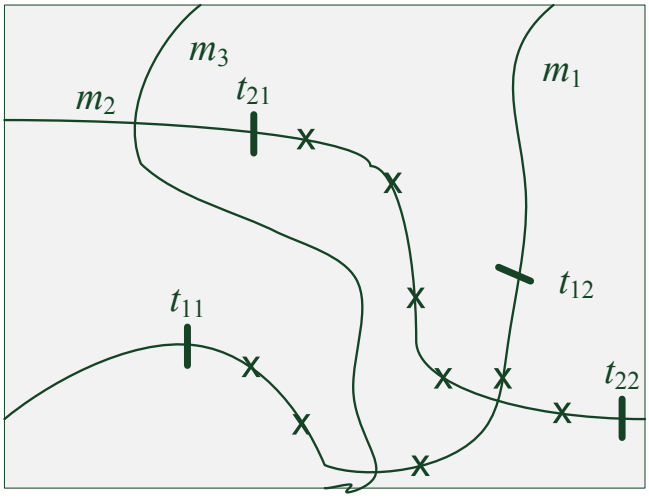

Figure 2 Movement traces and data transmissions

Let us further explain who controls the activation of subscriptions on MIOs and how the sensing density is controlled. The back-end cloud system is notified when an MIO enters the depicted geographical area since MIOs are configured to announce their available data sources when entering the area. This requires periodic GPS positioning on MIOs which is potentially energy-greedy, but if we assume that crowd sensing applications are used for a limited period of time, this process should not represent a major obstacle for application adoption. In addition, mobile devices need to be aware of area boundaries that are important for the application logic.

Since the back-end system is aware of the data samples already acquired over an observed area, it can decide whether to ship matching subscriptions to MIOs or not. In our example in Figure 2, the application logic has decided that there are sufficient measurements acquired for the depicted area from m_1 and m_2, and thus subscriptions were not forwarded to m_3.

2.3 MoPS Architecture

Mobile broker is a MoPS 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 *Cloud-based Publish/Subscribe Processing Engine* (CPSP Engine) and their "local" publishers and subscribers, as shown in Figure 3. CPSP engine is used as central processing unit for mobile broker entities. In addition to main tasks described in the deliverable D4.5.1, CPSP engine also implements subscriptions delivery to mobile brokers of their interest. In this section we present the interaction of the mobile broker with its environment, respectively with CPSP engine and its 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.

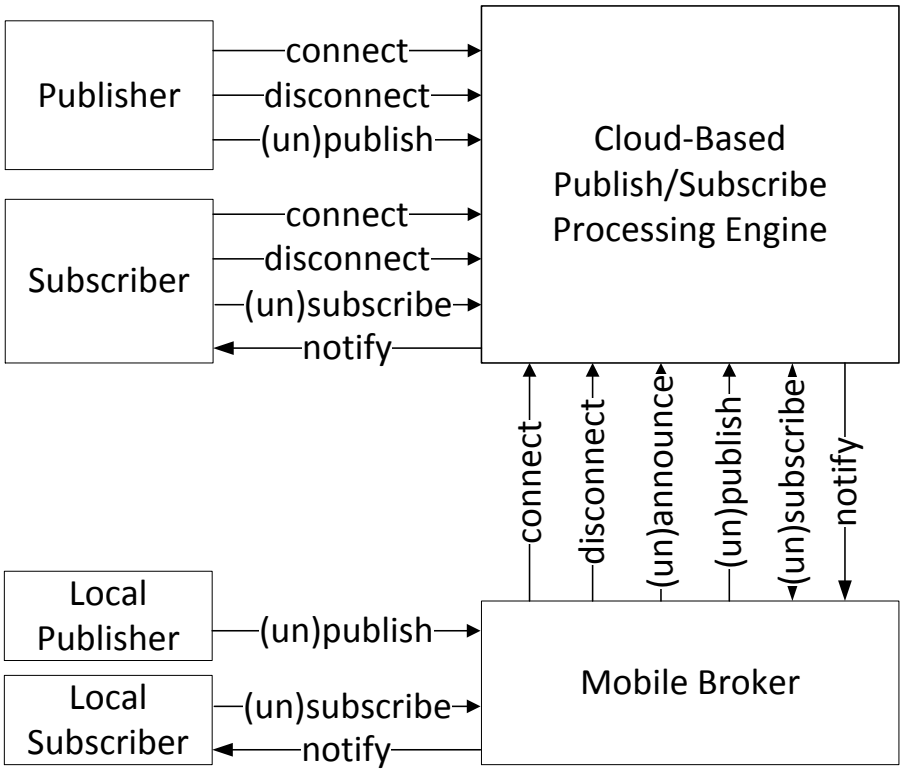

Figure 3 Interaction of mobile broker with the publish/subscribe system

Mobile brokers are a special type of broker 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.

As already stated, mobile broker serves as a proxy for local publishers and subscribers and handles all connections and disconnections from the CPSP engine. It is used to publish and delete 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. Mobile broker can start or stop the corresponding sensor 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 local publishers.

3 DATA PROCESSING ENGINE DESIGNED FOR MOBILE DEVICES

3.1 Mobile Broker Architecture

In this section we present the architecture of an Android application which serves as a gateway for locally connected MIOs, and additionally allows presenting of real-time sensor readings collected from other MIOs in the system. This application includes the functionality of previously explained mobile broker, as well as local subscriber and publisher entities. It performs the following tasks:

1. receives publications (i.e. sensor readings) from locally connected MIOs
2. manages local and received subscriptions
3. matches received subscriptions to publications
4. forwards matching publications to the CPSP engine
5. receives matching publications for local subscriptions from the CPSP engine
6. displays publications received from other MIOs
7. controls the locally connected MIOs

Figure **4** depicts the architecture of the Android application. This application consists of two background services (i.e. of the sensor and mobile broker services), GUI for controlling the services and presenting the live data, and the mobile broker component.

The main task of the sensor service is to get sensor readings from connected MIOs. It serves as a publisher which is connected to the mobile broker service. In practice, the communication between the sensor and mobile broker service is implemented through the Android internal intent broadcasting and filtering mechanism. This is the standard Android mechanism for both the intra-application communication between different application components and inter-application communication between the different applications. The communication between an MIO and the sensor service is based on the Bluetooth communication protocol.

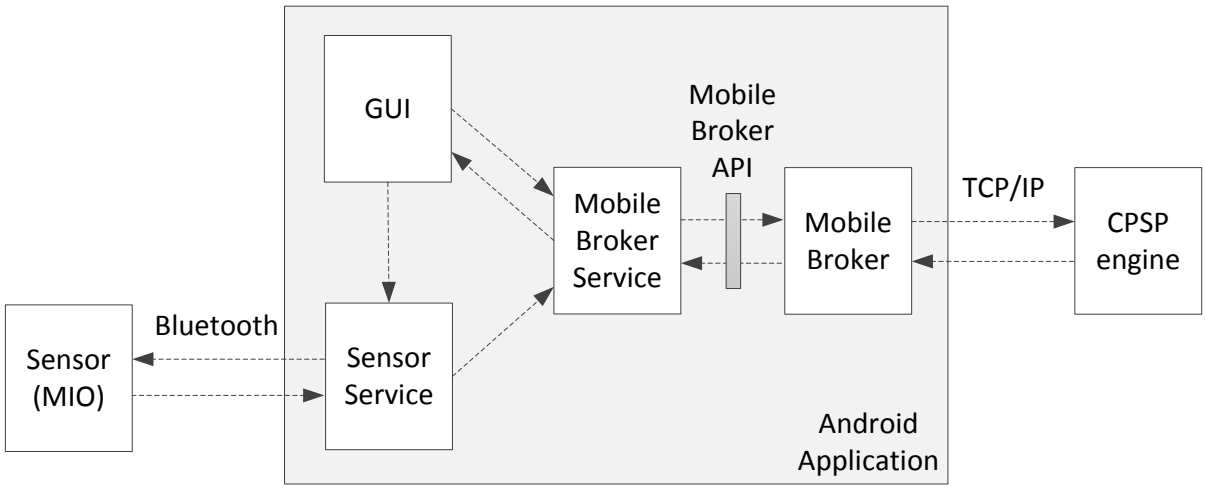

Figure 4 Android application architecture and interaction

The mobile broker service communicates with the mobile broker using the mobile broker API. It runs in the background and updates GUI elements when necessary (e.g. when a new publication is received and should be displayed to the user) and also forwards publications from locally connected MIOs.

The mobile broker component implements the core mobile broker functionality from our MoPS communication model: it communicates with the CPSP engine when needed and also does the required subscription management and matching. The communication between the mobile broker service and CPSP engine is done using the TCP/IP protocol and one of the three supported communication solutions: 1) persistent TCP connection, 2) connection-less communication over HTTP where a REST web service is running on a mobile phone, and 3) REST web service with Google Cloud Messaging. These solutions are explained in detail in Section 3.3. The mobile broker component is implemented in the standard Java API which allows the running of a mobile broker on a desktop computer. For example, this API can be used for publishing of Twitter streams from a desktop computer to the cloud, etc.

3.1.1 Communication with the CPSP engine

Figure **5** illustrates the communication between the Android application components and CPSP engine. The mobile broker takes care of the communication between the Android application and CPSP engine. Each call of a mobile broker method will produce the corresponding message which is transmitted to the CPSP engine.

When the client application is starting, the mobile broker sends a `Connect` message to the CPSP engine. In this message the CPSP engine gets the information about the mobile broker, such as its ID, name, IP address, port, etc. The CPSP engine then creates an object in its memory that represents the mobile broker.

When new data source (e.g. sensor node) is detected, the mobile broker sends an `Announce` message to the CPSP engine with the information about the data it can publish. The CPSP engine then checks all currently active subscriptions to find those that will potentially cover the publications that the mobile broker is going to publish in the future, and sends those subscriptions to the corresponding mobile brokers. The mobile broker stores the received subscriptions locally on a mobile device. If the response to the `Announce` message does not contain any active subscriptions, the mobile broker will not require air quality measurements from its locally connected sensor node (i.e. MIO). If the response message contains at least one active subscription, or the data source can generate data that will in the future satisfy conditions of the subscription, the mobile broker will initiate measurement process that periodically collects data from the sensor node. When the mobile broker receives air quality readings from the sensor node, it will first check whether the received information satisfies the requirements defined by at least one currently active subscription received from the CPSP engine in the previous step, and if not, the mobile broker will not forward the received publication to it.

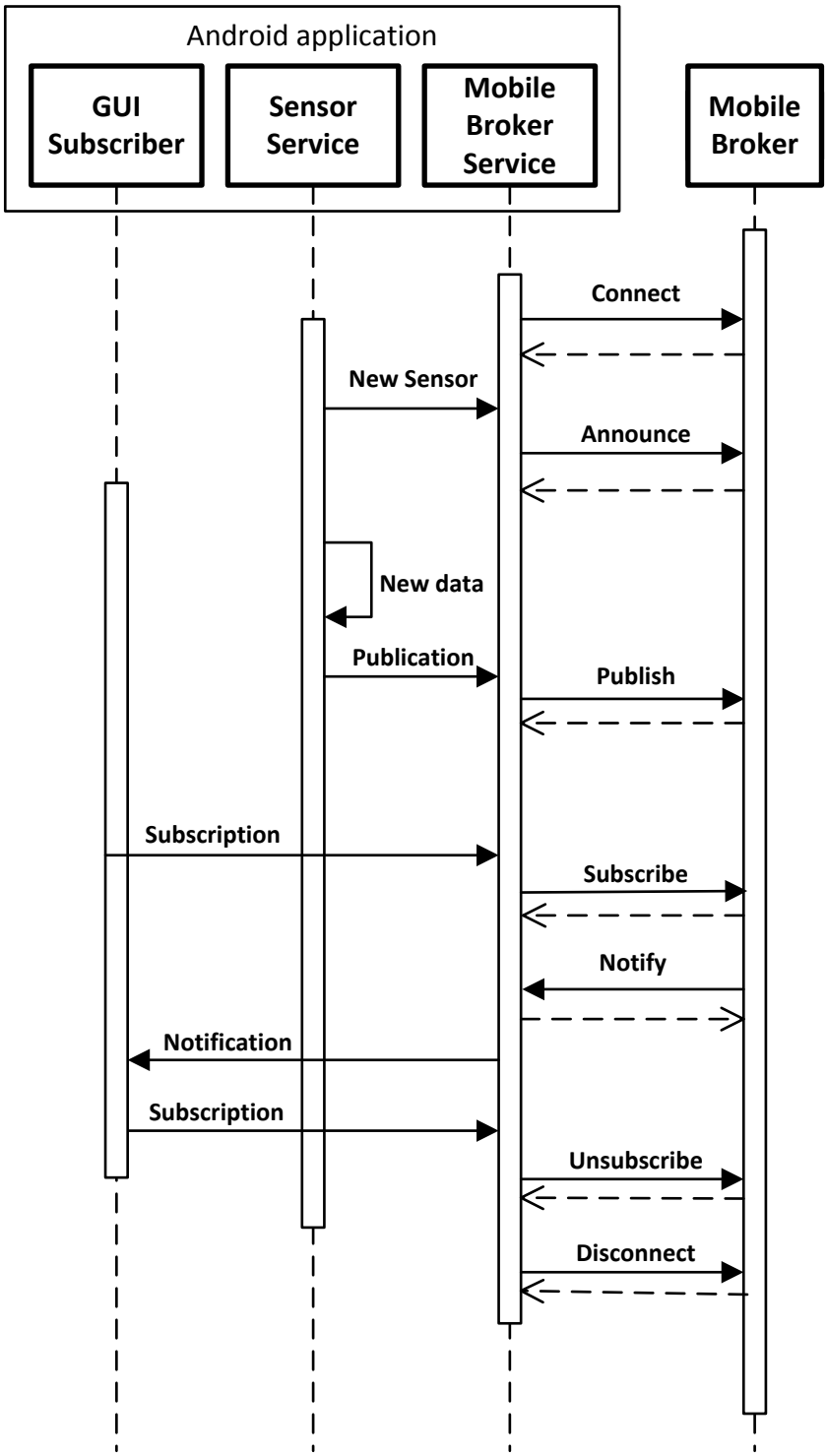

Figure 5 Communication between Android application components

When a user defines a desired subscription using the Android application, the mobile broker will then forward it to the CPSP engine inside of a `Subscribe` message. When the user wants to cancel an existing subscription, the mobile broker sends an `Unsubscribe` message to the CPSP engine.

When new sensor data is available on the client application, the CPSP engine will receive a `Publish` message. If the publication covers some of the currently active subscriptions on a stationary broker, the broker will send a `Notify` message to the corresponding subscribers.

At the end, the client application sends a `Disconnect` message to the cloud broker.

3.1.2 Communication with mobile sensor node

Any android mobile device can act as a publisher. To enable this functionality the smartphone has to be connected with a mobile wireless sensor node via the Bluetooth protocol, where the smartphone (i.e. the sensor service) has a role of the client, and the mobile sensor node acts as a server. The communication between the smartphone and the mobile sensor node is illustrated in Figure **6**.

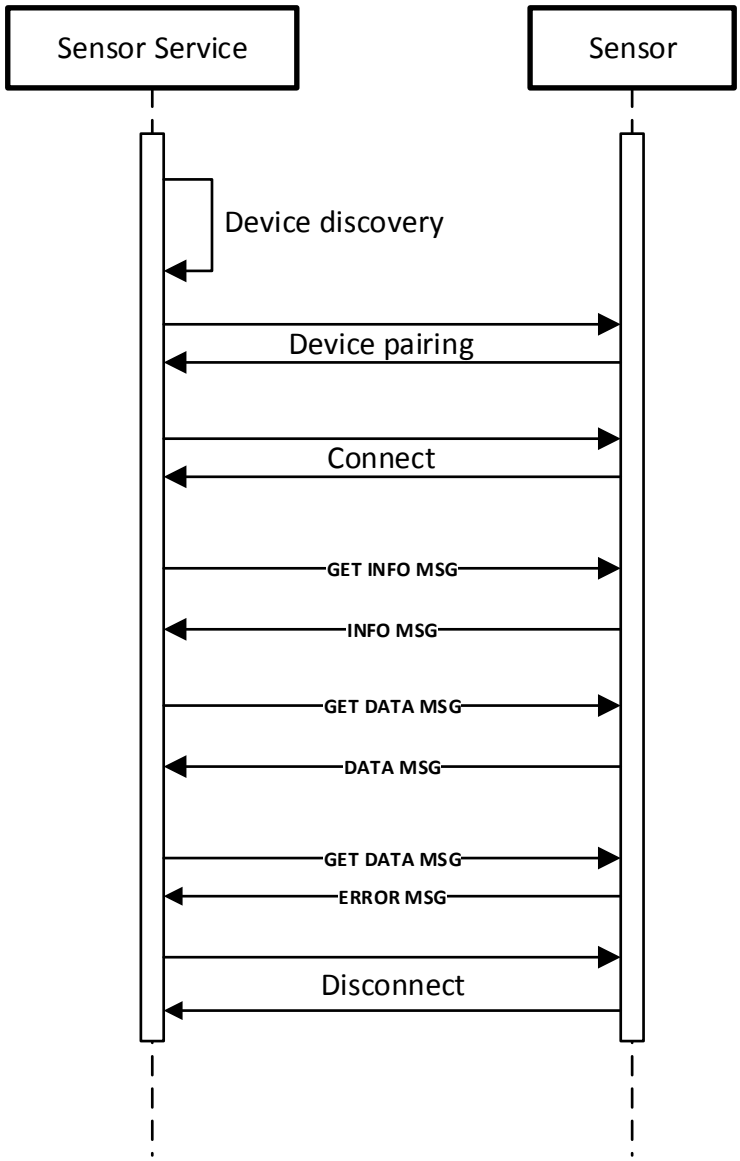

Figure 6 Communication between smartphone and sensor node

To get air quality information from a sensor node, the sensor service first has to start the discovering process (*Device discovery*) which identifies all nearby devices with an enabled Bluetooth module, and then requires information about discovered sensor node. The detection method can discover only visible devices. If a sensor node can be discovered, it will respond with the information about itself, such as the device name, class and the MAC address of a device. Using this information it is possible to establish a connection with the sensor device.

Before connecting to the sensor, the smartphone and sensor node have to be paired. There is a difference between paired and connected devices. If devices are paired it means they are aware of the existence of one another, they have exchanged the key that can be used for authentication, and they can make a secure connection. The devices are connected if they share the RFCOMM channel (*Radio Frequency Communication*, also known as SPP, *Serial Port Profile*) that can exchange data. The condition that must be fulfilled for pairing and connecting smartphone with the mobile sensor node is that the name of the Bluetooth module of a mobile device is "Blue-MSPDemo####", where "####" indicates four hexadecimal digits.

After successfully connecting with a sensor device the sensor service sends GET INFO MSG to a mobile sensor node to ask the sensor which sensing components are currently installed and functional. On the GET INFO MSG the message sensor node answers with an INFO message which contains the identifiers of the functional sensing components that a sensor node currently has installed.

A user can choose which information he wants to receive from the sensor node. After selecting the desired sensing information, the mobile broker sends an announce message to the CPSP engine with information about the data it can produce. As already stated, with the announcement message we reach all active subscriptions that will potentially cover future publications from a sensor node.

Now, the user can start the process of periodically requesting air quality information from a mobile sensor node. The sensor node begins with the air quality readings after receiving a GET DATA message from the sensor service. The GET DATA message contains the identifiers of all requested data. A request for a sensor reading will be launched only if there is an interest for those reading, i.e. if in the previous step at least one active subscription was fetched.

After completing the reading process from a mobile sensor node, which can take up to 30 seconds, the sensor node sends a DATA message to the sensor service, which consists of the sensing component identifier and the measured value. A user can access the current sensor readings on his smartphone. If the sensor data covers some active subscription, it will be expanded with the current location of the user, and published, i.e. the CPSP engine will be notified about it.

In case that an error occurs in one of the sensing components during the sensor reading process, the mobile sensor node will respond on received DATA message with ERROR message that contains the identifier of the occurred error.

The sensor service and the sensor node communicate with messages whose general form is:

MESSAGE_ID	BODY_SIZE	BODY

The message identifier (*MESSAGE_ID*) and the (*BODY_SIZE*) is 1 byte long. The rest of the message contains the message body (*BODY*), e.g. sensing component identifier, measured value, etc. Some messages may not have a body, and consist only of a message identifier.

GET INFO message form:

MESSAGE_ID=0x67

INFO message form:

MESSAGE_ID=0x66	size of message body	sensing components identifiers

GET DATA message form:

MESSAGE_ID=0x65	size of message body	sensing components identifiers

The message body of an INFO and GET DATA message consists of the sensing component identifiers, where each identifier has a size of 1 byte.

DATA message form:

MESSAGE_ID=0x64	size of message body	sensing component identifier + value

The message body contains a sensing component identifier (1 byte) and the measured value (rest of the message).

ERROR message form:

MESSAGE_ID=0x60	error identifier

The message body contains an error identifier which has a size of 1 byte.

3.2 Mobile Broker Interaction

Connect and disconnect. The two methods are used by the mobile brokers and also by the subscribers and publishers which are connected directly to CPSP engine. The method *connect* adds subscriber/publisher/mobile broker identifier into the list of connected components maintained by the CPSP engine, while the method *disconnect* removes them from the list. In case a subscriber or mobile broker reconnects to the CPSP engine, the engine first delivers all publications that have been matched to their active subscriptions while they were disconnected.

Publish. The mobile broker (MB) is an intermediate between the MIO and the CPSP engine. When it receives a publish message from the MIO, it will forward the message to the CPSP engine only if it previously received a matching subscription

from the CPSP engine. Otherwise, the MB retains the publication, in which no one is obviously interested. The MB stores all received subscriptions through *subscribe* requests to a list of active subscriptions. Each *publish* request from an MIO is subsequently matched to the list of stored subscriptions. The matching process identifies whether a publication should be forwarded to the CPSP engine or not. A sequence diagram depicting the delivery of a new publication to an interested subscriber is shown in Figure **7**.

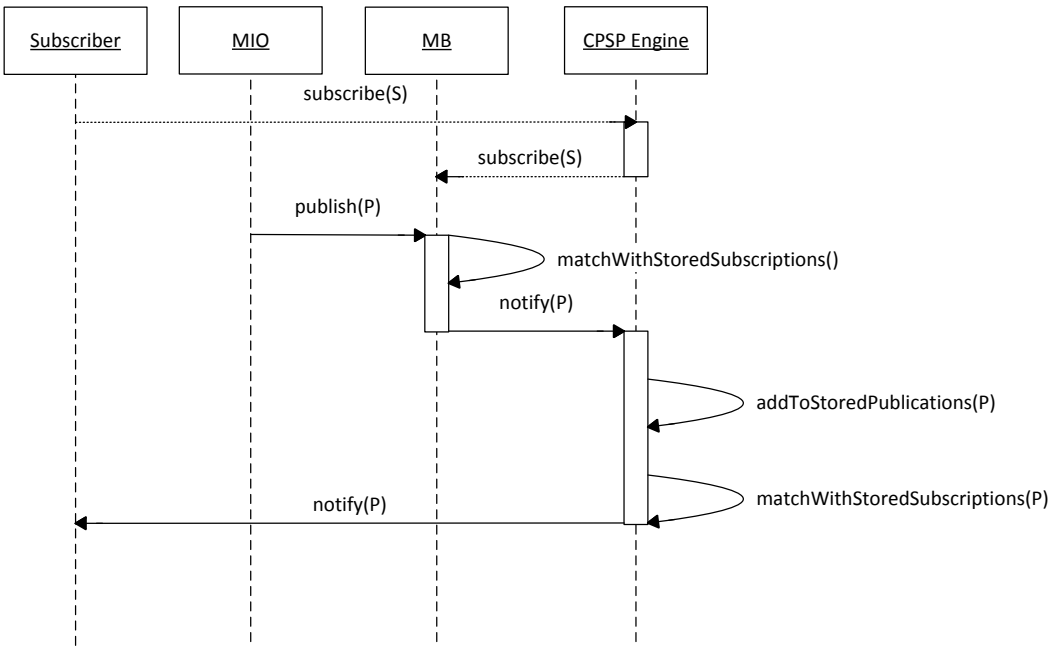

Figure 7 Delivery of a new publication

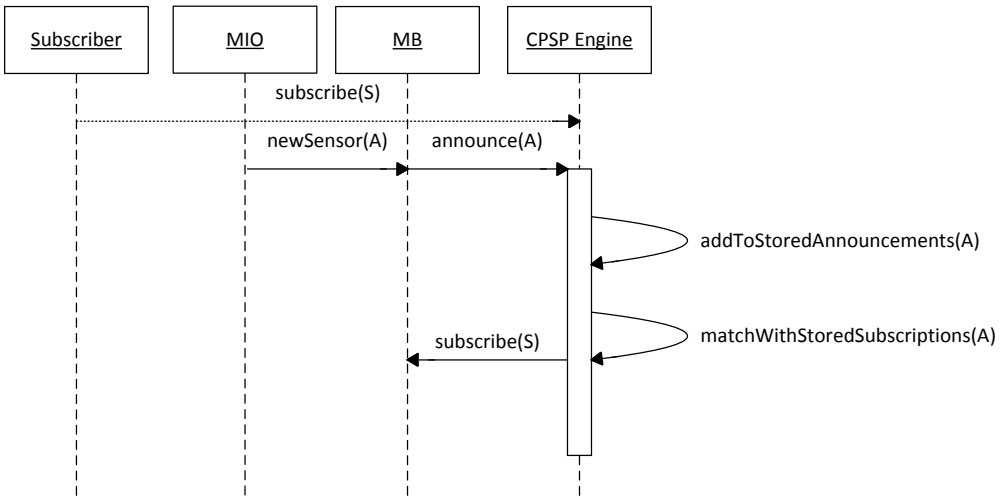

Figure 8 Announcing a new publisher

Announce. Figure **8** shows the sequence of events following a new *announce* event. When an MB receives a new sensor announcement from one of its MIOs, it announces a new publisher to the CPSP engine by sending the corresponding announce message to it. The CPSP engine then stores the announcement in a list of stored announcements and compares it with the list of stored active subscriptions. If there are interested subscribers with subscriptions matching the announcement, it may activate the publisher by forwarding the matching subscriptions to the MB in the corresponding subscribe message.

Unannounced. Any publisher connected to a mobile broker can revoke its previous announcement. As shown in Figure **9**, when this happens, the CPSP engine just needs to delete the announcement from its list of stored announcements. The revoke event can be initiated by the MB (e.g. to save the battery when its level is low) or MIO.

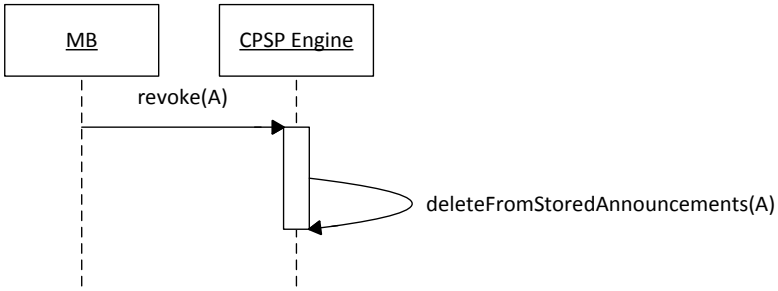

Figure 9 Revoking an announcement

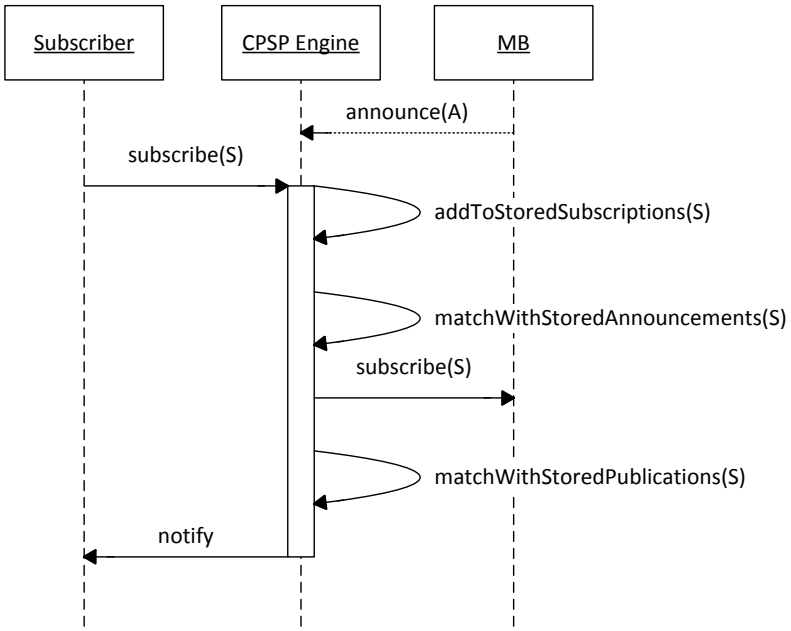

Figure 10 Activating a new subscription

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 **10**, 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 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.

Unsubscribe. Clients usually unsubscribe when they 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 “unsubscribe" message, as shown in Figure **11**.

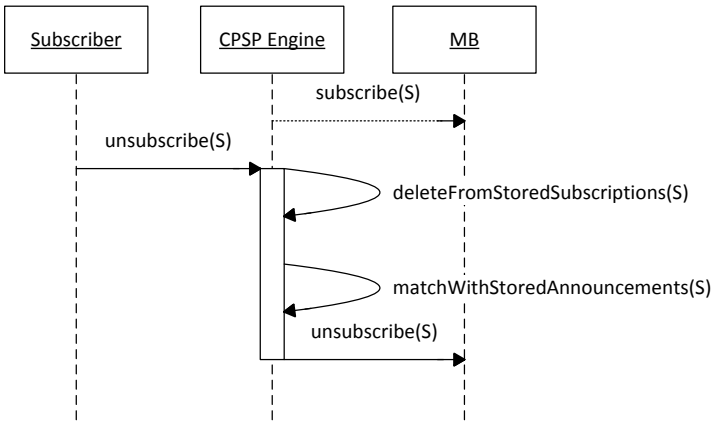

Figure 11 Cancelling a subscription

3.3 Potential Communication Solutions with CPSP engine

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

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

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

Connectionless REST-based communication between a mobile device and the cloud is an alternative to permanent TCP connections. Both the mobile device and server need to run a REST service: Whenever they want to communicate, they send HTTP messages to the REST service entry-point. In comparison to TCP connections, this mechanism is one step closer to push-based communication where situations of temporary connection losses and failed handover do not affect the communication mechanism.

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

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

3.4 Mobile Broker Data Processing Engine

3.4.1 Active subscription forest

The active subscription forest is a data structure which stores active subscriptions from end-users of the publish/subscribe middleware. Subscription forest is formed based on the subscription covering relationship which enables partial subscription ordering [Tarkoma2006]. It can reduce the matching time for comparing an incoming publications with a large number of subscriptions in order to detect a list of subscribers requiring a notification about the publication. Such a data structure is a key prerequisite for the implementation of energy-efficient and mobility-aware data collection on mobile devices.

Similar to the subscription forest data structure in the CPSP engine defined in deliverable D4.5.1, an active subscription forest is also used by a mobile broker for comparing publications with a set of subscriptions, but in this case such functionality is necessary to determine whether a newly created publication should be sent to the CPSP engine (i.e. if any user is interested in published data) or not. The objective of

the *matching* algorithm is to efficiently determine whether to disseminate the publication through the middleware, i.e., to the CPSP engine.

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 subscription and publication parameters. This is achieved by introducing triplets, i.e. each parameter is defined by a triplet: an object, value and operator. By this structure, each parameter can be compared to any other, and easily can be determined whether a subscription is covering a publication.

A simple example of a subscription data structure is shown in Figure **12**. 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 a partially ordered set, where each node covers all of their children nodes (i.e. a subscription of a parent node is more general than a child's subscription), and may have more children, but only one parent. The process of building the forest is incremental, so whenever a node is added or removed from the forest, all relationships between the rest of the forest is maintained.

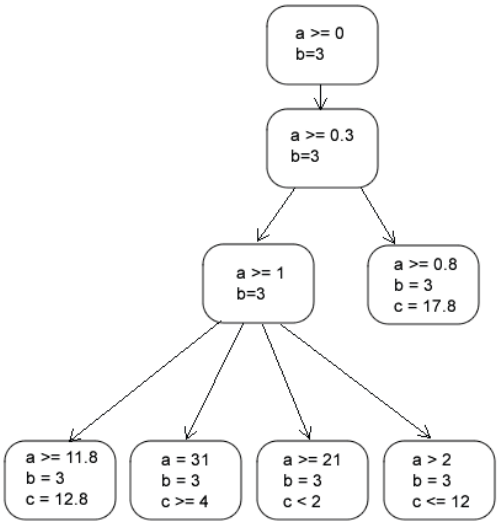

Figure 12 An example of a Boolean subscription forest

The processing of incoming publications is efficient since the algorithm can very quickly identify whether a forest is covering a new publication. In matching algorithm only the root nodes are processed, because if the root nodes do not cover the publication, nobody is interested in the published data and the rest of the forest can be neglected. If any of root nodes is covering a publication, the publication is then forwarded to the CPSP engine because obviously someone is interested in the publication data.

3.4.2 Active mobile Internet connected objects

The second part of enabling an energy-efficient and mobility-aware data collection is the management of connected data sources (i.e. connected MIOs). The management of connected data sources provides a possibility to a mobile broker to turn off the data source or change its state to the sleep mode.

When a new data source (e.g. sensor node) is detected, the mobile broker sends an `Announce` message to the CPSP engine with the information about the data it can publish and stores that information to memory extended with a parameter that indicates the status of the source (i.e. a parameter indicates if the sensor is switched off or if it is in a sleep mode). In a reply to the Announce message the mobile broker stores received subscriptions locally on a mobile device (the data structure is described in Subsection 3.4.1). If the response to the Announce message does not contain any active subscriptions, the mobile broker will not require any data from its data source. If the response message contains at least one active subscription, the mobile broker will initiate data production from the data source.

On each received active subscription mobile broker re-evaluates a status parameter of sensors that are influenced by the subscription. Because of that, at any moment a mobile broker has full and accurate view of each data source and its status.

4 MOBILE BROKER API SPECIFICATION AND EXAMPLES

The current release of the OpenIoT Mobile Broker implements the functionalities/capabilities that are reflected in the Table **1**.

Table 1 Mobile Broker public methods and constructor

```
MobileBroker (name:String, brokerIP:String, brokerPort: int):
MobileBroker

---

connect(): void

connect(brokerIP:String, brokerPort:int): void

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

announce(Announcement announcement): void

revokeAnnouncement(Announcement announcement): void

publish(publication: Publication): void

unpublish(publication: Publication): 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 **2**.

Table 2 Implemented Mobile Broker API definition

Service Name	Input	Output	Info
MobileBroker (constructor)	String name, String brokerIP, int brokerPort	MobileBroker	Used to create a MobileBroker entity. Requires as input an entity name in String format, IP address and port number of the broker. Output is the MobileBroker object that will be used for subscribing on behalf of a user, receiving incoming notifications and publishing data that someone has requested.
connect		void	A method used to connect to the 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 not 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 requests. The 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 a user's subscription.
publish	Publication publication	void	Used to publish new data. The innput parameter is a valid publication.
unpublish	Publication publication	void	Used to delete previously published data. The input parameter is the publication that is removed from the cloud broker but not from the subscribers that already received the notification with the specified data.

4.1 Download, Deploy & Run

4.1.1 Developer

4.1.1.1 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.

4.1.1.2 Download

To download the Mobile 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 in the "openiot/modules/pub-sub/mobileBroker" folder.

4.1.1.3 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 Mobile 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 Mobile Broker project.
    2. Type this command to build and deploy the archive:
        - mvn clean package
    3. This will build the service. The service is ready to be started by the command `java –jar MobileBroker.jar` in the terminal.

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

4.2 Source Code Example

Table **3** shows a code snippet that creates the Mobile Broker object with the specified name and it connects to the CPSP engine which is available on the address `broker.tel.fer.hr`, port 12345. After the mobile broker 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 notifications, but it can be extended to make more complex processing (a notification listener can be defined in a separate class). After that the broker connects to the CPSP engine, defines a new subscription and subscribes with it. Next, the mobile broker creates an announcement of the available data. In the code snippet the NO_x sensor is available and ready to publish data. Then, the mobile broker tries to publish data, whereas the publication will only happen if someone is interested in its content (in this example we previously defined a subscription that covers the publication). The last lines of the code snippet show how to disconnect from a CPSP engine.

Table 3 Mobile Broker source code snippet

```
//create new mobile broker
MobileBroker broker = new MobileBroker("Sensor_example",
"broker.tel.fer.hr", 12345);

//create notification listener
broker.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
broker.connect();

//define subscription
TripletSubscription ts1 = new TripletSubscription(-1,
System.currentTimeMillis());
ts1.addPredicate(new Triplet("NOx", 5.4, Operator.GREATER_OR_EQUAL));

//send subscriptions to broker;
broker.subscribe(ts1);

//select available sensors
TripletSubscription sensors = new TripletSubscription(-1,
System.currentTimeMillis());
ts1.addPredicate(new Triplet("NOx", 0, Operator.ANY));

//define announcement
Announcement announcement = new Announcement(-1,
System.currentTimeMillis());
announcement.setAnnouncement(sensors);

//announce available data
broker.announce(announcement);


HashtablePublication hp = null;

//define new publication
hp = new HashtablePublication(-1, System.currentTimeMillis());

//set publication propertis as name-value pairs;
//each publication can contain an arbitrary number of properties
hp.setProperty("NOx", 11.2);

//publish the publication
broker.publish(hp);

//disconnect from broker
broker.disconnectFromBroker();
```


5 ENERGY-RELATED MEASUREMENTS

5.1 Energy and Bandwidth Consumption on MIOs

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

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

In first energy consumption test, the server has sent 1000 data items with an average interval of 1 second between two consecutive notify operations on a Wi-Fi interface. In this case the phone did not enter a power-save mode. Measured values of energy consumption are shown in Table **4**. The power consumption is measured in mWatts, the duration of each paradigm runtime necessary for receiving the entire data sets is expressed in seconds, and the consumed energy is expressed in Joules. All three paradigms need approximately the same time for receiving 1000 data items. The GCM paradigm is the most favourable technique for sending notifications as it consumes almost 50% of the energy required for TCP-based solution, while REST has an overhead of almost 20% compared to TCP (Figure **13**).

Table 4 Energy consumption on a Wi-Fi interface for receiving 1000 data items

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

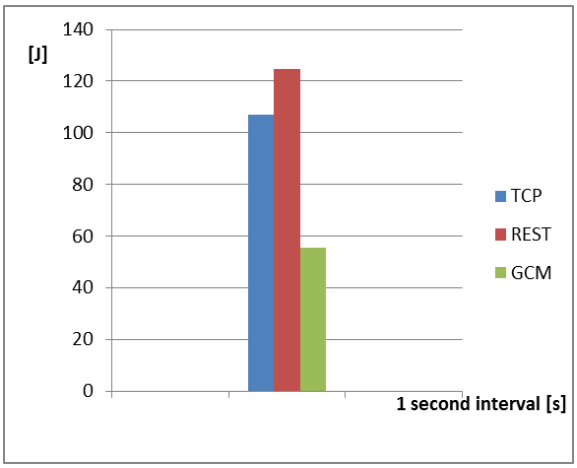

Figure 13 Energy consumption on a Wi-Fi interface for receiving 1000 data items

The second energy consumption test is done by sending 100 data items, with an average interval of 10 seconds between each notify operation. In this case the smartphone did enter a power-save mode between each receive operation. Results of the second test are shown in Table **5**. As one can notice the GCM paradigm once again has the best performance, but in this test the other two paradigms have much better results than in the first test scenario (Figure **14**). In general, the GCM service shows the best results regarding energy consumption because no additional network connections are needed while the processor can go to the power save mode.

Table 5 Energy consumption on a Wi-Fi interface for receiving 100 data items

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

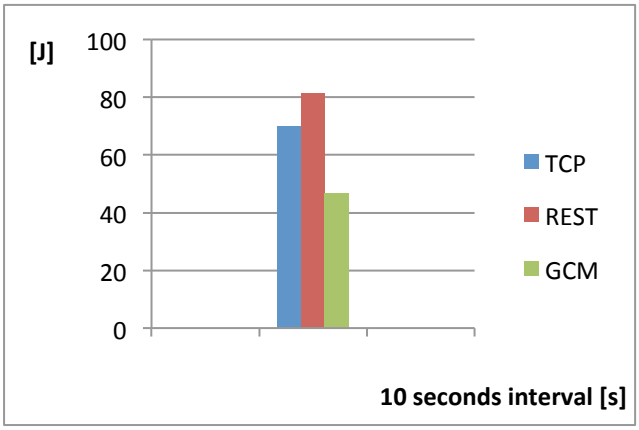

Figure 14 Energy consumption on a Wi-Fi interface for receiving 100 data items

Parallel with energy consumption tests, we also measured the bandwidth consumption with TrafficMonitor application on both Wi-Fi and 3G interface of a mobile phone. In the first bandwidth consumption test, the server sent 1000 data items on a Wi-Fi interface, with an interval of 1 second between them. The TCP-based solution generates the least amount of the traffic, and our REST-based solution generates the biggest amount of traffic (approximately 5 times bigger than TCP paradigm) as expected since entities communicate using the HTTP protocol. The TCP paradigm provides the best results because it has the least overhead. Data transferred through the GCM connection, except our data set, contains also identification of the intended recipient, and REST paradigm is used for registration on the server so generated traffic is bigger than in the TCP paradigm (Figure **15**). The REST paradigm generates much more traffic than the other two, especially in the direction to the server (i.e. upload) as shown in Table **6**. We also have made the same test on a 3G interface, where the server sent 1000 data items, with an interval of 1 second between them. Results presented in the table show that bandwidth consumption is approximately the same as when the mobile device was connected on a Wi-Fi interface, as it was expected because bandwidth consumption does not depend on the connection mode of a mobile device (Figure **15**).

Table 6 Bandwidth consumption for receiving 1000 data items

Communication paradigm	Total bandwidth [kB]	Download [kB]	Upload [kB]
TCP – Wi-Fi	264.13	256.51	7.62
REST – Wi-Fi	1402.88	1293.29	109.59
GCM – Wi-Fi	973.81	958.73	15.08
TCP – 3G	258.37	239.1	19.27
REST -3G	/	/	/
GCM -3G	812.4	738.77	73.63

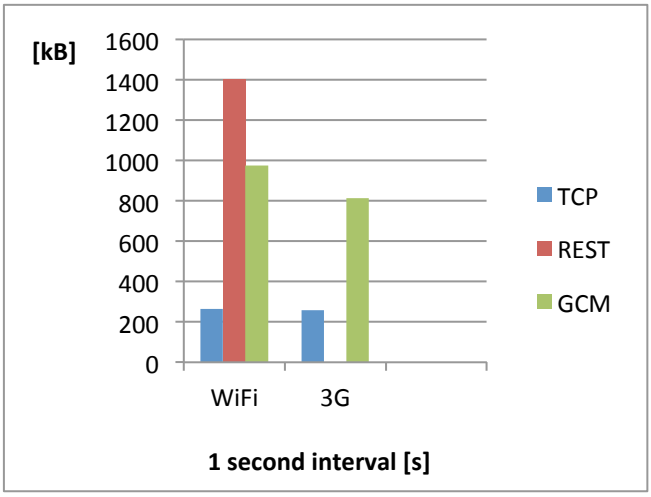

Figure 15 Bandwidth consumption for receiving 1000 data items

The second bandwidth consumption test was done by sending 100 data items on a Wi-Fi interface, with an average interval of 10 seconds between each notification. Results of the second test are shown in Table **7**. As one can notice the TCP based solution once again generated the least amount of the traffic, and REST based paradigm generated the biggest amount of data (Figure **16**). The same test scenario was made on a 3G interface of a mobile device. As it was expected, bandwidth consumption for TCP communication paradigm on a 3G interface was approximately the same as on a Wi-Fi interface of a mobile device.

Table 7 Bandwidth consumption for receiving 100 data items

Communication paradigm	Total bandwidth [kB]	Download [kB]	Upload [kB]
TCP – Wi-Fi	52.82	45.05	7.77
REST – Wi-Fi	986.77	961.29	25.48
GCM – Wi-Fi	203.24	189.67	13.57
TCP – 3G	52.53	34.21	18.32
REST – 3G	/	/	/
GCM – 3G	134.14	107.94	26.2

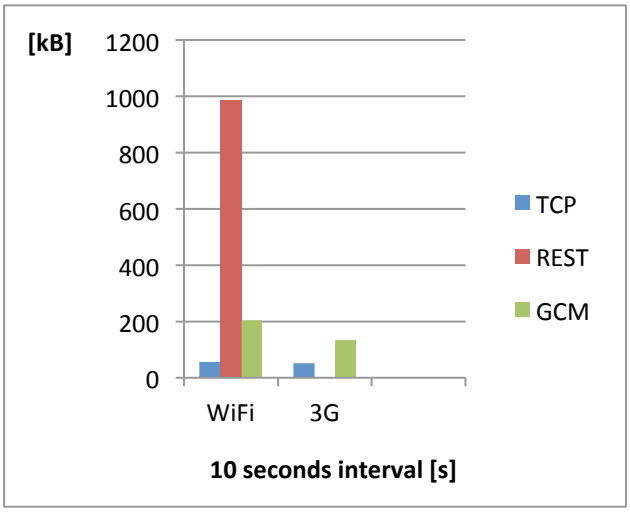

Figure 16 Bandwidth consumption for receiving 100 data items

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

5.2 Latency on MIOs

In our evaluation scenario we have also measured the latency on both wireless and 3G interface of a mobile device. We have tested the propagation time from publishing new data to receiving a notification about the same data, where between publishing and receiving, data is sent to test server. The test broker is publishing new data sequentially every 1 and 0.1 second.
The first latency test is done on a Wi-Fi interface of a mobile device. Results are shown in Table **8**. The REST paradigm has the best performance because the mobile device and test server are in the same wireless network, and there are only 2 hops between them, in contrast to the TCP and GCM paradigm where the mobile device and the test server are in different networks and there is more hops between them. The TCP and the GCM paradigm have approximately the same latency. When the interval between publishing new data is 1 second, the latency is 40 to 70 milliseconds bigger than when the interval is only 0.1 second, for both the TCP and the GCM paradigm because with a shorter interval between publishing the data, the mobile device uses all core processes, while with a bigger interval the mobile device can go in power save mode as shown in Figure **17**.

Table 8 Latency on a Wi-Fi interface

Communication paradigm	1 second interval [seconds]	0.1 second interval [seconds]
TCP	0.293737	0.254744
REST	0.06974	0.074989
GCM	0.286145	0.214721

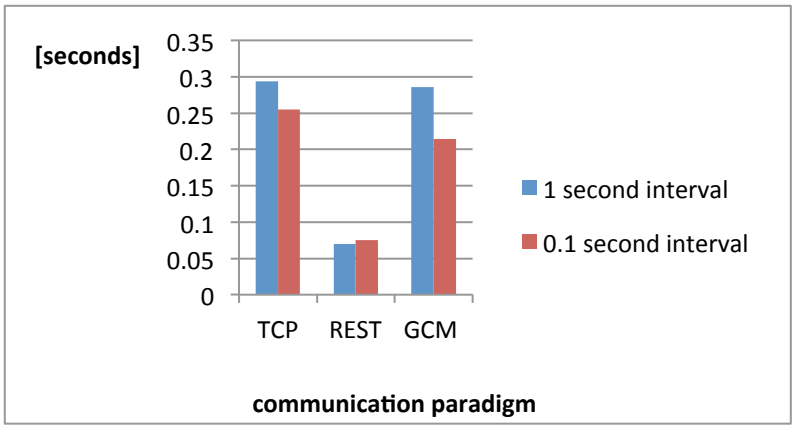

Figure 17 Latency on a Wi-Fi interface

The second latency test was done on a 3G interface of a mobile device. Results of measurements are shown in Table **9**. The TCP paradigm has approximately the same performance when data is published every 1 second as when the data is published every 0.1 second, but the REST and the GCM paradigm have much bigger latency when the data is published every 0.1 second, because when the mobile device is using a 3G network, it cannot handle all published messages as fast as

they are coming in which results in a congestion in the output queue of a mobile device (Figure **18**).

Table 9 Latency on a 3G interface

Communication paradigm	1 second interval [seconds]	0.1 second interval [seconds]
TCP	0.752121	0.612037
REST	0.391827	13.56158
GCM	0.393784	11.47304

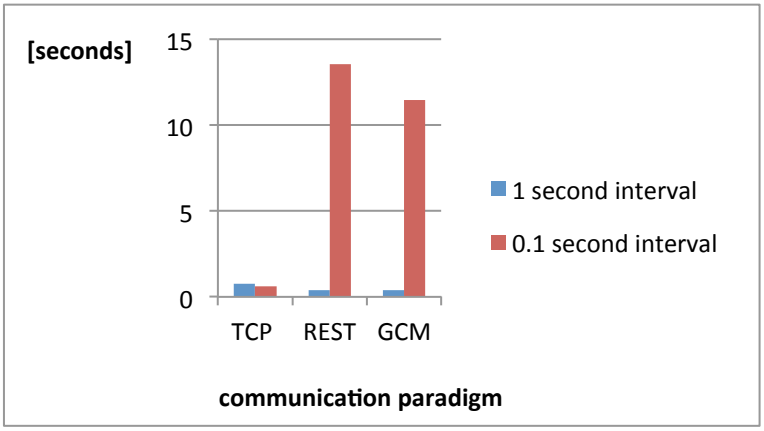

Figure 18 Latency on a 3G interface

If we compare latency for all three paradigms on Wi-Fi and 3G interface of mobile device when data is published every 1 second we can see that the latency is much bigger on a 3G interface (Figure 19). This is to be expected because there are more hops between the test server and mobile device on a 3G network than on a Wi-Fi network.

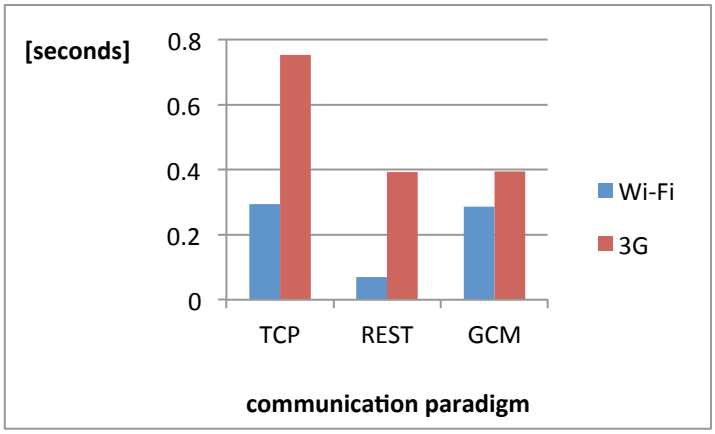

Figure 19 Comparison of latency on Wi-Fi and 3G interface when data is published every 1 second

6 RELATED CROWDSOURCING PLATFORMS

In this section, we briefly describe existing crowd sensing platforms and applications. First, we give an overview of two standardisation actions related to the area of Internet of Things (IoT) and then we focus on mobile crowd sensing applications for air quality monitoring. As a general comment on existing crowd sensing platforms, it can be noticed that they typically do not apply the publish/subscribe principles, neither for flexible data acquisition nor for sending notifications in near real-time to mobile users. Existing platforms are mostly gathering as much data as possible from mobile devices while some of them are enabling real-time alerting on mobile devices but based on local sensor data, and on cloud-generated alerts.

Protocol CoAP is being standardised by the Constrained RESTful Environments (CoRE) workgroup, and is available as an active (IETF) Internet-draft. CoAP is based on a request/response communication model between entities (e.g. a sensing node and server), including key concepts found on the Web, thus enabling easy integration with standard Web services [Shelby2013]. The CoRE workgroup aims to reuse the REST architecture of the Web as enabler of services provided by constrained nodes or environments. CoAP natively supports discovery of services or resources as well as multicast and asynchronous messaging.

The main features of CoAP are the following:

- web protocol designed for M2M requirements
- UDP binding with options for reliability, unicast or multicast
- asynchronous message exchange
- small size of a message header
- URI support
- simple mapping with HTTP
- security binding to Datagram Transport Layer Security (DTLS)

CoAP can be viewed as a composition of two sub-layers: the Messaging layer and the Request/Response layer. The Messaging layer deals with UDP and the asynchronous nature of the communication (sending an acknowledgment if requested), while the Request/Response layer is responsible for processing the method or response code. CoAP uses a four bytes header followed by various options and a payload.

CoAP implements simple semantics of a request by defining a method code while the response semantics are defined by a response code. CoAP uses the GET, PUT, POST and DELETE methods in a similar manner to HTTP, while keeping the support for adding new methods if necessary. The GET method retrieves a representation of a resource that is defined by the request URI. The POST method processes the representation of a resource enclosed in the request. Usually it means that a new resource is being created or an identified resource is updated. The PUT method requests that the resource identified by the request URI is updated or created with the enclosed representation. The DELETE method deletes the resource identified by

the request URI. Response codes are also similar to those defined for HTTP. CoAP defines three classes of response codes: a code starting with the number 2 means that the request was successfully received, understood and accepted; a code that starts with the number 4 is an answer to a request that cannot be fulfilled or has a bad syntax, while a response code starting with the number 5 means that a server cannot fulfil a valid request. Both response codes and methods can be extended by additional definitions.

CoAP uses the *coap* and *coaps* (if DTLS is used as the security layer) URI schemes for identifying CoAP resources and providing means of locating the resource. Resources are organised hierarchically and governed by a potential CoAP origin server. A generic CoAP URI has the following format:

```
coap-URI = "coap:" "//" host [ ":" port ] path-abempty [ "?" query ]
```

CoAP URI elements are in accordance with a generic URI syntax. A generic URI syntax is defined in the IETF document RFC 3986 (Berners-Lee 2005).

Service discovery by using CoAP is defined in the main document, which defines the protocol [Shelby 2013], while resource discovery on a node using CoAP is done in accordance with RFC 6690 [Shelby2012].

MQTT is a simple and lightweight messaging protocol designed for constrained devices and low-bandwidth, high-latency and unreliable networks [MQTT 2010]. It uses the publish/subscribe model of communication. The protocol minimises network bandwidth and device resource consumption while keeping the reliability and assurance of delivery. It is mainly used in M2M services. MQTT was invented by Dr Andy Stanford-Clark of IBM and Arlen Nipper of Arcom in 1999. In March 2013, OASIS started the standardization process based on the 3.1 specification of MQTT.

MQTT has a small transport overhead, since messages have a 2 byte header. MQTT has 14 message types, each indicating a request or a completion of previous request. Basic requests are the following: connect (a client requests a connection to the server), publish (publish a message), subscribe (a client expresses an interest for some information), unsubscribe (a client deletes the subscription), pingreq (ping request) and disconnect (a client disconnects from a server). Other messages indicate request completion.

In the context of the OpenIoT Project, OpenIoT leverages M2M/IoT techniques above (such as CoAP), but also adds richer semantics to the inter-object communication. Furthermore, it provides user-friendly service interfaces for people to object interactions.

The OpenIoT internal communication between sensors within a deployed network is considered as a “black box”. The input for OpenIoT sensor data delivery chain solely results from the data as provided by the gateway node. The applied communication protocols and routing algorithms to establish the multihop communication between sensor nodes themselves and between the sensor nodes and the gateway node are not within the scope of OpenIoT. Thus, the OpenIoT architecture can be mapped onto the high level M2M architecture defined by ETSI (

Figure **20**) that includes a gateway to the network domain hosting applications.

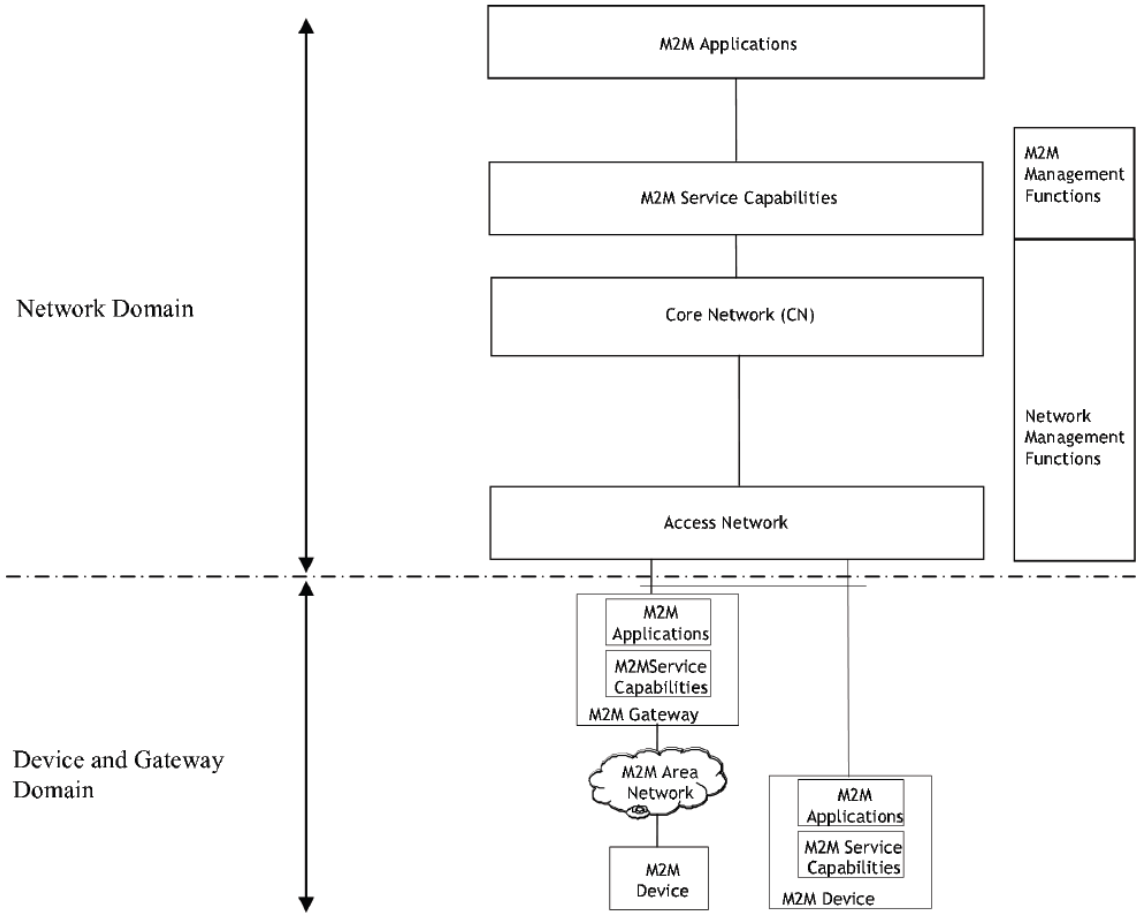

Figure 20 High-level M2M architecture (ETSI 2012)

PEIR (the Personal Environmental Impact Record) is a system which enables users to measure their personal exposure to pollution using location data sampled from their mobile phones [Mun2009]. Collected data is automatically uploaded to the server where it is processed. The PEIR system collects location records from GPS approximately every thirty seconds and then attempts to determine the most likely activity for each sample (staying, walking, driving), which affects the exposure models used later in processing on the server. They use four everyday impact and exposure metrics – carbon impact, sensitive site impact (emissions based on user transportation mode), smog exposure and fast food exposure. Every participant can access his own exposure on a web interface. The PEIR system also provides comparisons among the users in the same Facebook-based social network.

Haze Watch is a project in which mobile air pollution sensors are used to measure the concentration of carbon monoxide, ozone, sulphur dioxide, and nitrogen dioxide in the air while the user drives in the car [HazeWatch]. Measurements are sent to the smartphone via Bluetooth, and then uploaded via 3G network to the server where data is analysed and stored in a database. The database system consists of the web server layer, model layer and database layer. Aggregate readings from all participants can be displayed to the users on Google maps on the web interface, and

individual measurements can also be displayed on a user's mobile phone (iPhone). The application shows air pollution for individual's daily movements based on GPS readings and graphically displays exposure over time identifying regions of heavy pollution on user's daily routine.

OpenSense is an open platform for air pollution monitoring which deploys environmental sensors on buses [Yan2012]. The goal is to provide air pollution map from limited number of sensor readings and the challenge is to define an adequate sampling frequency over bus routes for adequate sensing coverage.

CAROMM, context-aware open mobile miner, is a middleware platform designed to acquire sensor data from mobile devices with a goal of reducing the amount of data or the number of data transmissions to be sent into the cloud [CAROMM]. CAROMM uses a data mining approach on mobile devices to compress raw data in clustered data which shows results in terms of battery consumption.

AirProbe is an application that collects concentrations of pollutants in the air from portable sensor and localizes them through GPS [AirProbe]. SensorBox is a small measuring system with different low-cost gas sensors which are calibrated against certified instruments detecting ultrafine particle (UFP) and Black Carbon, but reliability of measurements is affected by various environmental factors like wind, weather conditions, etc. Application and SensorBox are connected via Bluetooth. Further, the data is sent to the EveryAware server which collects data from all participants in real time. Users can access all acquired data through web applications.

CommonSense is a distributed air quality monitoring system in which a mobile phone collects environmental data (carbon monoxide, ozone, and nitrogen dioxide as well as light, temperature, humidity and orientation) from a sensor via the Bluetooth interface and sends all data with GPS space-time stamps to the database server using GPRS radio [Dutta2009]. Sensor data can be collected periodically, randomly or non-uniformly. Users can access exposure to pollutants through their mobile phones or on the web portal.

GasMobile is an air quality measurement system that contains low-power and low-cost sensors connected on a mobile phone via USB host mode and a server [Hasenfratz2012]. A user can start the sensor calibration, take measurements or upload the data on the server. The GasMobile system aims to provide high data accuracy by utilizing sensor readings near static governmental measurement stations to keep a sensor calibration up to date.

CitiSense is a system which collects and analyses real-time data, and visualizes the data on a web interface [Ziftci2012]. A small sensor device measures ozone, carbon monoxide and temperature and transmits the data to the Android phone. The mobile phone filters the data and provides real-time feedback to the user. All data is sent to the backend cyber-infrastructure where further analyses are performed over the data. Users can on their mobile phones or on web interface access information about the pollutants they are exposed to during the day.

Cambridge Mobile Urban Sensing (CamMobSense) is a project of the University of Cambridge, where the air quality is measured from mobile sensor nodes and stationary sensor nodes that are attached to a street lamp [CamMobSens]. Measurements from mobile sensor nodes are transferred to Nokia mobile phones via the Bluetooth protocol and stored to the database via GPRS. Newer mobile sensor nodes include GPRS and can store measurements in the database without a mobile phone. All measured data can be visualized with Google Earth.

Mobile Air Quality Monitoring Network (MAQUMON) is a system which uses mobile sensor nodes attached to cars for measuring the amount of various pollutants in the air [MAQUMON]. Measurements with timestamp and GPS location are periodically sent to the server where data is processed and published on the Microsoft SensorMap portal. Mobile sensor nodes use the Bluetooth module to connect to laptops or smart phones.

MAQS is a mobile sensing system for indoor air quality which collects data from sensors inside a room and sends them to a mobile phone via Bluetooth [Jiang2011]. Mobile phones use Wi-Fi network to send data on server, from where data is analysed and visualized on web server.

7 CONCLUSIONS

Deliverable D3.4.1 corresponds to the first official release of the open source implementation of the OpenIoT publish/subscribe middleware for mobile internet-connected objects which enables energy-efficient and mobility-aware data collection and dissemination in environments where mobile devices are used as gateways between mobile internet-connected objects and the cloud. This middleware is named Mobile Publish/Subscribe (MoPS). The release comprises a prototype implementation of the mobile broker component, 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 will be available within the Github infrastructure of the OpenIoT project (https://github.com/OpenIotOrg/openiot).

This deliverable presents the architecture and communication model used by the MoPS middleware. Moreover, it includes a thorough description of the technical implementation of the Android application which serves as a gateway for locally connected MIOs to the cloud-based publish/subscribe processing engine, including the mobile broker component, sensor and mobile broker services and GUI for controlling the services and presenting the live data. The description includes the application architecture, interfaces and the interaction between application components, as well between the application and the rest of the system.

This deliverable also presents preliminary results of energy and bandwidth consumption measurements for three supported communication solutions between the mobile broker and cloud. Finally, it also compares MoPS middleware with the related crowdsourcing platforms.

8 REFERENCES

[AirProbe] Air Probe, http://cs.everyaware.eu/event/airprobe/about

[CamMobSens] Cambridge Mobile Urban Sensing, http://www.escience.cam.ac.uk/mobiledata/

[CAROMM] Arkady Zaslavsky, Prem Jayaraman, Shonali Krishnaswamy (2013) "ShareLikesCrowd: Mobile Analytics for Participatory Sensing and Crowd-sourcing Applications: MoDA'2013 w/s @ ICDE'2013, Brisbane, April, 2013

[Dutta2009] Dutta, P., Aoki, P.M., Kumar, N., Mainwaring, A., Myers, C., Willett, W., Woodruff, A. Common Sense: Participatory Urban Sensing Using a Network of Handheld Air Quality Monitors. In *SenSys'09*, ACM (New York, NY,USA, 2009), 349-350

[Hasenfratz2012] Hasenfratz, D., Saukh, O., Sturzenegger, S., Thiele, L. Participatory Air Pollution Monitoring Using Smartphones. *2nd International Workshop on Mobile Sensing*, ACM (Beijing, China, 2012)

[HazeWatch] Haze Watch, http://www.pollution.ee.unsw.edu.au/

[Jiang2011] Jiang, Y., Li, K., Tian, L., Piedrahita, R., Yun, X., Mansata, O., Lv, Q., Dick, R. P., Hannigan, M., Shang, L. MAQS: A Personalized Mobile Sensing System for Indoor Air Quality Monitoring. In *UbiComp'11*, ACM (New York, NY, USA, 2011), 271-280

[MAQUMON] Mobile Air Quality Monitoring Network, http://www.isis.vanderbilt.edu/projects/maqumon

[MQTT2010] "MQTT V3.1 Protocol Specification" Internet Business Machines Corporation (IBM) and Eurotech, 2010

[Mun2009] Mun, M., Reddy, S., Shilton, K., Yau, N., Burke, J., Estrin, D., Hansen, M., Howard, E., West, R., Boda, P. PEIR, the Personal Environmental Impact Report, as a Platform for Participatory Sensing Systems Research. *MobiSys'09*, ACM/USENIX, (Krakow, Poland, 2009)

[Shelby2012] Shelby, Z. "Constrained RESTful Environments (CoRE) Link Format" RFC 6690, Internet Engineering Task Force, http://www.ietf.org/rfc/rfc6690.txt, August 2012.

[Shelby2013] Shelby, Z., Hartke, K. and Bormann, C. "Constrained Application Protocol (CoAP)" draft-ietg-core-coap-18, Internet Engineering Task Force, http://www.ietf.org/id/draft-ietf-core-coap-18.txt, 2013.

[Tarkoma2006] Tarkoma, S. and Kangasharju, J. Optimizing content-based routers: posets and forests. Distributed Computing, vol. 19, no. 1, 2006, 62-77

[Yan2012] Yan, Z., Eberle, J., and Aberer, K. Optimos: Optimal sensing for mobile sensors. In MDM '12, IEEE Computer Society (Washington, DC, USA, 2012), 105–114.

[Ziftci2012] Ziftci, C., Nikzad, N., Verma, N., Zappi, P., Bales, E., Krueger, I., Griswold, W. G. Citisense: mobile air quality sensing for individuals and communities. In *SPLASH'12*, ACM (Tucson, Arizona, USA, 2012), 23-24

