Future Generation Computer Systems 56 (2016) 607–622

Contents lists available at ScienceDirect

Future Generation Computer Systems

journal homepage: www.elsevier.com/locate/fgcs

A mobile crowd sensing ecosystem enabled by CUPUS: Cloud-based publish/subscribe middleware for the Internet of Things

Aleksandar Antonić, Martina Marjanović, Krešimir Pripužić, Ivana Podnar Žarko *

University of Zagreb, Faculty of Electrical Engineering and Computing, Unska 3, HR-10000 Zagreb, Croatia

HIGHLIGHTS

- A generic ecosystem for MCS services based on a publish/subscribe middleware.
- High-performance publish/subscribe processing middleware.
- Selective and data-driven acquisition of sensor data on mobile devices.
- Evaluation based on an MCS air quality monitoring campaign.

ARTICLE INFO

Article history:
Received 15 November 2014
Received in revised form
28 June 2015
Accepted 6 August 2015
Available online 28 August 2015

Keywords:
Mobile crowd sensing
Publish/subscribe middleware
Internet of Things
Cloud service

ABSTRACT

Mobile crowd sensing (MCS) is a novel class of mobile Internet of Things (IoT) applications for community sensing where sensors and mobile devices jointly collect and share data of interest to observe phenomena over a large geographic area. The inherent device mobility and high sensing frequency has the capacity to produce dense and rich spatiotemporal information about our environment, but also creates new challenges due to device dynamicity and energy constraints, as well as large volumes of generated raw sensor data which need to be processed and analyzed to extract useful information for end users. The paper presents an ecosystem for mobile crowd sensing which relies on the CloUd-based PUblish/Subscribe middleware (CUPUS) to acquire sensor data from mobile devices in a flexible and energy-efficient manner and to perform near real-time processing of Big Data streams. CUPUS has unique features compared to other MCS platforms: It enables management of mobile sensor resources within the cloud, supports filtering and aggregation of sensor data on mobile devices prior to its transmission into the cloud based on global data requirements, and can push information of interest from the cloud to user devices in near real-time. We present our experience with implementation and deployment of an MCS application for air quality monitoring built on top of the CUPUS middleware. Our experimental evaluation shows that CUPUS offers scalable processing performance, both on mobile devices and within the cloud, while its data propagation delay is mainly affected by transmission delay on wireless links.

© 2015 Elsevier B.V. All rights reserved.

1. Introduction

We are witnessing an increasing interest in *Mobile Crowd Sensing* (MCS) applications [1] which leverage "the power of the crowd" to observe and measure phenomena over a large geographic area by means of wearable sensors and mobile devices. Users carrying such mobile sensing devices are becoming a rich source of data about their environment and enablers of community sensing applications. MCS applications have the power to create awareness about a specific large-scale phenomena and to ignite "crowd intelligence" [2]. Moreover, the inherent user/device movement and ubiquitous connectivity create opportunities for dense spatial and temporal sensing that would otherwise be impossible to achieve.

MCS applications are characterized by a vast amount of moving data sources generating sensor streams falling in the Big Data domain. They have been categorized in the literature as either participatory [3] (assuming the active involvement of participants in choosing to contribute data), or opportunistic [1] (referring to autonomous data collection, not requiring explicit user interaction). Sensor data streams are typically preprocessed on mobile devices and then transmitted to a remote cloud for further processing [4].

* Corresponding author.
E-mail addresses: aleksandar.antonic@fer.hr (A. Antonić), martina.marjanovic@fer.hr (M. Marjanović), kresimir.pripuzic@fer.hr (K. Pripužić), ivana.podnar@fer.hr (I. Podnar Žarko).

http://dx.doi.org/10.1016/j.future.2015.08.005
0167-739X/© 2015 Elsevier B.V. All rights reserved.

This is in line with cloud-centric IoT solutions where data analytics, storage and real-time processing are performed in the cloud [5]. A crucial feature of MCS applications is their ability to inform users in near real-time about their surroundings [6]: users can receive context-related notifications and alerts affecting their decisions and behavior while being mobile.

For example, an MCS traffic monitoring application involves numerous individuals who continuously contribute location data to cloud servers which estimate traffic conditions by processing a large number of location readings in real-time. The users can opt to continuously provide their location data in exchange of the ability to track traffic conditions on their mobile devices. Traffic statistics can be used by city officials to detect highly-congested roads and traffic hotspots, or to correlate this data, e.g., with air quality. The sensed location data is typically provided to cloud servers autonomously and periodically, and can be classified as *opportunistic sensing*: It can potentially quickly drain out the battery on mobile devices and lead to certain geographical areas being densely covered by sensor readings, while other areas may suffer from lack of available data readings [7]. Moreover, the aggregated traffic data provided to users from the cloud needs to be up-to-date and may frequently change.

In view of the previously depicted usage example, we can identify the following *challenges* as vital for the evolution of MCS applications: Since MCS applications run in dynamic environments comprising sensors, various mobile devices and the cloud, it is vital to achieve *energy-efficient and context-aware* orchestration of the sensing process with data transmission from sensors through mobile devices into the cloud, as well as from the cloud to mobile devices. In other words, both the sensing process and data transmission from mobile devices to the cloud need to be controlled so that valuable data is collected when it is indeed required by an MCS application. In addition, *end-to-end data propagation delay* represents a key quality measure from the user's perspective since frequent changes of user context can quickly make information obsolete or irrelevant. Other open MCS issues are identified in [8] as follows: the need for standardization, enabling complex data mining, integrating cloud capabilities and moving towards fog computing.

Publish/subscribe middleware offers the mechanisms to deal with the previously identified challenges: It enables filtering of sensor data on mobile devices, forwarding of this data into the cloud, 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 infrastructure for continuous data processing and push-based messaging in distributed environments with the ability to efficiently process large amounts of data due to simple subscription languages and efficient matching algorithms [9]. We have already identified publish/subscribe middleware as a key enabler for MCS ecosystems [10,11] and in this paper we provide a comprehensive evaluation of our publish/subscribe solution named CloUd-based PUblish/Subscribe (CUPUS) in terms of scalability and data propagation delay. CUPUS is an open-source publish/subscribe middleware designed for mobile IoT environments developed within the framework of the FP7 project OpenIoT, *Open Source blueprint for large scale self-organizing cloud environments*, which builds an open-source cloud platform for the IoT.[1] Its modular design successfully tackles MCS challenges regarding elastic utilization of a cloud environment while complex data mining tasks can be introduced by specialized components.

[1] CUPUS is one of the core OpenIoT platform components published under the LGPL licence and is available at the Github infrastructure of the OpenIoT project (https://github.com/OpenIotOrg/openiot).

CUPUS comprises two main software components: a *mobile broker* responsible for data filtering/aggregation on mobile devices, and *cloud broker* for continuous big stream processing within an IoT cloud. CUPUS has unique features compared to other MCS platforms: It supports selective and flexible acquisition and filtering of sensor data on mobile devices, efficient continuous processing of sensor data streams within the cloud, and near real-time delivery of notifications from the cloud to mobile devices. CUPUS is tailored to the requirements of mobile and resource-constrained environments with a goal to reduce the overall energy consumption, from data acquisition to data transmission and delivery, while controlling the MCS data density, both in space and time. This is achieved by filtering or aggregating data close to its production place and by suppressing the data transmission from mobile devices into the cloud when there is no global interest in the produced sensor data, or when there are redundant data sources located, e.g., within the same geographical area.

CUPUS supports content-based publish/subscribe, both within the cloud and on mobile devices. It is not reinventing the publish/subscribe principles, but is rather adjusting the well-known content-based advertise–subscribe–publish interaction pattern, originally proposed in the SIENA event-based system [12], to an MCS ecosystem. Our solution leverages the power of the crowd (1) to enable selective and data-driven acquisition of sensor data on mobile devices in accordance with the global and real-time requirements for such data and (2) to enable individual near real-time environment awareness with a personalized user notification mechanism. The major paper contribution is the design of a generic ecosystem for MCS services based on a publish/subscribe middleware and the implementation of high-performance publish/subscribe processing components, both for mobile devices and the cloud.

To demonstrate the usability of the proposed MCS ecosystem and to evaluate CUPUS performance in real-world deployments, we have built an application for crowd-sensed air quality monitoring [13]. We present the results of an experimental evaluation of CUPUS performance by using a real sensor dataset collected during an air quality measurement campaign organized in July 2014 in Zagreb, Croatia. The experiment evaluates the scalability of CUPUS components, both on mobile devices and the cloud, as well as the elasticity of the cloud broker. We also measure the average sensor data propagation delay, from the sensing process until notification delivery to mobile devices, to assess the usability of CUPUS in real-world deployments.

The paper is structured in the following way: Section 2 introduces the MCS ecosystem enabled by the CUPUS middleware. Section 3 presents implementation details of the CUPUS middleware, both regarding the mobile broker and cloud broker components. We introduce the use case for air quality monitoring in Section 4, which is used to compare CUPUS to the standardized messaging protocols for IoT in Section 5. The results of an extensive experimental evaluation of the CUPUS middleware are given in Section 6. Section 7 provides an overview of related work by comparing prominent MSC platforms, while Section 8 concludes the paper and gives pointers to future work.

2. Mobile crowd sensing ecosystem

An MCS ecosystem needs to provide the means for collecting sensor data from mobile devices over large geographical areas where devices may generate the data either with or without explicit user actions. Due to huge volumes of sensor data, a cloud infrastructure is suitable for the implementation of an elastic, scalable and modular MCS ecosystem. All components responsible for processing of integrated data streams steaming from various data sources are located in the cloud, while data sources and

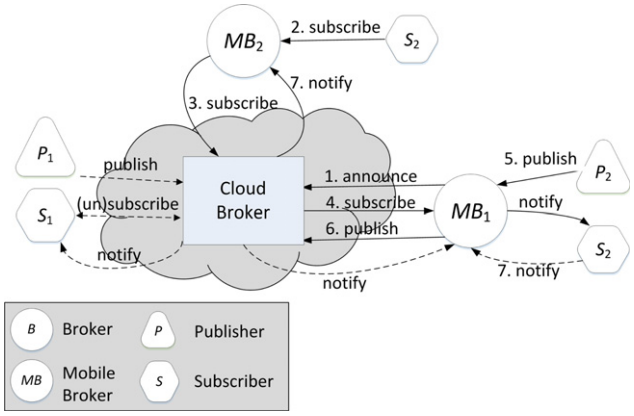


Fig. 1. Mobile publish/subscribe model.

data consumers are deployed on mobile devices with connections to the Internet. The ecosystem is enabled by a publish/subscribe middleware for flexible data acquisition, near real-time processing of sensor data, and data sharing across mobile devices and through the cloud. In this section we first briefly present the publish/subscribe communication model integrating sensors and mobile devices with the cloud, and second, we introduce the MCS ecosystem services and main components.

2.1. Publish/subscribe communication model

Publish/subscribe middleware is suitable for flexible data collection and dissemination in MCS environments where mobile devices are used as gateways for collecting and transmitting sensor data into the cloud, while at the same time mobile devices receive the data of interest from the cloud. In contrast to existing centralized database solutions which typically send all sensed data into the cloud, a publish/subscribe-based solution can control the data acquisition process on mobile devices, and transmit preprocessed and possibly aggregated data from mobile devices into the cloud only in situations when the data is indeed required by an MCS application. Additionally, the data is pushed back from the cloud to mobile devices only if it is relevant and of interest to end-users (depends on their context and information needs).

We denote data sources in MCS applications as Mobile Internet-connected Objects (MIOs). An MIO is IP-addressable and its geographical location changes over time. Examples are smartphones with built-in accelerometer and temperature sensors or wearable air quality sensors which communicate over Bluetooth with a mobile device. The data produced by MIOs needs to be geotagged, either by an exact location measured by GPS or cell identifier (e.g., a mobile network cell or MGRS area identifier[2]) since location represents the major contextual information for an MIO reading. In an MCS scenario, large areas of interest need to be covered by a predefined amount of sensor readings determining the required MCS data density for an area over time. However, typical mobility patterns of users and MIOs lead to a dynamic sensing coverage with certain areas being densely covered by sensor readings, while other areas may suffer from lack of available data readings, as we have shown when analyzing true data traces in [7]. Moreover, the interest in MCS data changes over time so that certain areas might not require sensor coverage at all. The following publish/subscribe model empowers the MCS ecosystem to manage data acquisition from MIOs, as well as to balance data demands and match them to available data sources.

The publish/subscribe model comprises a set of publishers P_i and a set of subscribers S_j that interact over a hierarchical two-tier publish/subscribe network composed of *mobile brokers*, MB_k, and a *cloud broker*, *CB*. An example model is shown in Fig. 1. Publishers, e.g., wearable or built-in sensors, *publish* data items and send them either to mobile brokers or directly to the cloud broker. Subscribers, e.g., processes on mobile devices, can activate and dismiss subscriptions by sending messages *subscribe* and *unsubscribe* to mobile or cloud brokers, which in turn use the message *notify* for push-style delivery of matching data items, i.e., items that satisfy subscription constraints, to subscriber processes. The cloud broker is responsible for efficient matching of data items to active subscriptions, as well as their subsequent delivery to either subscribers, mobile brokers, or other remote services, i.e., components that have defined matching subscriptions.

Hereafter we describe the interactions between model components by an example. After their initial registration with a cloud broker, mobile brokers can *announce* the type of data that can be produced by their publishers. For example, P_2 in Fig. 1 is a wearable gas sensor detecting levels of nitrogen dioxide (NO_2) and ozone (O_3). After MB_1 detects P_2 because they exchange signaling information over a Bluetooth connection, MB_1 can, in addition to its location, provide the type of data items it is able to transmit to *CB* in the future. MB_1 sends a message *announce*(NO_2, O_3, x, y), where $x = 45.81302$ and $y = 15.97781$ represent MB_1's current geographical latitude and longitude. The reason for creating the announce message is the following: We need to activate subscriptions from the cloud broker on MB_1, but only those that can potentially match future publications created by P_2. Obviously, these subscriptions are location-dependent, and thus P_2 announces its new location each time it significantly changes, e.g., when P_2 enters a new cell. If there are no active subscriptions on MB_1, MB_1 does not forward any data to *CB*.

Location-based subscriptions and announcements reduce the number of subscriptions to be processed by MB_1 as it is not desirable to activate all subscriptions from the cloud on a single mobile device. This is achieved by matching the announce message to existing subscriptions on *CB*. For example, when S_2 in Fig. 1 defines a new subscription s_i=[$NO_2 > 40\ \mu gm^{-3}$ AND $45.81 < lat < 45.82$ AND $15.96 < long < 15.98$], s_i is forwarded to *CB* by MB_2. Based on the previously received announce message, *CB* identifies s_i as the subscription potentially matching future publications of P_2. Thus, *CB* sends a message *subscribe* to activate s_i on MB_1. Further on, when MB_1 receives sensor readings from P_2 which match s_i, MB_1 transmits P_2's geotagged data items into the cloud. Note that the announce mechanism enables *location management* of MIOs over time.

By reusing the inherent features of the two-tier publish/subscribe model, it is possible *to control the MCS data density* over a predefined area since adequate subscriptions are activated on mobile brokers, as instructed by the back-end cloud system based on current requirements of an MCS application. If we assume that the density demand is predefined for an area as required by MCS application requirements, MIOs residing in the 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, or (2) to restrain from such transmissions since the cloud has already acquired sufficient data samples for the area.

This is the main mechanism for flexible data acquisition which enables reduced energy consumption on mobile devices due to reduced frequency of data transmissions from MIOs into the cloud, as shown experimentally in [14].

[2] The Military Grid Reference System (MGRS) is the geocoordinate standard used by NATO for locating points on the earth.

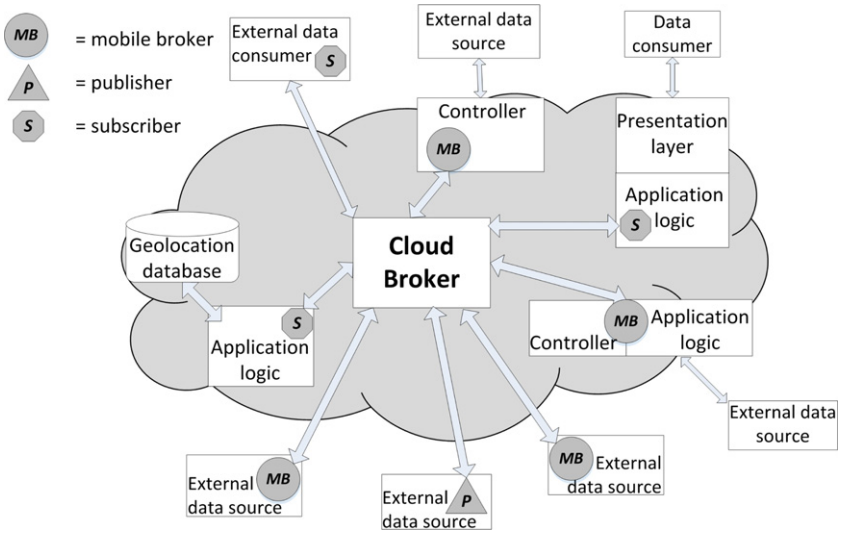


Fig. 2. Crowd sensing platform enabled by CUPUS middleware.

2.2. Ecosystem design

The ecosystem, as shown in Fig. 2, integrates three classes of components in the cloud: the main processing engine (Cloud Broker), application logic and controller component. MCS data sources and data consumers are running on devices connected to the ecosystem over an IP connection. The ecosystem offers the following services:

- Location management of MIOs: The ecosystem keeps track of all MIOs, their locations and sensing capabilities by use of the publish/subscribe announce mechanism;
- Sensor discovery: Based on MIO's locations and their sensing capabilities, an MCS application can search for available sensors within the ecosystem;
- Data acquisition and management: A control of the sensing process is realized within the cloud by combining the information acquired through sensor location management and sensing requirements for certain geolocations;
- Data density management: Controlled data acquisition takes into account sensing coverage, as well as specific MCS application and end user requirements.

The central component of the MCS ecosystem is the Cloud Broker as the basic data processing element. The Cloud Broker is responsible for data acquisition from MIOs, data processing and dissemination to external data consumers, e.g., actuators or mobile devices. The Cloud Broker is optimized for efficient data processing, i.e., matching of incoming sensor readings (*publications*) with continuous data queries (*subscriptions*). A publication can be either a raw data item collected from sensor nodes or preprocessed and aggregated data object. The Cloud Broker's ability to elastically adapt to an incoming publication rate and to scale with an increasing number of data sources and consumers ensures that the performance of the ecosystem and service responsiveness remains at a satisfactory level. The Cloud Broker collects MIO locations and enables management of MIO resources using the announce mechanism without additional user intervention.

An application logic component implements the core functionality of an MCS application and orchestrates the data acquisition process in cooperation with other components. It integrates a subscriber which receives and monitors selected data streams being processed by the Cloud Broker. It can be linked to a spatial database to store either raw, aggregated or preprocessed sensor data for future data mining tasks.

A controller component is responsible for location management of MIOs and data density management. It may implement complex management methods since the data density and quality depends on a number of parameters, e.g., sensor mobility, available MIO battery level and quality of sensor readings.

Data sources and data consumers are the key ingredients of an MCS ecosystem which are located outside the cloud infrastructure. Data sources are MIOs which can run either plain publishers contributing all sensor readings to the cloud broker, or mobile brokers with data preprocessing capabilities. In addition to global requirements guiding the data acquisition on MIOs, a mobile broker enables an MIO to define its local rules for deciding whether the data may be contributed to the cloud or not. This is an important feature since MCS applications can invade user privacy.

3. Cloud-based publish/subscribe middleware (CUPUS)

CUPUS implements the publish/subscribe model presented in Section 2.1 which results in a *flexible and energy-efficient sensor data acquisition process*. It provides the mechanisms for both *content-based and topic based publish/subscribe processing* based on the Boolean subscription model. Similar to [15], 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. 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. The following relational operators are supported for numerical values: $<, \leq, >, \geq, =$ and BETWEEN. Additionally, the middleware supports String operators such as CONTAINS, BEGINS_WITH and ENDS_WITH.

Further requirements imposed on publish/subscribe middleware for MCS applications are related to data delivery: CUPUS supports *at-least-once delivery semantics* since typical MCS applications are not safety-critical. In terms of system performance, the *end-to-end data propagation delay* needs to be low to ensure delay with near real-time properties for applications that typically involve rapid context change. System performance is affected, on the one hand, by the Cloud Broker implementation which needs to be elastic to efficiently use cloud resources in accordance with the processing load. On the other hand, the preprocessing tasks

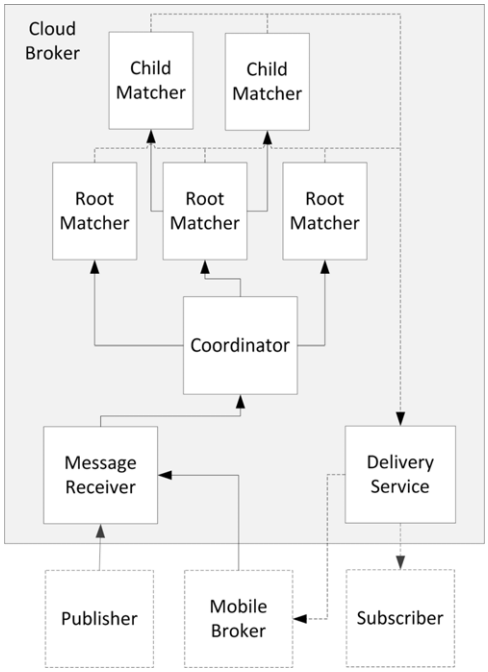


Fig. 3. Cloud Broker architecture.

performed by a Mobile Broker also influence the data propagation delay. Hereafter we review the design of CUPUS Cloud Broker and Mobile Broker components.

3.1. Cloud broker

The Cloud Broker architecture is based on the generic data stream processing architecture defined in [16] where continuous queries, i.e., subscriptions, are maintained in processor memory for efficient processing of incoming data objects, i.e., publications. As depicted in Fig. 3, it uses a Message Receiver component which accepts and validates subscriptions and publications from distributed sources, while the Delivery Service component outputs multiple data streams, where each data stream is pushed to a distributed subscriber or mobile broker. Publications from a single data source can be regarded as an incoming data stream. Each incoming publication is seen only when entering the Cloud Broker unless explicitly stored in memory.

The Cloud Broker is used as a central processing unit within the cloud to perform the following tasks:

1. to receive and manage publications and subscriptions received from data sources and destinations,
2. to perform matching of active subscriptions to publications,
3. to deliver matching publications to subscribers, and
4. to deliver subscriptions to mobile brokers.

The matching of publications to subscriptions is the most demanding task of the Cloud Broker, and thus we need to replicate the matchers and dynamically allocate cloud resources to them in accordance with the current processing load. For the implementation of the matching algorithm we use an approach similar to [17] which stores active subscriptions into a forest based on the covering relationship between subscriptions. A *subscription forest* is a data structure which stores all active subscriptions from end-users and enables partial subscription ordering. It can reduce the matching time when comparing an incoming publication with a large number of subscriptions to detect a list of subscribers requiring a notification about the publication. Such data structure is a key prerequisite for the implementation of efficient matching. Further optimizations can be achieved by dividing the load among the matcher processes which can be organized in either flat or hierarchical architectural structures. These structures need to dynamically adapt to the processing load to ensure elastic Cloud Broker processing.

The flat cloud broker architecture uses independent Root Matcher components in parallel. Those matchers are created and initialized on demand by the Coordinator component. The coordinator is a central component of the Cloud Broker which starts the matcher components at startup, distributes the processing load among them, and has a direct connection to all of them. The coordinator can allocate and start new matchers, or stop existing ones in accordance with the processing load.

The flat architecture has a simple way of processing subscriptions and publications. The coordinator component distributes received subscriptions in a round-robin fashion to root matcher components which results in N root matchers having a subscription data structure containing one N-th of all subscriptions being processed by the Cloud Broker. Incoming publications are replicated by the coordinator and sent to all matchers. This can potentially slow down the coordinator since it has to make N sequential communication operations, which makes the coordinator a potential system bottleneck. We use the flat cloud broker architecture in our current CUPUS implementation.

The hierarchical cloud broker architecture uses two kinds of matcher components: root and child matchers. Each matcher component is responsible for a part of the subscription forest that spans over all active subscriptions received by the Cloud Broker. A root matcher is the top matcher which holds the root node of one of the subscription forest trees. The number of root matchers equals the number of forest trees. A child matcher holds the root of a subtree and is subordinate to one of its root matchers. In this architecture, the coordinator maintains the list of root node subscriptions and forwards each publication to a single root matcher. When a root matcher needs to forward an incoming publication further down the matcher tree, it is forwarded to a single matcher process. Thus, a forest based matcher implementation has the potential for an improved performance compared to the flat architecture. However, its implementation is much more complex.

3.2. Mobile broker

The CUPUS Mobile Broker runs on mobile devices and serves as a gateway for locally connected sensors. Its main tasks are the following:

1. to control the locally connected sensors and corresponding data acquisition process, to preprocess acquired data and transmit it to the cloud, and
2. to receive publications from the cloud and display those which are of user interest.

The first task is rather complex because whenever a user changes his/her context or location, the application needs to inform the Cloud Broker about the change so that the Cloud Broker has an up-to-date information about connected data sources and subscriptions that are related to the new context. All contextual changes are reported via announce messages. The Cloud Broker's reply to the announce message are subscriptions representing requests from ecosystem controllers or other users of the ecosystem which match the announcement. In other words, only a subset of cloud subscriptions, which has the potential to match its future publications, is forwarded to the Mobile Broker. This is the mechanism which ensures that the number of active subscriptions per Mobile Broker is low so that it can be processed in parallel on mobile devices: The Cloud Broker can perform further actions, e.g. subscription merging, to further reduce their number. Note

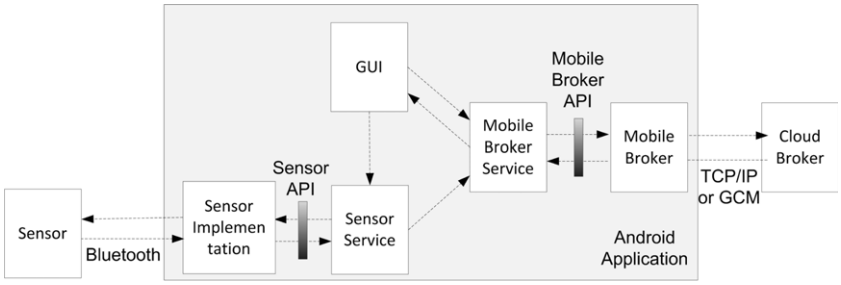


Fig. 4. Mobile broker architecture for Android device.

also that the set of subscriptions on a Mobile Broker is dynamic which further affects its processing performance. The received set of subscriptions provides vital information for the management of attached sensors since it instructs the Mobile Broker to interact with an adequate data source to acquire data which is potentially needed for an MCS task. After the data is acquired, additional preprocessing is performed to assess whether the obtained data should be sent to the cloud or not. The two step management process is necessary because the Mobile Broker first needs to determine which sensors should be turned on, and in the second step we are preprocessing the obtained data to check whether it triggers additional processing and a publication event to deliver the data to the Cloud Broker.

The second task is concerned with push-based delivery of publications from the cloud to mobile devices [18] which creates most problems to programmers [19]. In addition, it can incur large energy costs since a recent study shows that periodic transfers in mobile applications which account for only 1.7% of the overall traffic volume contribute to 30% of the total handset radio energy consumption [20]. Thus hereafter we briefly report two solutions that we have tested to enable delivery of notify messages in the CUPUS system: (1) persistent TCP connection, and (2) 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. Furthermore, mobile operators typically block client requests to mobile phones.

For a fully 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 Brokers in case of CUPUS) and mobile devices running the Android OS. GCM uses a simple message format limited to 4 kB. A mobile service does not need to be in an active state to receive such notifications since the Android OS starts or wakes up the service upon a received message. The mechanism does not create, handle or destroy any additional connections, which makes it a mobile push mechanism without additional overhead. Since the support for the GCM service is an integral part of the Android OS, GCM only requires that a radio interface is online, and allows the processing unit to go to the 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 the minimum. The main drawback is limited availability (only for Android OS) and dependency on a third party solution.

Fig. 4 depicts the architecture of the Mobile Broker Android application. The application consists of two background services (a sensor and mobile broker service), Graphical User Interface (GUI) for controlling the services and presenting live data, and mobile broker entity. The main task of the sensor service is to acquire sensor readings from connected sensors. The communication between a physical or virtual sensor and the sensor service depends on the available communication means (typical communication protocol is the Bluetooth protocol). The communication between the sensor service and the mobile broker service is implemented through the Android internal intent broadcasting and filtering mechanism. We have reused the standard Android mechanism for intra-application communication between different application components and inter-application communication between different applications (e.g., if some other application on the device is using the data acquired from the ecosystem or from the physical sensor). The mobile broker service communicates with the mobile broker entity using the mobile broker API. It runs in the background and updates GUI elements when necessary (e.g., when a new notification is received and should be displayed to the user) and also sends publications if the data acquired from local sensors is needed by the ecosystem.

The Mobile Broker data processing engine performs the matching of received sensor reading with a set of local subscriptions to determine whether the obtained data should be sent to the cloud broker or not. We have developed the matching algorithm using two different subscription data structures because it is not clear which approach offers better processing performance on Android devices: Subscriptions received from the Cloud Broker are stored either in a linked list or a subscription forest similar to the one used by the Cloud Broker. Pre-processing of a newly created publication ends when a first matching subscription is found in the structure because the publication should be forwarded to the Cloud Broker.

The *linked list* has obvious shortcoming because the complexity of the matching process is $O(n)$ in the worst case. Since we expect to receive similar subscriptions relevant to MIO-generated data as enabled by the announce mechanism, it is possible that in practice the matching process will not be performed on all elements of the list. An additional benefit from using the linked list is reduced complexity regarding its management (i.e. adding or deleting a subscription to the structure).

The *forest structure* is similar to the subscription structure used by the Cloud Broker. The difference is in the possible depth of the structure, because we have observed that an Android application can become unstable with forests of high depth. During matching only root subscriptions are matched to a publication, because if a publication does not match a root subscription, it cannot match any of its leaf subscriptions. The matching complexity is $O(\log n)$, but in real world scenarios it tends to $O(1)$. The shortcoming of the forest structure is additional overhead caused by increased structure management time.

In Section 6 we compare the propagation delay of notification messages with either TCP/IP or GCM over WiFi and 3G links, as well as performance of the matching processes within the mobile application with two different subscription structures.

4. Use case application: urban crowd sensing for air quality monitoring

Air pollution is a serious threat in urban environments with significant negative impact on human health. Therefore, air quality

Fig. 5. Wearable sensors for air quality monitoring and smartphones running the monitoring application.

Table 1
Summary of the SenseZGAir dataset.

Time range	2014-07-04 08:38:47 to 2014-07-10 12:49:30
Number of datapoints	151,436
Unique coordinates (lat, long)	13,905
$T/P/H$ Readings	16,835/16,835/16,811
$CO/NO_2/SO_2$ Readings	16,815/7505/9295
Sensor and device battery readings	16,835/16,835

monitoring is very important in urban environments, especially in big cities. Since air pollution is location-dependent and varies over time, e.g., highly-congested roads and industrial areas increase the concentration of toxic gases, air quality should be monitored in city areas densely, both in time and space, which is not feasible by static meteorological stations. This can be achieved by involving citizens in the air quality monitoring process so that they carry wearable sensors for air quality monitoring while moving through the city. Mobile measurements can help ecologists, public health officers and city officials to understand urban dynamics and put new perspectives in population-wide empirical public health research [21]. For example, it has been recognized that car drivers are highly exposed to airborne pollution, and are at risk of developing several types of cancer.

The Urban Air Quality Crowd Sensing use case is realized by an opportunistic MCS application built on top of CUPUS which requires active citizen involvement. The sensing and data acquisition process requires volunteers carrying air quality sensors and smartphones to contribute the sensed data to application servers. Application users are potentially all citizens that can consume the information of interest produced by volunteers to observe a phenomenon typically in their close vicinity. Citizens can define special areas of interest for which they would like to receive air quality alerts in near real-time, and these alerts can impact their movement through the city. For example, cyclists can choose a route with favorable air quality in real-time while cycling.

The air quality monitoring application integrates the data produced by low-cost wearable mobile sensors measuring temperature, humidity, atmospheric pressure, and levels of air pollutants. Wearable sensors shown in Fig. 5 are custom designed and built from off-the shelf components and communicate over Bluetooth with a smartphone. They integrate electrochemical gas sensors for measuring atmospheric sub-ppm level concentrations of carbon monoxide (CO), and either nitrogen dioxide (NO_2) or sulfur dioxide (SO_2). Two gas sensors are put onto a single sensing node to reduce its size, so that it can be easily mounted on a bicycle or warn on clothes or backpacks. Wearable sensor is powered by a rechargeable Li-ion battery with estimated autonomy of 3 days. The communication protocol between a sensor and smartphone enables an application user to start or stop the sensing process on sensor nodes, as shown on the right-hand side smartphone in Fig. 5. After the sensing process is started, measurements are produced periodically on the sensing node and are delivered to the mobile application, i.e., the mobile broker, which decides whether to transmit the data into the cloud or not. The smartphone on the left-hand side in Fig. 5 shows the perspective of an application user who wants to receive air quality alerts in the geographical area where he/she is currently residing. The application depicts the received sensor readings on a map.

Table 1 summarizes the characteristics of the SenseZgAir dataset acquired during our air quality measurement campaign in July 2014 within the area of Zagreb [13]. A subset of sensor data is visualized by a web application.[3] The web interface depicting individual sensor readings is shown in Fig. 6(a), while the corresponding heatmap displaying the data density is in Fig. 6(b). The main task of the web application is to depict changes of the dataset over time, while it also includes measurements from fixed environmental monitoring stations which are publicly available for Zagreb area.

5. Comparing CUPUS to the standardized IoT messaging protocols

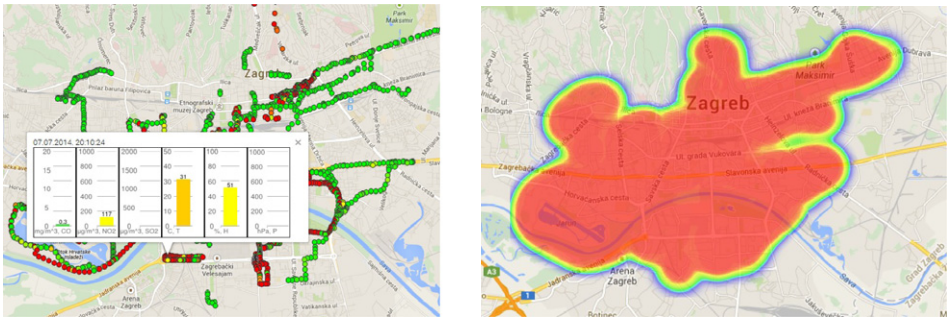
In this section we compare CUPUS with the two standardized messaging protocols for the IoT which are potential candidates for implementing MCS applications. We provide an overview of the Message Queue Telemetry Transport (MQTT) protocol and Constrained Application Protocol (CoAP), followed by their comparison based on the requirements for MCS applications.

5.1. Message Queue Telemetry Transport (MQTT) protocol

MQTT is a lightweight publish/subscribe messaging protocol suitable for the development of IoT/Machine-to-Machine (M2M) applications. It is optimized to connect physical devices generating events with enterprise servers and other consumers. MQTT was originally created as a proprietary protocol in 1999, but is since 2014 adopted as a standard by OASIS (Advancing open standards for the information society).

MQTT implements the publish/subscribe messaging pattern between two entities: a client which serves as a subscriber to a specific topic and/or as a publisher of certain content, and a broker which is responsible for processing publications and subscriptions organized in a hierarchical topic-based subscription model. MQTT

[3] http://openiot.tel.fer.hr/osjetizgzrak/.



(a) Sensor measurements. (b) Heatmap showing data density.

Fig. 6. Web application depicting a subset of data collected on July 7th, 2014.

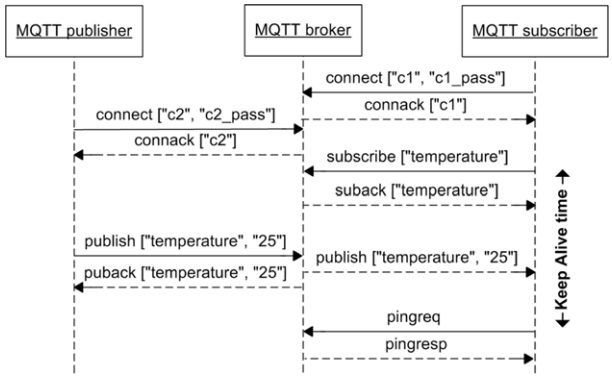


Fig. 7. Communication between MQTT entities.

implements all of the system complexities on the broker's side, while it uses a simple implementation on the client's side [22]. It is situated on the application layer and uses TCP/IP protocol for the message exchange.

Fig. 7 shows an example interaction between a publisher and subscriber via an MQTT broker. Since MQTT is a connection-based protocol, a client first sends a *connect* message to the broker with parameters required for client identification. Afterwards, a client (subscriber) registers its interest on a specific topic by sending a *subscribe* message. A client (publisher) publishes some content by sending a *publish* message to the broker which contains data published on a certain topic. The broker forwards received *publish* message to all subscribers interested in the topic. Each message is followed by an acknowledgment sent by a responder (i.e., *suback* and *puback* messages). Additionally, if a client wants to unsubscribe from a topic it can send an *unsubscribe* message and will receive an *unsuback* message as a reply. To enable persistent communication, an MQTT broker needs to be aware of the connectivity state of the clients. Therefore, the MQTT protocol defines the *pingreq* and *pingresp* messages which are exchanged between a broker and client if no other messages have been sent within a certain time-frame.

5.2. Constrained Application Protocol (CoAP)

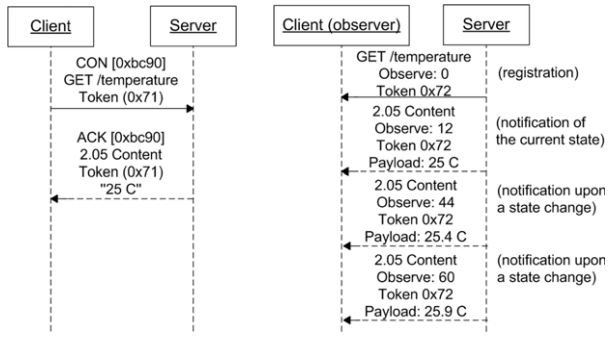
The Constrained Application Protocol (CoAP) is a lightweight messaging protocol suitable for communication of resource constrained devices defined in RFC 7252 [23]. It is designed for the development of M2M/IoT applications such as smart home, smart energy or building automation applications. CoAP was designed in 2010 as a generic web protocol based on Representational State Transfer (REST) architecture which supports the request/reply interaction model similar to HTTP. All standardization activities regarding CoAP are carried out by the Constrained RESTful environments (CoRE) Working Group established by the Internet Engineering Task Force (IETF).

CoAP is based on the client/server model, where an endpoint can act as both a client and server. Typically, a CoAP client sends a request for a resource, identified by the Uniform Resource Identifier (URI), to a CoAP server, which responds with the requested content. CoAP uses the *coap* and *coaps* URI schemes for identifying CoAP resources and to provide the means to locate resources. The *coap* and *coaps* URI schemes can thus be compared to the *http* and *https* URI schemes, respectively [23]. Unlike HTTP and MQTT, CoAP uses a datagram-oriented transport protocol such as UDP. As UDP is inherently not reliable, CoAP provides its own reliability mechanism by defining a *Confirmable* (CON) message which requires, as a response, an *Acknowledgment* (ACK) message with the same *Message ID*. Additionally, there is a *Non-confirmable* (NON) message which does not require reliable transmission (e.g., a single sensor measurement can be sent by using a NON message). If a server is not able to process a CON or NON message it can respond with a *Reset* (RST) message. CoAP offers built-in discovery of services and resources, as well as multicast support and asynchronous message exchanges. CoAP also supports a subscribe/notify interaction model, where a server sends a *notify* message to a client about a change of resource identified by a given URI. Note that an URI corresponds to a topic in the MQTT model. This extension can be provided by including the Observe Option in a GET request, as defined in the observer design pattern [24].

Fig. 8 exemplifies interactions between a client and server using CoAP. Fig. 8(a) shows a typical request/reply interaction when a client sends a basic GET request for a temperature reading carried in a *Confirmable* message and a server answers with a piggybacked response (i.e., the response is sent within the *Acknowledgement* message). Each response includes a *Token* which is used to match responses to requests. Fig. 8(b) depicts the observer design pattern: a CoAP client (i.e., an observer) registering its interest in the current state of the resource by sending an extended GET request to the server which stores the client in the list of observers of that resource. Whenever a resource changes its state, the server notifies each interested client. The client in Fig. 8(b) receives three notifications: the first upon its registration and then two upon a state change. Both the registration request and notifications are identified by the Observe Option included in the GET request. Additionally, a response (i.e., a notification) includes a *Token*, so the client can easily correlate it to the previously sent request.

5.3. Comparison between CUPUS, MQTT and CoAP

To put the CUPUS middleware in the context of existing messaging solutions for MCS applications, hereafter we compare it with the two previously described standardized solutions: MQTT and CoAP. They are compared based on the following IoT-related



(a) Request/reply model. (b) Observer design pattern.

Fig. 8. Communication between the CoAP entities.

Table 2
Comparison between the CUPUS middleware and standardized messaging protocols.

	CUPUS middleware	MQTT protocol	CoAP
Architecture	Centralized	Centralized	Decentralized
Application and platform independent	Yes, requires Java runtime environment	Yes, support for many programming languages	Yes, support for many programming languages
Standardized	No	Yes, OASIS standard	Yes, RFC 7252
Information flow	Semi-defined, preprocessed data goes through the Cloud Broker	Semi-defined, data goes through an MQTT broker	Ad hoc, supports peer-to-peer communication
Object discovery	Yes (announcement mechanism)	No	Yes (Resource Directory)
Query language	Content-based	Topic-based	URI-based
Flexible data processing support	Yes, by adding a new processing component	No	No

requirements identified in [25,26]: solution architecture, support for various platforms and applications, standardization efforts, modeling of information flow, support for discovery of objects (e.g., devices, resources, measuring capabilities, etc.), query support and data processing support. Table 2 summarizes the comparison.

As the MQTT protocol is a minimalistic publish/subscribe protocol standardized by OASIS, this results in available implementations for various platforms (Arduino, Android) and programming languages (Java, C, Python). Similarly to the CUPUS middleware, MQTT uses a central processing component that matches user subscriptions with incoming publications; however, the CUPUS middleware offers an expressive subscription language compared to the hierarchical topic-based subscriptions offered by the MQTT protocol. In contrast to the CUPUS middleware which uses the announce mechanism to discover objects that can respond to a user subscription, MQTT entities always send acquired data to a broker, regardless of its usefulness. In addition, the MQTT protocol does not consider more complex data stream processing functions (e.g., average, sum) to be offered as part of the protocol: it only forwards messages published on defined topics from the broker to interested users.

The CoAP protocol is another well-known and standardized messaging solution which implements the Observer pattern, a communication paradigm used by constrained devices in IoT applications that benefit from direct communication channels between devices. More details on how to achieve ad hoc communication using CoAP in various environments, often protected by firewalls, can be found in [27]. CoAP has a decentralized architecture and lacks a central processing component, meaning that each data source processes and delivers data to all interested clients when the observe pattern is used. A centralized component of the CoAP protocol is the Resource Directory that serves as a directory to discover available devices, i.e., CoAP servers, so that a user can make a request to a device of interest. A CoAP server supports resource discovery. Although its subscription mechanism is quite different than the one used by the MQTT protocol, expressiveness of the subscription language is similar, with a major difference that in the CoAP protocol each device handles its own subscriptions unlike the MQTT protocol where all subscriptions are handled by the MQTT broker.

Messaging footprint. To compare the messaging footprint of the three analyzed IoT messaging solutions, we count the number of messages exchanged within an MCS application assuming that only a part of the sensor data is needed by the application and its users. We have used the SenseZGAir dataset in this simulation experiment, while subscriptions are generated at random to achieve a subscription coverage between 30% and 90%.[4] After selecting subscriptions which achieve a satisfactory level of coverage, all sensor readings were published and processed by all three solutions. Since subscriptions were generated in a random fashion, we have performed 1000 iterations of each experiment to obtain the distribution of messaging footprint which includes all messages communicated by data sources and messages controlling data acquisition.

Since CoAP is not designed to natively support MCS applications, mostly because of its nature where clients communicate directly with data sources publishing data over a CoAP server, we assumed in the experiment that a single MCS application server will consolidate all data readings from mobile sensors and devices running CoAP servers to act as a single point-of-entry for end-users. Thus, an end-user subscribes at the MCS server for desired information, and the MCS server creates individual CoAP subscriptions on mobile devices to be able to forward data to

[4] Subscription coverage means that a subscription is defined on a sensor reading, subsequently causing that the reading produces at least one notification.

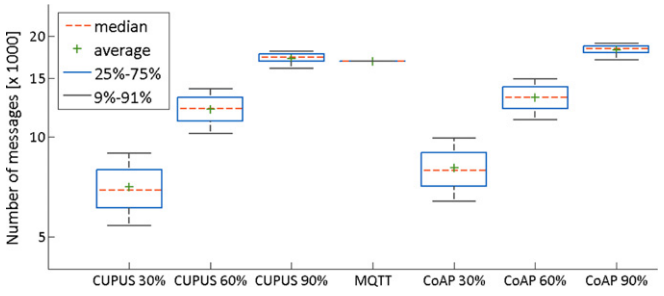


Fig. 9. Messaging footprint of the CUPUS middleware, MQTT protocol and CoAP solution while varying the subscription coverage.

end-users. If an MCS application would be implemented in a P2P fashion without a central application server then end-users would have to communicate directly with mobile data sources. In a P2P case, the data source would have to send data updates individually to all interested users, leading thus to high energy and bandwidth consumption on resource limited devices. Fig. 9 depicts the distribution of messaging footprint for the three observed solutions with various subscription coverage. The MQTT protocol has only one chart since it always publishes all available sensor readings, thus subscription coverage does not affect the messaging footprint. The CUPUS and CoAP messaging footprint obviously increases with the increase of subscription coverage as more messages are delivered from data sources to the MCS application server. Both solutions need to report a change of context: the CUPUS middleware uses an announce message and CoAP server sends a request to the Resource Directory. These messages are also counted in the messaging footprint of the two solutions in addition to data messages. We notice an offset between the CUPUS middleware and CoAP solution, where the offset is the number of context changes. Whenever a CUPUS entity changes its context, it sends an announce message and as a reply receives related subscriptions. In the CoAP solution a central MCS application server has to additionally send those subscriptions whenever it notices a change of the device context in the Resource Directory, thus communicating three messages for every context change (two with Resource Directory and one with a subscription). Both CUPUS and CoAP achieve significant savings in terms of exchanged messages compared to MQTT when the subscription coverage is below 60%, but with high percentage of subscription coverage the messaging footprint of the CUPUS middleware and CoAP is higher than the MQTT protocol because of the overhead introduced through the data acquisition control. Thus in cases when all data acquired by mobile sources is required by an MCS application, MQTT is the best option in terms of the messaging footprint at mobile sensors and devices. Otherwise in situations with redundant and irrelevant measurements which are more realistic in typical applications, CUPUS shows the best messaging footprint and can thus greatly reduce the energy consumption of the overall MCS solution.

6. Experimental evaluation

In this section we evaluate CUPUS as an enabler for the MCS ecosystem mainly in terms of its end-to-end propagation delay from a data source to a data consumer. This delay consists of the following: processing time on a Mobile Broker, data propagation time on wireless links and processing time on the Cloud Broker. First, we evaluate the scalability of broker implementations when increasing the number of subscriptions both on mobile and cloud broker instances. In addition, we evaluate the elasticity of the cloud implementation. Second, we provide measurements of data propagation delay when either TCP/IP or GCM are used on wireless links. Note that instead of measuring only the processing time on the Cloud Broker we first measure the end-to-end propagation delay within the cloud when CUPUS publishers and subscribers are running on a virtual machine within the cloud since this represents a more realistic performance indicator.

6.1. Cloud broker evaluation

To measure the processing performance of the Cloud Broker we have used the SenseZgAir dataset. Publications generated during the experiment are selected at random from the dataset. As the number of collected subscriptions during the measurement campaign is insufficient to perform the experiment, we generate them in the following way: first we choose at random a measurement from the SenseZgAir associated with a random numerical operator (excluding BETWEEN), and second we generate time-related subscriptions for random time periods during the measurement campaign. The time-related subscriptions are more general and without much overlap, and thus create independent trees which increase the processing time on the Cloud Broker. We assume that the generated subscription set is a good approximation of real subscriptions as it is skewed: Users typically have similar interests, e.g., a sensor reading for a specific gas on a busy location is interesting to many users, while at the same time a large number of subscriptions is interesting to few users, for example all sensor readings for a specific time period.

Experiments investigating the processing performance of the Cloud Broker are done using two different virtual machine instances running on a Dell PowerEdge R720 rack server with the Citrix XenServer virtualization software. The first instance uses 2 vCPU cores with 4 GB of RAM (VM-2), while the second instance has 4 vCPU cores with 8 GB of RAM (VM-4). Each virtual instance uses Debian as the operating system with Java 7 installed. Two different matcher configurations are tested on VM-2 and VM-4. The first configuration starts a single matcher component so that all subscriptions are stored in a single forest data structure ($config_1$). The second configuration consists of five matcher components which evenly divide all subscriptions ($config_5$), each forming its own independent forest structure. The experiment starts by defining the subscriptions and building the forest structure, and afterwards the Cloud Broker starts receiving and processing publications. Incoming publications are forwarded in a round robin fashion to all running matchers in case of $config_5$.

We investigate the data propagation delay in the following experiments. The delay includes the propagation time between a publisher and the Cloud Broker, processing time at the Cloud Broker, and propagation time to deliver a notification to a subscriber. In this set of experiments publishers and subscribers are running on another virtual machine of the rack server to generate processing load for the Cloud Broker. Publications are sent sequentially without any delays causing a constant load on the Cloud Broker in experiment 1, while for experiment 2 we generate changing processing load to investigate the elasticity of our implementation.

Experiment 1: Scalability. To test the scalability of the Cloud Broker, we vary the number of subscriptions from 1 to 20,000. Fig. 10 depicts the average data propagation delay incurred in the cloud with different matcher configurations ($config_1$ and $config_5$). Dashed lines represent the performance of the Cloud Broker on VM-2, while solid lines represent results obtained on VM-4. In our tests the best performance is obtained with $config_5$ running on VM-4. It is interesting to notice that an increase in the available resources did not result in significant improvement of the average data propagation delay when we use $config_1$, but we obtain significantly improved results with $config_5$ running on a VM-4 compared to $config_5$ running on a VM-2. As expected, the worst results are obtained when $config_5$ runs on VM-2: We notice performance degradation of the Cloud Broker when a large number of spawned

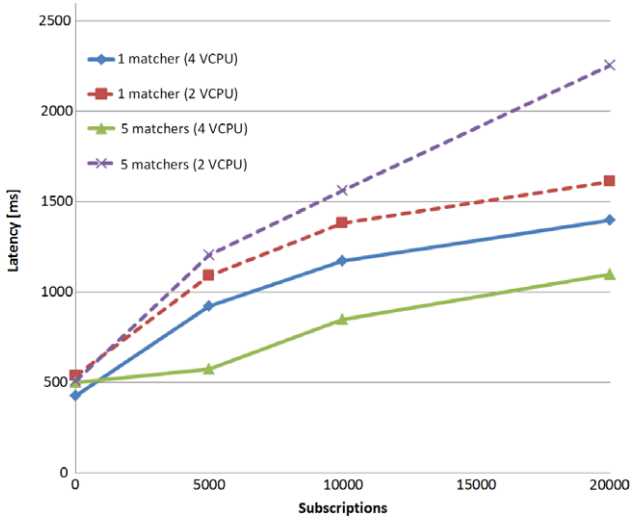


Fig. 10. Data propagation delay in the cloud when increasing the number of subscriptions under constant publication rate.

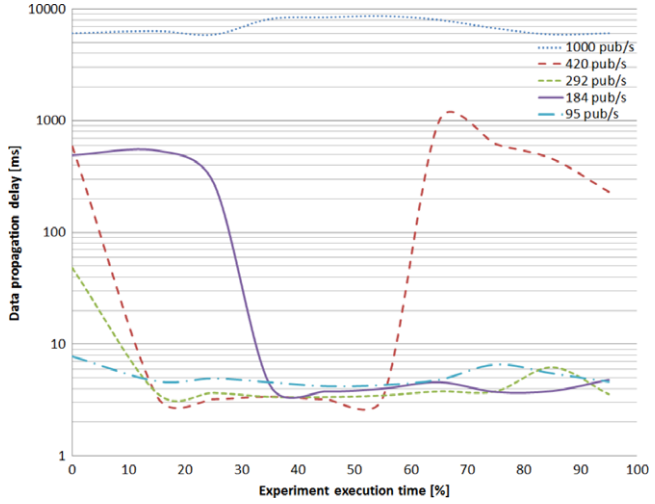


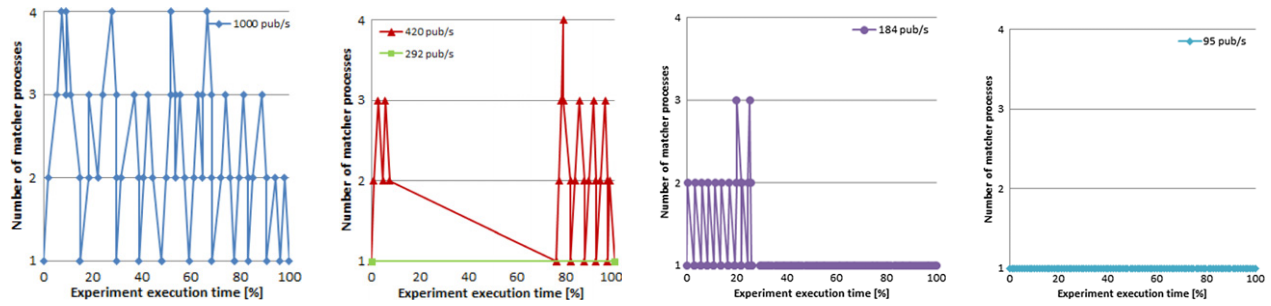
Fig. 11. The data propagation delay in the cloud depending on the incoming publication frequency with a dynamic Cloud Broker configuration.

processes are running in parallel on only 2 vCPU cores. It can be concluded that the Cloud Broker scales well when adequate resources are assigned to specific configurations, while the best-performing average propagation delay in the cloud is up to 1 s ($config_5$ running on a VM-4), even when 20,000 subscriptions are processed simultaneously. In case of a larger number of subscriptions, more cloud resources and an increasing number of matchers should be used.

Experiment 2: Elasticity. This experiment evaluates the Cloud Broker performance with dynamic configurations to inspect solution elasticity, i.e., whether it adapts well to increasing/decreasing processing load. The Cloud Broker is running on the VM-4 machine while a subscriber and publisher are located on VM-2. We use a setup with 5000 subscriptions and 50,000 publications in this experiment to investigate the data propagation delay while varying the publication frequency from 95 publications per second to 1000 publications per second. We also monitor matcher configurations which are dynamically created by the Cloud Broker: The number of active matchers in the system changes autonomously to adjust Cloud Broker performance to the current processing load. Overloaded matchers generate requests for splitting of matchers, while underloaded matchers try to merge. Splitting events are triggered by high processing load on a matcher that creates a new matcher to take over part of subscriptions from an overloaded matcher. Merging events are triggered on underloaded matchers which are to be released while other matchers take over their subscriptions. Similar to the previous configuration all subscriptions are static and defined prior to experiment execution.

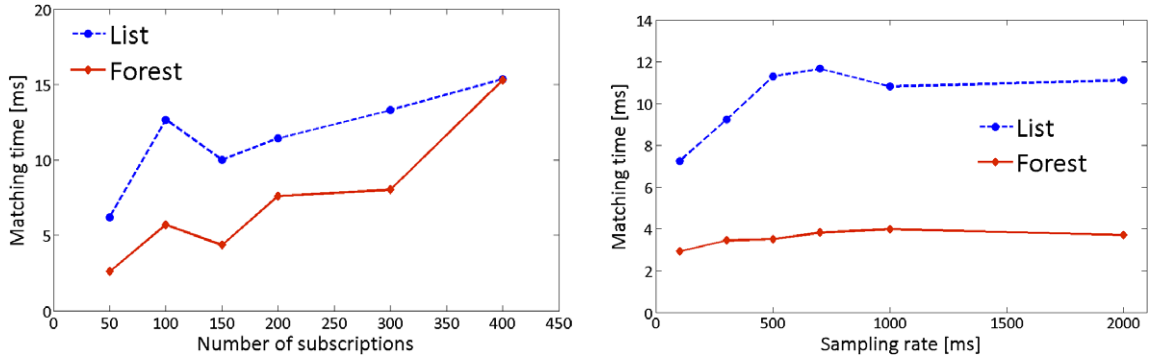
Fig. 11 depicts the data propagation delay in the cloud depending on different publishing frequencies. Our tests show that under high publishing frequency the Cloud Broker performance with dynamic configuration has higher propagation delay compared to the fixed configuration. This occurs mostly because of the maintenance job to spawn or destroy matchers, but possible benefits from this approach outweigh the shortcoming. With a dynamic configuration, an optimal number of matcher processes is used, thus allowing significant savings of cloud resources during the period of low processing load. We can notice that the data propagation delay drops from a high level at the beginning of the experiment to lower levels later during the experiment because the engine autonomously reconfigures to use a minimal number of matcher processes with respect to the processing load. Sudden increases of the propagation delay are mostly related to splitting and merging requests which cause reconfiguration of matcher components. This is clearly visible in the experiment with the rate of 420 publications per second.

Fig. 12 shows the Cloud Broker configurations during our experiment with varying publishing load. Fig. 12(a) depicts the number of matchers running in the case of very high load: The number of matchers changes frequently during the experiment since the engine constantly attempts to balance the number of matchers and their processing load, but it does not have adequate cloud resources to cope with the load. This is the main reason for the observed significant increase of the data propagation delay. Fig. 12(b) presents the engine configuration for two medium publishing loads. We can observe a strong correlation between propagation delays and configuration changes of the Cloud Broker, which is especially visible in the experiment with publication rate of 420 publications per second. After the initial reconfiguration



(a) High publishing load. (b) Medium publishing load. (c) Low publishing load. (d) Very low publishing load.

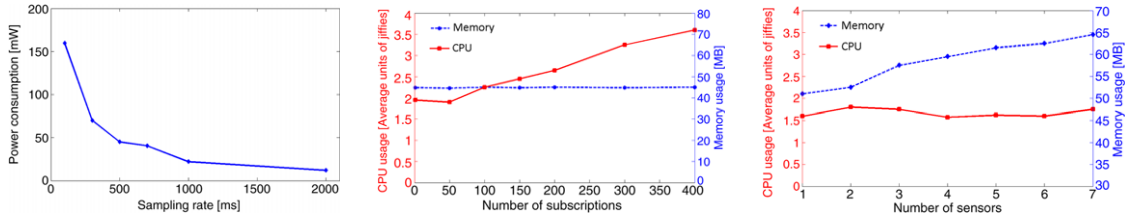
Fig. 12. The cloud broker configuration during an experiment with varying publishing load.



(a) Matching time compared to the number of subscriptions.

(b) Matching time compared to the publication frequency.

Fig. 13. Matching time on a mobile broker.



(a) Energy consumption compared to the publication frequency.

(b) CPU and memory consumption compared to the number of subscriptions.

(c) CPU and memory consumption compared to the number of sensors.

Fig. 14. Resource consumption on a mobile broker.

of the engine, due to high load on a single matcher, the splitting event is triggered causing a sudden high propagation delay. It is interesting to observe the configuration changes in case of 292 publications per second although the number of used matcher processes is constantly one. One can notice that this graph does not contain any changes which represent the splitting or merging events, concluding that the processing load of the matcher is below the threshold for creating a new matcher. The experiment with low publishing load depicted in Fig. 12(c) started with two matchers running in parallel, but the Cloud Broker rearranged its configuration to use only one matcher since it was sufficient to process all incoming messages. In the experiment with very low load depicted in Fig. 12(d) we can see that a single underloaded matcher processes all publications. The data propagation time is higher than expected due to unsuccessful merge requests.

We can conclude that dynamic matcher configurations in case of adequate computing resources can keep the data propagation delay in the cloud below 1 s, even in the case of merging and splitting events which cause significant overhead.

6.2. Mobile broker evaluation

To evaluate the matching process on the Mobile Broker we have performed scalability tests with an increasing number of subscribers and with different subscription data structures, as defined in Section 3.2. Additionally, we have measured the resource consumption on a mobile device during the data acquisition and preprocessing steps. Experiments were conducted using a Samsung Galaxy S4 smartphone with the Android Jelly Bean 4.3 operating system.

The matching time is the processing time required to determine whether a received sensor reading satisfies constraints of at least one subscription which is active on the Mobile Broker. The matching time is measured while varying the number of subscriptions from 100 to 400 while keeping the publication rate constant at 1 s. The set of active subscriptions is generated at random within the limits of sensor sensing capabilities. Each subscription constraint is constructed using a random value and one of the relational operators ($\geq$, $\leq$, $=$). The results of the first experiment (Fig. 13(a)) show that the forest structure enables better matching time compared to the list of subscriptions, especially when a small number of subscriptions is processed by a Mobile Broker. When a large number of subscriptions is processed, the difference in performance between the two data structures is reduced due to the matching function optimization which searches for a single matching subscription. Note that the smartphone used in this experiment could not handle more than 400 subscriptions. The main reason is limited support for recursion in the device's process virtual machine (Dalvik). However, a large number of subscriptions is not expected to be matched on a Mobile Broker in practice due to the announce mechanism.

In the second experiment, we measure the matching time while varying the sampling period to obtain sensor readings from 100 to 2000 ms while processing a constant number of 100 subscriptions which were generated at random, as in the previous experiment. Fig. 13(b) shows that the matching time is not significantly affected by the sensor sampling rate, as it depends mostly on the number of active subscriptions on the Mobile Broker.

Furthermore, in the third experiment we measure the average power consumption of the CUPUS Android application while matching 500 publications with a fixed number of 100 subscriptions. During the experiment we varied the publication sampling rate from 1/100 to 1/2000 ms. The PowerTutor application [28] was used for measuring the application power consumption. Fig. 14(a) depicts the obtained experimental results. Similar to [29], the power consumption decreases exponentially when decreasing the sampling rate. Similar to the findings in [30,31], high sampling frequencies increased the power consumption since devices are not allowed to enter a low power mode.

Fig. 14(b) and (c) show CPU and memory usage when the number of subscriptions and built-in sensors as data sources in the experiment increases. The CPU usage in this experiment is measured in jiffies, which are defined as the time between two ticks of the system timer interruption. Obviously, it is not an

Table 3
Measuring the delay on wireless interfaces (WiFi and 3G).

	TCP-WiFi (s)	GCM-WiFi (s)	WiFi [32]
1 s interval	0.294	0.286	1.5×10^{-4}–0.068 s
0.1 s interval	0.255	0.215	
	TCP-3G (s)	GCM-3G (s)	GPRS [32]
1 s interval	0.752	0.394	0.4–7.16 s
0.1 s interval	0.612	11.47	

absolute time interval unit, since its duration depends on the frequency of clock interruptions on a particular hardware platform. From both graphs we can notice that the CPU consumption generally grows when increasing the number of subscriptions or sensors generating publications. However, memory usage remains at a constant level since the processing of a newly created publication does not require an additional amount of memory when the number of active subscriptions is fixed.

The previous experiments show that the processing delay on a Mobile Broker is in the range of milliseconds and thus does not significantly influence the end-to-end data propagation delay, while it scales well to an increasing number of data sources, publication frequency and number of active subscriptions.

6.3. Delay on wireless links

As MCS applications are used on mobile devices, we need to measure the data transmission delay introduced on wireless links. Note that detailed performance of wireless communication is outside the scope of this paper; instead we measure the overhead introduced by a wireless communication link to evaluate end-to-end propagation delay for MCS applications. A detailed analysis of performance characterization in real heterogeneous wired/wireless network is given in [32]. We have measured the delay in terms of data propagation time when pushing data objects from the Cloud Broker to a Mobile Broker running on a mobile device. The data is pushed sequentially every 1 or 0.1 s while the CUPUS middleware is not under heavy load.

The experiments are performed on both WiFi and 3G mobile device interfaces and the obtained results are shown in Table 3. In parallel to the obtained results we provide the baseline performance of the WiFi and GPRS communication technology presented in [32]. As the baseline we use half of the Round Trip Time (RTT) for end-to end paths (Ethernet-to-WiFi and Ethernet-to-GPRS), irrespective of the application layer. Both the TCP/IP and GCM solution show approximately the same performance over WiFi links irrespective of the data publication rate. When the interval between publications is large, the delay is a bit larger because the device enters a power save mode. However, on 3G links, the results are a bit different. TCP/IP shows approximately the same performance irrespective of publication frequency, while GCM suffers from a larger delay when the data is published frequently. Thus, GCM is not the best strategy for push-based data delivery to mobile devices in case of high messaging frequency on a 3G interface. However, it performs well with lower publication frequencies and is usually not blocked by mobile operators. Note that for comparison we provide the baseline performance of a 2G mobile network (GPRS), while we tested the delay using a 3G mobile network.

To conclude, we can estimate the true end-to-end data propagation delay in CUPUS by combining the delay measured under high Cloud Broker load with the data propagation delay on wireless links. It represents the time period which a user can experience when using CUPUS in MCS applications to receive sensor readings acquired by other users and relevant to his/her context. The propagation delay can be in the range of 1–2 s when using TCP/IP. The situation is similar when using GCM but only in case of low notification frequency. The end-to-end data propagation delay of 1–2 s should be acceptable for most MCS use cases, e.g., environmental or traffic monitoring application, which require relatively simple continuous processing operators on sensor data stream as those provided by typical publish/subscribe subscription languages.

7. Related work

There are a large number of MCS applications published in the literature which utilize mobile IoT sensing. The overview and comparison of such applications are available in [6,1]. The authors of [33,1] identify the need for a unified architecture for MCS applications, and provide an outline of such architecture. Here we want to depict the main characteristics of MCS platforms which are comparable to the proposed MCS ecosystem which is to our knowledge the first solution to leverage the well-known announce–subscribe–publish pattern in the context of large-scale and context-aware MCS applications.

The CenceMe [34] project is one of the early works on sensing to infer information related to human activity which was the first to combine presence inference with sharing of this information through social networks. This approach enables personal visualization of human activity with a possibility for group interaction, but does not require a near real-time processing approach proposed by CUPUS. PEIR, the Personal Environmental Impact Report, is one of the first platforms for opportunistic sensing [35] which estimates personal exposure to pollution and environmental impact based on location data sampled from user mobile phones. PEIR architecture is purely client–server where mobile devices are applied for sensor data collection and are automatically uploaded to the server for further processing. The collected data and exposure metrics are available for personal usage and can be viewed through a web interface. The PEIR system is also not real-time in nature such that it does not sent notifications to users and does not offer data filtering options on mobile phones.

Medusa [36] is an interesting framework for collecting user experiences from users who contribute sensor data for self-reflection or environmental awareness. A requestor gives tasks to smartphone users to acquire and annotate sensor data while Medusa focuses on ensuring the privacy of sensor data contributions. It is also not real-time and does not deal with the problem of redundant data.

Lifestreams is a modular and open source platform designed to explore and analyze personal data streams captured from mobile phones [37]. In particular, Lifestreams modules include feature extraction from raw sensor data, feature selection, inference and data visualization. It is built on top of Ohmage, a personal data collection platform for the acquisition of a rich and heterogeneous dataset from mobile devices, both in an opportunistic and participatory way. Lifestreams main goal is to identify key behaviors and trends which are relevant to an individual's health as well as to enable researchers from health domains to identify behavioral-indicators from large volumes of raw and heterogeneous data streams. In that respect it needs to collect as much data as possible from an individual and thus has different goals compared to CUPUS.

The MobiSense [38] platform is used for activity recognition and applies the client–server implementation with asynchronous messages used for adjusting the sensing task assignments. MobiSense uses topic-based messaging for adjusting the sampling frequency of sensors, stores the data locally and uploads it when the network connection is available, but compared to CUPUS, it is not used for data acquisition and data filtering on mobile devices, nor for delivery of near real-time notifications.

Another example is Pogo [39], a middleware infrastructure supporting easy access to sensor data on mobile phones. Pogo is built upon the principles of publish/subscribe, it uses simple topic-based subscriptions to manage access to sensor data and reports significant energy gains due to topic-based filtering of sensed data on mobile devices. CUPUS also aims to achieve energy-savings on mobile devices and sensors by suppressing the transmission of redundant and irrelevant data into the cloud and by adjusting the sensing process.

In the work reported in [40], the authors propose a multidimensional context-aware social network architecture for mobile crowd sensing applications. They propose a mobile ecosystem that integrates context information from multidimensional sources and provides semantic matching of collected data for personalized understanding of user requirements, which can improve data delivery rate and reduce energy consumption on mobile devices.

Another example is Mobile Sensor Data EngiNe (MOSDEN) [41], a collaborative mobile sensing framework for developing and deploying opportunistic sensing applications. MOSDEN is designed to operate on Android-based smartphones and capture and share sensed data between multiple distributed applications and users. By supporting processing and storage on end user smartphone devices, the platform aims to reduce the necessary data transmission to a centralized server, consequently achieving bandwidth and energy efficiency. The problem of energy efficiency in MCS applications is addressed in [42] where the authors present an approach for model-driven adaptive environmental sensing. They aim to reduce the amount of redundant sensor readings by maintaining local models of expected sensor readings on mobile devices and pushing updates to the back-end server only when predicted values do not match actual sensor readings, which consequently reduces the amount of communication between client devices and a back-end data collection server. This approach is complementary to the selective and flexible acquisition and filtering of sensor data on mobile devices proposed in the CUPUS middleware.

The CitiSense project [43] aims to monitor environmental conditions and air pollution to which users are exposed to during their daily activities. The mobile phone analyzes the data from a small wearable sensor to provide real-time feedback to the user about the ambient air pollution. All collected data is sent to the backend server for further analysis and visualization so that users can observe their personal historical data. It is vital to notice that the CitiSense real-time alerting functionality is performed exclusively based on local readings from sensors connected to mobile phones, and does not enable real-time data sharing between the users to distribute alerts due to readings collected by other users. In that respect it does not require any (near) real-time notifications from the backend server to mobile devices.

To summarize, publish/subscribe middleware is used mainly for either data dissemination within wireless sensor networks [44], or for data acquisition from stationary Internet-connected objects [45,46]. There are few publish/subscribe solutions addressing MIOs as well as their interaction with the cloud. The authors of [47] propose a publish/subscribe platform for opportunistic data collection from stationary sensors which is an approach the most similar to ours. They present a hiking trail application where mobile phones are applied as "mules" to selectively gather data from stationary sensors, in contrast to our middleware which enables collecting of environmental data from mobile and wearable sensors carried by users and control over the sensing coverage and battery usage for devices in an MCS application scenario. Lim et al. [48] focus on energy-efficient data acquisition from multiple sensor streams such that they choose the most favorable source with respect to continuous queries, but in contrast to our solution do not relate the processing performed on one device with the cloud or other mobile devices. Sheng et al. [49] present theoretical and simulation results to evaluate algorithms for determining energy-efficient sensing schedules of moving sensors while our approach provides the means to implement such algorithms in real systems.

We have already identified publish/subscribe middleware as an adequate enabler of MCS applications [10] where we present the two-tier communication model comprising cloud brokers and mobile brokers. The initial version of this paper is published in [11] which proposes an MCS ecosystem based on the publish/subscribe primitives and focuses on evaluating CUPUS performance on mobile devices. This paper is significantly different from [11], both in terms of contribution and content because this paper presents an overall evaluation of CUPUS in the context of an MCS application for air quality monitoring.

8. Conclusion

The paper presents an ecosystem for MCS which relies on the CloUd-based PUblish/Subscribe middleware (CUPUS) to acquire sensor data from mobile devices. We outline the underlying model and design decisions of the CUPUS middleware which enables flexible and controllable data acquisition from MIOs, as well as near real-time processing of sensor data streams along with delivery of notifications from the cloud to mobile devices. The middleware is energy-efficient as it suppresses the transmission of sensor readings from MIOs and filters out either redundant or irrelevant sensor readings. To demonstrate the applicability of the proposed ecosystem, we present an environmental monitoring application built on top of CUPUS. It integrates a wearable gas sensor which communicates over a Bluetooth connection with a smartphone running a mobile application to regulate and relay sensor readings to the cloud back-end. The mobile application also receives air quality alerts from the cloud in real-time in accordance with user preferences and context.

We present the comparison of the CUPUS middleware with two standardized messaging solutions, the MQTT protocol and CoAP, with respect to MCS application requirements. Our experimental results with real user traces collected during an air quality measurement campaign in the City of Zagreb show that our MCS solution is scalable when increasing the number of subscriptions which are processed in parallel under constant publication load, while it also adapts well to varying publication load. Additionally, our measurements show that the end-to-end data propagation delay observed by end users is within the range of 1–2 s, which is acceptable for typical MCS usage scenarios. Evaluation of the mobile application has shown that it is scalable and energy-efficient even when processing a large number of subscriptions in parallel and sensor readings with relatively high publication rates. Furthermore, the processing time on a mobile device is within the range of 10 ms for typical loads which does not represent a major overhead for the end-to-end data propagation delay while offering an overall energy-efficient MCS performance.

As future work we plan to investigate hierarchical broker architectures which use mobile brokers on sensor nodes, while mobile devices could act as gateways for a number of sensors and their mobile brokers, e.g., in fog environments. We are currently in the process of adapting our middleware to run on large cloud infrastructures and we need to perform large-scale tests to investigate the influence of cloud management solutions on the CUPUS processing performance and data propagation delay.

Acknowledgments

This work has been partially carried out in the scope of the project ICT OpenIoT Project FP7-ICT-2011-7-287305 co-funded by the European Commission under FP7 program. This work has been supported in part by the Croatian Science Foundation under the project number 8065 (Human-centric Communications in Smart Networks).

References

[1] R.K. Ganti, F. Ye, H. Lei, Mobile crowdsensing: current state and future challenges, IEEE Commun. Mag. 49 (11) (2011) 32–39.

[2] B. Guo, D. Zhang, Z. Yu, Y. Liang, Z. Wang, X. Zhou, From the Internet of things to embedded intelligence, World Wide Web 16 (4) (2013) 399–420.

[3] J.A. Burke, et al., Participatory sensing, in: Proc. of WSW'06 at ACM SenSys 2006, ACM, 2006.

[4] A. Zaslavsky, P. Jayaraman, S. Krishnaswamy, ShareLikesCrowd: Mobile analytics for participatory sensing and crowd-sourcing applications, in: 2013 IEEE 29th International Conference on Data Engineering Workshops, ICDEW, 2013, pp. 128–135.

[5] J. Gubbi, R. Buyya, S. Marusic, M. Palaniswami, Internet of things (IoT): A vision, architectural elements, and future directions, Future Gener. Comput. Syst. 29 (7) (2013) 1645–1660.

[6] N.D. Lane, E. Miluzzo, H. Lu, D. Peebles, T. Choudhury, A.T. Campbell, A survey of mobile phone sensing, Commun. Mag. 48 (9) (2010) 140–150.

[7] L. Skorin-Kapov, K. Pripuzic, M. Marjanovic, A. Antonic, I. Zarko, Energy efficient and quality-driven continuous sensor management for mobile iot applications, in: 2014 International Conference on Collaborative Computing: Networking, Applications and Worksharing, CollaborateCom, 2014, pp. 397–406.

[8] A. Botta, W. de Donato, V. Persico, A. Pescape, On the integration of cloud computing and Internet of things, in: 2014 International Conference on Future Internet of Things and Cloud, FiCloud, 2014, pp. 23–30.

[9] G. Mühl, L. Fiege, P. Pietzuch, Distributed Event-Based Systems, Springer, 2006.

[10] I. Podnar Zarko, A. Antonic, K. Pripužic, Publish/subscribe middleware for energy-efficient mobile crowdsensing, in: Proceedings of the 2013 ACM Conference on Pervasive and Ubiquitous Computing Adjunct Publication, UbiComp'13 Adjunct, ACM, New York, NY, USA, 2013, pp. 1099–1110.

[11] A. Antonic, K. Rozankovic, M. Marjanovic, K. Pripuzic, I. Podnar Zarko, A mobile crowdsensing ecosystem enabled by a cloud-based publish/subscribe middleware, in: Proc. of the 2nd International Conference on Future Internet of Things and Cloud, FiCloud-2014, 2014.

[12] A. Carzaniga, D.S. Rosenblum, A.L. Wolf, Design and evaluation of a wide-area event notification service, ACM Trans. Comput. Syst. 19 (3) (2001) 332–383.

[13] A. Antonic, V. Bilas, M. Marjanovic, M. Matijasevic, D. Oletic, M. Pavelic, I. Podnar Zarko, K. Pripuzic, L. Skorin-Kapov, Urban crowd sensing demonstrator: Sense the zagreb air, in: Proc. of the 22nd International Conference on Software, Telecommunications and Computer Networks, SoftCOM2014, 2014.

[14] Martina Marjanović, Lea Skorin-Kapov, Krešimir Pripužić, Aleksandar Antonić, Ivana Podnar Žarko, Energy-aware and quality-driven sensor management for green mobile crowd sensing, J. Netw. Comput. Appl. (2015) http://dx.doi.org/10.1016/j.jnca.2015.06.023.

[15] M. Sadoghi, H.-A. Jacobsen, BE-tree: an index structure to efficiently match boolean expressions over high-dimensional discrete space, in: Proc. of the 2011 ACM SIGMOD, ACM, New York, NY, USA, 2011, pp. 637–648.

[16] L. Golab, M.T. Ozsu, Issues in data stream management, SIGMOD Rec. 32 (2) (2003) 5–14.

[17] S. Tarkoma, J. Kangasharju, Optimizing content-based routers: posets and forests, Distrib. Comput. 19 (1) (2006) 62–77.

[18] I. Podnar, M. Hauswirth, M. Jazayeri, Mobile push: Delivering content to mobile users, in: J. Bacon, L. Fiege, R. Guerraoui, A. Jacobsen, G. Mühl (Eds.), in Proceedings of the 1st International Workshop on Distributed Event-Based Systems, DEBS'02, 2002.

[19] I. Warren, A. Meads, S. Srirama, T. Weerasinghe, C. Paniagua, Push notification mechanisms for pervasive smartphone applications, IEEE Pervasive Comput. 13 (2) (2014) 61–71.

[20] F. Qian, Z. Wang, Y. Gao, J. Huang, A. Gerber, Z. Mao, S. Sen, O. Spatscheck, Periodic transfers in mobile applications: network-wide origin, impact, and optimization, in: Proceedings of the 21st International Conference on World Wide Web, WWW'12, ACM, New York, NY, USA, 2012, pp. 51–60.

[21] H.-Y. Liu, E. Skjetne, M. Kobernus, Mobile phone tracking: in support of modelling traffic-related air pollution contribution to individual exposure and its implications for public health impact assessment, Environ. Health 12 (2013).

[22] U. Hunkeler, H.L. Truong, A. Stanford-Clark, MQTT-S—A publish/subscribe protocol for wireless sensor networks, in: 3rd International Conference on Communication Systems Software and Middleware and Workshops, 2008. COMSWARE 2008. 2008, pp. 791–798.

[23] Z. Shelby, K. Hartke, C. Bormann, The constrained application protocol (coap), RFC 7252, RFC Editor, June 2014.

[24] K. Hartke, Observing resources in coap, Internet-Draft draft-ietf-core-observe-16, IETF Secretariat, December 2014.

[25] K. Framling, S. Kubler, A. Buda, Universal messaging standards for the iot from a lifecycle management perspective, IEEE Internet Things J. 1 (4) (2014) 319–327.

[26] A. Antonic, M. Marjanovic, P. Skocir, I. Podnar Zarko, Comparison of the CUPUS middleware and MQTT protocol for smart city services, in: Telecommunications (ConTEL), 2015 13th International Conference on, 2015, pp. 1–8. http://dx.doi.org/10.1109/ConTEL.2015.7231225.

[27] S. Kubler, K. Främling, A. Buda, A standardized approach to deal with firewall and mobility policies in the IoT, Pervasive Mob. Comput. 20 (2015) 100–114.

[28] L. Zhang, B. Tiwana, Z. Qian, Z. Wang, R.P. Dick, Z.M. Mao, L. Yang, Accurate online power estimation and automatic battery behavior based power model generation for smartphones, in: Proceedings of the Eighth IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis, CODES/ISSS'10, ACM, New York, NY, USA, 2010, pp. 105–114.

[29] C. Perera, A.B. Zaslavsky, P. Christen, A. Salehi, D. Georgakopoulos, Capturing sensor data from mobile phones using global sensor network middleware, in: PIMRC, IEEE, 2012, pp. 24–29.

[30] C. Thompson, D.C. Schmidt, H.A. Turner, J. White, Analyzing mobile application software power consumption via model-driven engineering, in: C. Benavente-Peces, J. Filipe (Eds.), PECCS, SciTePress, 2011, pp. 101–113.

[31] A. Krause, M. Ihmig, E. Rankin, D. Leong, S. Gupta, D. Siewiorek, A. Smailagic, M. Deisher, U. Sengupta, Trading off prediction accuracy and power consumption for context-aware wearable computing, in: Proceedings of the Ninth IEEE International Symposium on Wearable Computers, ISWC'05, IEEE Computer Society, Washington, DC, USA, 2005, pp. 20–26.

[32] A. Botta, A. Pescapè, G. Ventre, Quality of service statistics over heterogeneous networks: Analysis and applications, European J. Oper. Res. 191 (3) (2008) 1075–1088.

[33] G. Chatzimilioudis, A. Konstantinidis, C. Laoudias, D. Zeinalipour-Yazti, Crowdsourcing with smartphones, IEEE Internet Comput. 16 (5) (2012) 36–44.

[34] E. Miluzzo, N.D. Lane, K. Fodor, R. Peterson, H. Lu, M. Musolesi, S.B. Eisenman, X. Zheng, A.T. Campbell, Sensing meets mobile social networks: The design, implementation and evaluation of the cenceme application, in: Proceedings of the 6th ACM Conference on Embedded Network Sensor Systems, SenSys'08, ACM, New York, NY, USA, 2008, pp. 337–350.

[35] M. Mun, S. Reddy, K. Shilton, N. Yau, J. Burke, D. Estrin, M. Hansen, E. Howard, R. West, P. Boda, PEIR: the personal environmental impact report, as a platform for participatory sensing systems research, in: Proc. ACM/USENIX Int. Conf. Mobile Systems, Applications, and Services, MobiSys, Krakow, 2009.

[36] M.-R. Ra, B. Liu, T.F. La Porta, R. Govindan, Medusa: A programming framework for crowd-sensing applications, in: Proceedings of the 10th International Conference on Mobile Systems, Applications, and Services, MobiSys'12, ACM, New York, NY, USA, 2012, pp. 337–350.

[37] C.-K. Hsieh, H. Tangmunarunkit, F. Alquaddoomi, J. Jenkins, J. Kang, C. Ketcham, B. Longstaff, J. Selsky, B. Dawson, D. Swendeman, D. Estrin, N. Ramanathan, Lifestreams: A modular sense-making toolset for identifying important patterns from everyday life, in: Proceedings of the 11th ACM Conference on Embedded Networked Sensor Systems, SenSys'13, ACM, New York, NY, USA, 2013, pp. 5:1–5:13.

[38] P. Wu, J. Zhu, J.Y. Zhang, MobiSens: A versatile mobile sensing platform for real-world applications, Mob. Netw. Appl. 18 (1) (2013) 60–80.

[39] N. Brouwers, K. Langendoen, Pogo, a middleware for mobile phone sensing, in: Proceedings of the 13th International Middleware Conference, Middleware'12, Springer-Verlag New York, Inc., New York, NY, USA, 2012, pp. 21–40.

[40] X. Hu, X. Li, E.-H. Ngai, V. Leung, P. Kruchten, Multidimensional context-aware social network architecture for mobile crowdsensing, IEEE Commun. Mag. 52 (6) (2014) 78–87.

[41] P.P. Jayaraman, C. Perera, D. Georgakopoulos, A.B. Zaslavsky, MOSDEN: A scalable mobile collaborative platform for opportunistic sensing applications, CoRR abs/1405.5867.

[42] N. Nikzad, J. Yang, P. Zappi, T.S. Rosing, D. Krishnaswamy, Model-driven adaptive wireless sensing for environmental healthcare feedback systems, in: 2012 IEEE International Conference on Communications (ICC), IEEE, 2012, pp. 3439–3444.

[43] E. Bales, N. Nikzad, N. Quick, C. Ziftci, K. Patrick, W. Griswold, Citisense: Mobile air quality sensing for individuals and communities design and deployment of the citisense mobile air-quality system, in: 2012 6th International Conference on Pervasive Computing Technologies for Healthcare, PervasiveHealth, 2012, pp. 155–158.

[44] P. Costa, C. Mascolo, M. Musolesi, G.P. Picco, Socially-aware routing for publish–subscribe in delay-tolerant mobile ad hoc networks, IEEE J. Sel. Areas Commun. 26 (5) (2008) 748–760.

[45] E. Souto, G. Guimaraes, G. Vasconcelos, M. Vieira, N. Rosa, C. Ferraz, J. Kelner, Mires: a publish/subscribe middleware for sensor networks, Pers. Ubiquitous Comput. 10 (1) (2005) 37–44.

[46] O. Jurca, S. Michel, A. Herrmann, K. Aberer, Continuous query evaluation over distributed sensor networks, in: ICDE, 2010, pp. 912–923.

[47] X. Tong, E.C.H. Ngai, A ubiquitous publish/subscribe platform for wireless sensor networks with mobile mules, in: DCOSS, 2012, pp. 99–108.

[48] L. Lim, A. Misra, T. Mo, Adaptive data acquisition strategies for energy-efficient, smartphone-based, continuous processing of sensor streams, Distrib. Parallel Databases 31 (2) (2013) 321–351.

[49] X. Sheng, J. Tang, W. Zhang, Energy-efficient collaborative sensing with mobile phones, in: INFOCOM, 2012, pp. 1916–1924.

Aleksandar Antonić is a Ph.D. student and Research Associate at the University of Faculty of Electrical Engineering and Computing. He was a research intern at the Digital Enterprise Research Institute (DERI), National University of Ireland, Galway (2011). He has experience of participating in projects funded by the EU Framework Programme, Croatian Ministry of Science, Education and Sports or industry. His interests comprise large-scale distributed systems, real-time data stream processing and information retrieval and recommender systems.

Martina Marjanović received her M.Sc. degree in information and communication technology from the University of Zagreb in 2013. Currently she is a Ph.D. student and Research Associate at the Department of Telecommunications, Faculty of Electrical Engineering and Computing, University of Zagreb. She was a guest student at the Vienna University of Technology (TU Wien) in 2012. Her interests comprise mobile crowd sensing architectures and applications, contextual services and quality of service in Internet of Things.

Krešimir Pripužić received his Ph.D. degree in computer science from the University of Zagreb in 2010. Currently he is an Assistant Professor at the Department of Telecommunications, Faculty of Electrical Engineering and Computing, University of Zagreb. He spent academic year 2006–2007 at the Distributed Information Systems Laboratory at EPFL (Ecole Polytechnique Fédérale de Lausanne), Switzerland, as a scholarship holder of the Swiss Government scholarship for university, fine arts and music schools for foreign students. He has experience of participating in projects founded by the EU Framework Programme, Croatian Ministry of Science, Education and Sports or industry. He has co-authored over 25 scientific journal and conference papers. He is a member of IEEE Computer Society, IEEE Communications Society and IEEE Geoscience & Remote Sensing Society. His interests comprise algorithms and data structures, large-scale distributed systems (publish/subscribe, P2P networks), real-time data stream processing, big data, information retrieval and information filtering systems (recommender systems).

Ivana Podnar Žarko is an Associate Professor at the University of Zagreb Faculty of Electrical Engineering and Computing. She was a Guest Researcher (2000) and Research Associate (2001) at the Technical University of Vienna, and a Postdoctoral Researcher (2005–2006) at the Swiss Federal Institute of Technology in Lausanne (EPFL). Her research interests are in the area of large-scale distributed information systems, in particular, data stream processing and publish/subscribe middleware, Internet of Things, recommender systems and information retrieval in decentralized environments. Currently she leads the University of Zagreb team which participates in the EU-funded FP7 project "OpenIoT: Open Source Solution for the Internet of Things into the Cloud". She has published around 50 scientific papers in journals and conference proceedings in the area of distributed information systems and serves as a peer reviewer for a number of international journals and conferences. She is a member of the IEEE and Section Chair of the IEEE Communications Society Croatia Chapter.