Cuttlefish: A Flexible and Lightweight Middleware for Combining Heterogeneous IoT Devices

Andreas Pamboris
University of Central Lancashire and JARVIC LTD
apamboris@uclan.ac.uk, apamboris@jarvic.eu

Charalampos Kozis
Cyprus University of Technology
ca.kozis@edu.cut.ac.cy

Herodotos Herodotou
Cyprus University of Technology
herodotos.herodotou@cut.ac.cy

***Abstract*—The Internet of Things (IoT) extends connectivity beyond traditional computing devices to different types of smart objects, equipped with various sensors and actuators. These objects range from smart lightbulbs and thermostats to smart watches and fitness trackers, or even heavy machinery used in various industrial sectors. Due to device heterogeneity, the complexity of developing applications that require the collection and sharing of data across multiple IoT devices is high, as developers need to be familiar with a diverse set of supported services and APIs. While existing approaches have proposed solutions to this challenge, they rely on the use of resource-intensive cloud-based components, they do not offer the degree of extensibility desired by developers, and they often trade off some of the richness of real-time data for ease of use. Cuttlefish is a flexible and lightweight middleware that offers a unified API to help with the development of applications that utilize multiple heterogeneous IoT devices. It abstracts away much of the complexity involved with orchestrating different devices at runtime. At the same time, it avoids the aforementioned caveats of existing approaches through a simple and efficient design, yet one that offers a rich set of capabilities to developers.**

***Index Terms*—IoT Middleware, IoT Communication Protocol**

I. INTRODUCTION

Over time, many have proposed different types of middleware to support the development and operation of distributed systems of different kinds [1]–[7]. With the rise of the Internet of Things (IoT), new requirements for this kind of middleware have emerged. The new status quo involves an abundance of different types of connected devices, offering a wide range of capabilities, but also numerous ways of interaction with them. These devices vary from home and industrial appliances, environmental monitoring stations, smart wearables, and other embedded electronic devices. They typically contain different types of sensors and actuators, which, however, are exposed to third-party clients through different services and APIs.

IoT applications rely heavily on the exchange of data between such (often diverse) devices. To date, the number of connected devices is estimated to roughly 11 billion units, and is forecasted to grow significantly in the next few years [8]. The rate at which new devices join the IoT landscape suggests the need for supporting seamlessly the cooperation and coordination of heterogeneous platforms. This entails acquiring sensed data and effecting actions across multiple devices with ease in real time. Ideally, developers should not have to familiarize themselves with all different types of services and APIs exposed by various platforms.

To this end, previous work has proposed different types of middleware to facilitate the development of IoT applications across heterogeneous devices [9] (discussed in Section II). However, to the best of our knowledge, no existing approach has managed to combine all of the following desired IoT middleware features: (i) no reliance on cloud resources during operation; (ii) support for dynamic service discovery; (iii) support for device composition and local processing; and (iv) support for user security and privacy when dealing with sensitive data.

This paper presents a new middleware for IoT applications, coined Cuttlefish, which enables developers to easily orchestrate heterogeneous IoT devices, while addressing all aforementioned challenges. Cuttlefish comprises three main interconnected components: the Clients, the Manager, and the Devices. *Clients* use a unified API through which applications can obtain information about connected devices and their capabilities (sensors and actuators), but also query sensor data and execute actuator commands in real time. The *Manager* is responsible for managing all connected devices and processing corresponding requests issued by Clients based on a publish-subscribe model. Finally, *Devices* are the actual IoT endpoints that have access to different types of sensors and actuators.

The main contribution of Cuttlefish lies in its ability to support all main features desired by an IoT middleware (such as dynamic service discovery, device composition, and local processing), while remaining lightweight enough for it to be hosted even on resource-constrained devices (along with the corresponding IoT applications built on top of it). As a result, it requires minimum deployment provisioning, which directly impacts the associated deployment cost and efforts. At the same time, it provides users with full control over the data collected and shared through the middleware, by eliminating the need for relaying data to machines that are not explicitly managed by the user (e.g., cloud-based servers). Cuttlefish achieves this by adopting a simple and efficient architecture design, where all communications and data exchanges happen through a lightweight publish-subscribe engine.

The remainder of this paper is organized as follows: Section II discusses existing IoT middleware solutions and

This work was co-funded by the European Regional Development Fund and the Republic of Cyprus through the Research Promotion Foundation (STEAM Project: INTEGRATED/0916/0063 and RABIT Project: STARTUPS/0618/0053).

TABLE I
COMPARING SERVICE-, CLOUD-, AND ACTOR-BASED IOT MIDDLEWARE WITH CUTTLEFISH

Feature	Service-based	Cloud-based	Actor-based	Cuttlefish
Deployable on Cloud and powerful servers	Yes	Yes	Yes	Yes
Deployable on resource-constrained devices	No	No	Yes	Yes
Data accessibility mechanism	Web Apps/SDK	RESTful APIs	Event/Actor-based	Direct stream-based
Support for service discovery	Yes	Limited	No/Limited	Yes
Support for device composition	No/Limited	No/Limited	Limited	Yes
Support for local processing/aggregation	No	No	Yes	Yes
Security and privacy	Access control	Access control	Limited	High

compares them to Cuttlefish; Section III presents the overall architecture of Cuttlefish; Section IV discusses the Cuttlefish communication protocol; Section V presents some preliminary evaluation results; and Section VI concludes this paper.

II. Background and Related Work

The increasing popularity of IoT systems has lead to the rise of a plethora of IoT middleware, that is software systems designed to act as intermediary between IoT devices and applications [9], [10]. The main purpose of IoT middleware is to enable seamless connectivity for a large number of heterogeneous IoT devices and simplify the application development. Many IoT middleware platforms have been developed in recent years, each proposing a different architecture and programming abstraction. The current architectures can be categorized into three main classes, namely service-based, cloud-based, and actor-based middleware [9].

Service-based IoT middleware such as Hydra [11], XGSN [12], and SATware [13] are inspired by the service oriented architecture and abstract the functionalities of Things as software services [14]. The typical architecture consists of three layers, the physical plane (sensors and actuators), the service plane, and the applications. All collected data from the physical plane is transmitted to the service plane for processing and storage. Applications are then consuming data and functionality via Web services or platform-specific SDKs. Such middleware are typically heavyweight and deployed on the Cloud or powerful gateways [9]. They are not designed to be deployable on resource-constrained IoT devices and, hence, do not support local processing or aggregation. In addition, device composition support (i.e., the ability to connect and integrate cross-vendor IoT devices) is fairly limited. On the other hand, service-based middleware work well with unknown or dynamic IoT topologies through the use of discovery methods that are largely based on the traditional service and resource discovery approaches [9].

Cloud-based IoT middleware is the most popular class and employs Cloud services for gathering, processing, analyzing, and serving IoT data [15]. Some prominent examples are Xively [16], Ptolemy [17], Google Fit [18], and Thingspeak [19]. While such middleware can provide powerful features and good performance scalability, they typically limit the user on the type of IoT devices they can deploy and restrict the applications to use only cloud-supported RESTful APIs. Similar to service-based solutions, cloud-based middleware offer limited composition support and no support for local processing or aggregation. Even though the Cloud offers good permission and user control options, it requires users to trust the Cloud provider to protect the privacy and integrity of their data [9].

Finally, *actor-based IoT middleware* (e.g., Calvin [20], Node-RED [21]) advocate the actor model based on which IoT devices are exposed as reusable actors and are distributed in a network. Actor-based middleware do not enforce a particular communication standard like a RESTful API but rather advocate the use of a particular programming model. These middleware are designed to be lightweight and embedded in both resource-constrained IoT devices and powerful servers. Hence, the computations can be distributed in the network and take place in any layer. However, they offer limited to no support for dynamic service discovery and device composition, while security and privacy are typically lacking due to the distributed nature of the model.

Table I summarizes the key features of the three aforementioned middleware classes and offers a direct comparison with Cuttlefish. In particular, Cuttlefish is a lightweight middleware capable of running on both the Cloud and resource-constrained IoT devices, supporting device composition and local processing. Service discovery is also supported by semantically describing the capabilities of IoT devices and subscribing to the middleware dynamically. Finally, Cuttlefish offers several security features including authentication and encryption, while applications can bypass the middleware and directly consume device data streams, alleviating privacy concerns.

III. The Cuttlefish Architecture

As illustrated in Figure 1, the Cuttlefish architecture consists of three main types of components: the Client, the Manager, and the Devices. A *Client* is used by an application for interfacing with the Cuttlefish middleware through a dedicated Client API. The *Manager* is the central component through which Cuttlefish manages all connected devices and processes requests issued by Clients. *Devices* are the actual IoT endpoints with access to different sensors and/or actuators, which can read data and execute commands, respectively. This section elaborates on these components, while the corresponding APIs are presented in Section IV.

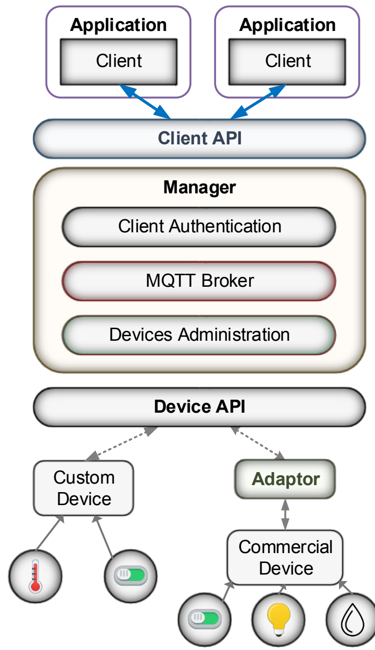


Fig. 1. Cuttlefish System Architecture.

A. Clients

A Client serves as the gateway for an IoT application to the Cuttlefish middleware. Clients are currently implemented in Java and interface with the Manager through the Client API, which can be used by applications to: (i) dynamically retrieve a list of connected Devices and Things (sensors and actuators) from the Manager along with their properties; (ii) register event listeners on available Devices and Things for receiving sensor data; and (ii) execute supported actions on actuators. Clients are also responsible for registering and connecting to the Manager. Responses to requests issued by a Client are asynchronously provided via callback functions.

B. Manager

The Cuttlefish Manager controls and manages Devices and Clients that are part of the Cuttlefish network. It comprises the following three main modules: the *MQTT Broker*, a lightweight publish-subscribe mechanism for exchanging messages and data between all components; the *Client Authentication* module, which is responsible for authenticating clients trying to register to the Manager (through an SQLite database maintained by the Manager); and the *Devices Administration* module, which (i) handles the device registration process when a new IoT device joins the network, (ii) keeps track of the availability of devices at runtime, and (iii) continuously monitors the status of the sensors/actuators of connected devices.

A unique aspect of Cuttlefish is that all communication between Clients, the Manager, and Devices is done through *Message Queuing Telemetry Transport (MQTT)* [22]. MQTT is a lightweight publish-subscribe protocol designed for low-power devices and unreliable networks. Through the MQTT protocol, messages are published to topics others subscribe to. It requires minimum network bandwidth and runs on devices with limited resources, while ensuring a good level of reliability and assurance of message delivery. The MQTT protocol involves a MQTT Broker/Server, MQTT Clients, and topics generated from clients. Multiple clients may subscribe to the same topics and may also publish messages to them via the broker. Its simple design allows any type of networked device to connect to the broker and share data. Cuttlefish employs MQTT to provide Clients with fast access to streams of sensor data generated by Devices, as well as the ability to send requests and control actions to the Manager.

The choice of MQTT as the main mechanism for communication and data sharing was mainly guided by the fact that it is simple, flexible, and lightweight, all of which align perfectly with Cuttlefish's objectives. Nevertheless, MQTT was also chosen due to some of its more interesting features in terms of supported functionality. Through MQTT, clients are able to subscribe to topics by either defining the exact topic of interest or using *wildcards*. The latter approach allows for subscribing to a collection of topics that fall under a specified topic category. For example, a client may subscribe to 'vitals/+', which will result in receiving messages from publishers to both the 'vitals/HeartRate' and 'vitals/Temperature' topics.

Another important feature of MQTT lies in its ability to tune the level of Quality of Service (QoS) desired, thus trading off some of the latency and bandwidth requirements of the underlying publish-subscribe mechanism employed. More precisely, QoS in MQTT is defined at three distinct levels, which determine how reliably the broker and clients deliver messages. The *QoS 0* level exchanges messages without acknowledging their receipt. *QoS 1* delivers a message at least once and uses acknowledgments. Lastly, *QoS 2* is the most reliable level as it delivers a message using a four-step handshake protocol. By default, Cuttlefish uses *QoS 1*, however, this can be configured accordingly by developers depending on the application's requirements.

C. Devices

A Device in Cuttlefish abstractly represents an IoT endpoint with access to a set of sensors and/or actuators. In other words, we differentiate Devices from *Things* (i.e., raw sensors or actuators), which is what existing approaches typically consider as an endpoint. A Device in Cuttlefish can either be: (i) a custom standalone device with sensors and/or actuators physically attached to (or embedded within) it, or (ii) a commercial device that may transparently manage a subnet of attached devices (e.g., using platform-specific APIs). The former comprise devices that are typically programmable such as Raspberry Pi [23], Arduino [24], and Intel-IoT [25] devices. The latter are typically devices that target automation in various contexts, with some examples being the Philips Hue platform [26], Belkin WeMo Home Automation [27], and Ecobee [28]. Regardless of its type, each Device in Cuttlefish is identified by a unique String ID, which is either the MAC address or UID of the corresponding device.

For Commercial devices, most vendors release specific APIs that developers must use to interact with their products. To handle this type of devices, a Cuttlefish *Adaptor* needs to be implemented, which is tasked with the objective to translate commands issued by Cuttlefish's Device API to the corresponding commands supported by the API of the Commercial device. Thus, as shown in Figure 1, the Adaptor API sits in between the Commercial device and the Manager.

IV. CUTTLEFISH COMMUNICATION PROTOCOL

This section describes all Client and Device activities, focusing on the communication protocol between Cuttlefish components and IoT applications built on top of them.

A. Registration and Administration Processes

1) Client Registration: A Client must be given all required parameters and credentials to be able to connect and register with the Manager. This information includes the Domain/IP and registration port to connect to, as well as appropriate authentication credentials, namely a valid *username* and *password*. Upon successful authentication, the Manager responds back to the Client with a unique *ClientID* that has been assigned to it, and the Client initiates the connection process to the MQTT Broker. The Manager then subscribes to the 'cuttlefish/client/{ClientID}/request' topic, through which it accepts future Client requests. Similarly, the Client subscribes automatically to 'cuttlefish/client/{ClientID}/response' to receive future responses from the Manager.

2) Device Registration: For registering with Cuttlefish, Devices simply publish to a corresponding device registration topic ('cuttlefish/device/{DeviceID}') using their ID/MAC, together with some basic information required by the Manager. This information includes the Device ID, name, type, a set of properties, and lists of available sensors and actuators. After validation and authentication, the Manager responds back with an acknowledgment (ACK). If the process fails for any reason (e.g., due to a network failure), it is repeated until the Device receives an ACK successfully.

Upon successful registration of a Device, all *Things* (i.e., sensors and actuators) attached to it need to register separately. The process is identical to Device registration, however, registration requests are now published to a topic ('cuttlefish/device/{DeviceID}/update'), which is created upon the successful registration of the Device and is specific to the Device at hand. Such requests are accompanied by information regarding each Thing, including its ID, type, and capabilities. For Devices that support the dynamic addition of sensors/actuators, the same topic is used to inform the Manager of the addition. Hence, Cuttlefish supports dynamic addition and removal of both Devices and sensors/actuators on Devices.

3) Heartbeat Mechanism: The Manager periodically checks the status and availability of registered Devices through a standard heartbeat mechanism. This mechanism has Devices periodically send *alive* messages to the Manager; similarly, the Manager responds back with the same message content, ensuring that Devices are also aware of the status of the Manager at all times.

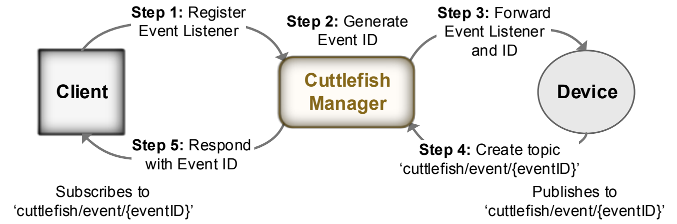


Fig. 2. The processing flow for making a data request to register an event listener for reading sensor data from a Device.

B. Client API

The Client API is a developer's main interface with Cuttlefish. Different types of requests are supported through this API, which are listed in Table II and described next.

1) Find Requests: Find requests are directed to the Manager and aim at identifying dynamically the current availability and capabilities of Devices and Things connected to Cuttlefish. These requests allow for specifying filters based on the Device's ID, type (e.g., thermostat), status (e.g., unreachable), and its supported sensors/actuators (e.g., presence of temperature sensor). The response contains a list with the matched Devices and Things, along with their key properties and capabilities declared during the registration process.

2) Data Requests: Data requests can only be directed to Devices with access to sensors. They are responsible for managing the entire lifecycle of event listeners (aiming to read sensor data from such Devices). These requests are classified into three main types, namely, requests for registering, updating, and removing event listeners, respectively. Registration requests are first intercepted by the Manager, who assigns a unique ID, and then forwards them to the corresponding Device they refer to (see Figure 2). When the Device receives a registration request, it creates a new topic with the MQTT Broker ('cuttlefish/event/{eventID}') and starts publishing the requested data. Finally, the Manager responds to the Client with the corresponding event ID. For update and removal requests, a similar process is followed. On update, event listener attributes (explained below) can be changed. On removal, the event listener on the Device (and all associated topics published to the MQTT Broker) are destroyed.

Cuttlefish supports three types of event listeners, namely, *Interrupt-Based*, *Comparison*, and *Frequency* listeners. The former applies to sensors that do not emit data continuously, but rather wait for an event to happen and then react to it in a particular way. An example would be a motion sensor, which emits data only when it detects motion. Comparison listeners generate events depending on the actual values emitted by a sensor, which are read periodically (e.g., every second). In other words, a comparison filter (e.g., greater, equal, or less than) is applied based on a specified threshold value. An example would be to request temperature data only when the temperature exceeds 30 degrees Celsius. Finally, Frequency listeners are used to request sensor readings at a given fre-

TABLE II
REQUESTS SUPPORTED VIA THE CLIENT API

Request	Scope	Filters / Types	Response
Find	Devices / Sensors / Actuators	ID, Name, Type, State, Thing	A list containing requested Devices/Things
Data	Sensor	Interrupt-based / Comparison / Frequency	The event ID created / updated / deleted
Action	Actuators	None	Success or Failure
Status	Sensor/Actuator	None	Current Sensor/Actuator state

quency (specified in seconds). All data are asynchronously passed to the application via a callback mechanism.

3) Action Requests: Action requests can only be directed to Devices with access to actuators. They are responsible for triggering actions on these Devices. The different actions that these requests may cause depend on the type of actuator targeted. For example, in the case of a smart lamp, the available action requests involve setting the brightness levels of the lamp to some value and switching their state from ON to OFF (and vice versa). For both types of actions, a callback is registered to notify the caller of the action's success or failure.

4) Status Requests: Status requests are used to poll the value/state of a sensor or actuator at a specific point in time. Devices receiving a status request will read, for example, the state of a sensor/actuator at that specific time instance and return its value in real-time. The response to such requests (via corresponding callbacks) comprises a list that contains pairs in the form $< value, attribute >$. For example, a status request on a smart lamp would return the state of the lamp (ON or OFF) and its current brightness level.

V. EVALUATION

Our preliminary evaluation focuses on Cuttlefish's resource utilization as a proxy of its lightweight nature. For the experiments, a web application was built on top of Cuttlefish, which involves two IoT devices, a custom device with sensors running on a Raspberry Pi 3, and a commercial device (Philips Hue [26]). The application and Philips Hue adaptor run on a laptop with a Quad-core 2.4GHz CPU and 12 GB of RAM. The Cuttlefish Manager runs on the Raspberry Pi 3 device.

A. Application Use Case

A home automation web application was built using the Cuttlefish Client API. The Devices used are the following:

Custom Raspberry Pi 3 Device. A low-powered device that uses System on Chip (with an integrated ARM-compatible CPU and GPU), which allows for directly attaching sensors and actuators to it. For the purposes of our experiments, an ultrasonic distance sensor and a DHT11 temperature/humidity sensor were used. This Device is programmed directly to interface with Cuttlefish through its Device API.

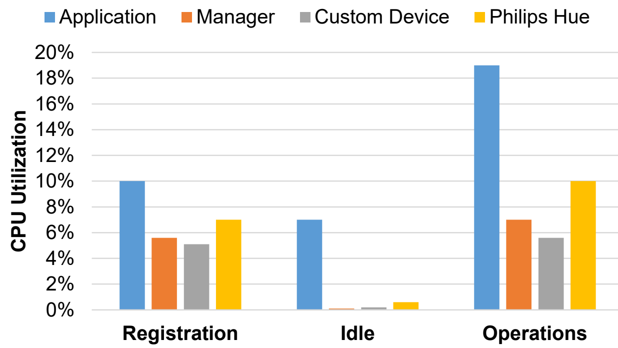
Philips Hue Device. A home automation platform that supports dimmable lamps, allowing a user to manipulate their brightness levels. HUE lamps are connected to a ZigBee network and are controlled from a bridge device. For this project, a kit of Philips Hue Bridge and two dimmable lamps were used. An API and SDK are available for accessing and interfacing with the Hue platform, which are based on a RESTful interface provided by the Hue Bridge. An adaptor/translator software component was built to map the functionality supported by the Devices API to that of the HUE SDK (recall Figure 1).

The web application developed includes a dashboard that lists all available Devices and their corresponding sensors/actuators at runtime. Selecting a particular Device has the effect of filtering the sensors and actuators that belong to it. The application also offers an option for filtering Devices by type. Furthermore, when a particular sensor/actuator is selected, information regarding its state appears on the dashboard, along with all registered events that concern it. As part of our use case, rules were implemented to showcase how automation can be accomplished through the Client API. More precisely, all different types of supported event listeners were created (Interrupt-based, Comparison, and Frequency event listeners) through which different events are raised. These events trigger specific actions such as changing the status of a lamp (from ON to OFF) or adjusting its brightness level.

B. Experimental Methodology

We study the resource utilization during three distinct periods of time: (i) *registration*, during which the two Devices first connect to the Manager and register themselves along with the connected sensors and actuators; (ii) *idle*, during which there is no exchange of data between the components; and (iii) *operations*, during which the application issues data requests continuously and reads sensor data from both Devices.

Figures 3(A) and (B) show the CPU and memory utilization, respectively, of the application, the Manager, and the two Devices during registration, idle time, and data operations. The application has the highest CPU and memory consumption as it is running within a Tomcat server and using the Spring framework, both of which can be resource intensive. The Manager and the two Devices experience very low CPU utilization, 5% – 10% during both registration and data operations. In the idle stage, the CPU usage is negligible ($<$0.5%) and accounts for the heartbeats sent from the Devices to the Manager to ensure their liveness. Memory usage is low and fairly stable across the three time periods, ranging between 81MB and 100MB for the Philips Hue and between 20MB and 40MB for the Manager and the Custom Device. Overall, these results showcase the low resource requirements of the Cuttlefish platform, its ability to run on a resource-constrained device, as well as its support for both Custom and Commercial Devices.



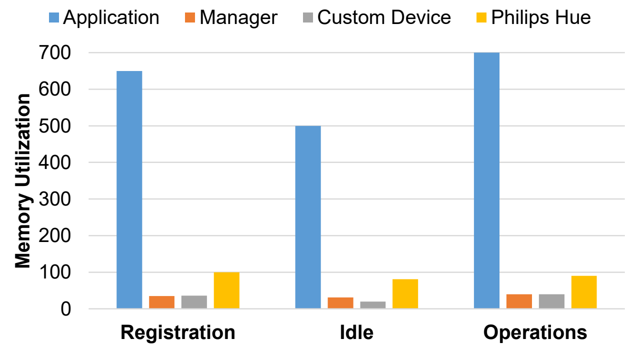


Fig. 3. (A) CPU and (B) memory utilization during registration process, idle time, and data operations.

To further investigate the impact of Cuttlefish to the application development process, we implemented the application with and without taking advantage of the Cuttlefish platform. The number of physical lines of code in the first case was only 1620 compared to 17988 in the latter, showing the significant benefits Cuttlefish can have in terms of implementation effort.

VI. Conclusions

Cuttlefish is a middleware that allows developers to combine multiple heterogeneous IoT devices with ease. It adopts a simple and efficient architecture that has all communications and data exchanges happen through a lightweight publish-subscribe engine. It supports: (i) a unified API for real-time access to heterogeneous IoT devices with minimal developer efforts; (ii) dynamic service discovery; (iii) device composition and local processing; and (iv) user security and privacy for dealing with sensitive data. We have shown experimentally that our prototype implementation is lightweight enough to allow it to be hosted on resource-constrained devices, along with the IoT applications that use it. By avoiding the use of cloud resources, the deployment of Cuttlefish-enabled applications is simpler and more cost effective, while any data collected remain under the control of the users throughout execution.

References

[1] A. Pamboris, P. Andreou, H. Herodotou, and G. Samaras, "MULTI-WEAR: A Multi-Wearable Platform for Enhancing Mobile Experiences," in *IEEE Consumer Communications & Networking Conference (CCNC)*, 2018.

[2] S. E. Alshaal, S. Michael, A. Pamboris, H. Herodotou, G. Samaras, and P. Andreou, "Enhancing Virtual Reality Systems with Smart Wearable Devices," in *IEEE Intl. Conf. on Mobile Data Management (MDM)*, 2016.

[3] A. Pamboris, P. Andreou, I. Polycarpou, and G. Samaras, "FogFS: A Fog File System For Hyper-Responsive Mobile Applications," in *IEEE Consumer Communications & Networking Conference (CCNC)*, 2019.

[4] A. Pamboris, M. Báguena, A. L. Wolf, P. Manzoni, and P. Pietzuch, "Demo:: NOMAD: An Edge Cloud Platform for Hyper-Responsive Mobile Apps," in *ACM Intl. Conf. on Mobile Systems, Applications, and Services (MobiSys)*, 2015.

[5] A. Pamboris and P. Pietzuch, "C-RAM: Breaking Mobile Device Memory Barriers Using the Cloud," *IEEE Transactions on Mobile Computing*, vol. 15, no. 11, pp. 2692–2705, 2015.

[6] A. Pamboris, G. Antoniou, C. Makris, P. Andreou, and G. Samaras, "AD-APT: Blurring the Boundary Between Mobile Advertising and User Satisfaction," in *ACM Intl. Conf. on Mobile Software Engineering and Systems (MOBILESoft)*, 2016.

[7] A. Pamboris and P. Pietzuch, "EdgeReduce: Eliminating Mobile Network Traffic Using Application-Specific Edge Proxies," in *ACM Intl. Conf. on Mobile Software Engineering and Systems (MobileSoft)*, 2015.

[8] R. van der Meulen. "Gartner Says 8.4 Billion Connected Things Will Be in Use in 2017, Up 31 Percent From 2016". [Online]. Available: http://www.gartner.com/newsroom/id/3598917

[9] A. H. Ngu, M. Gutierrez, V. Metsis, S. Nepal, and Q. Z. Sheng, "IoT Middleware: A Survey on Issues and Enabling Technologies," *IEEE Internet of Things Journal*, vol. 4, no. 1, pp. 1–20, 2016.

[10] M. A. Razzaque, M. Milojevic-Jevric, A. Palade, and S. Clarke, "Middleware for Internet of Things: A Survey," *IEEE Internet of Things Journal*, vol. 3, no. 1, pp. 70–95, 2015.

[11] M. Eisenhauer, P. Rosengren, and P. Antolin, "Hydra: A Development Platform for Integrating Wireless Devices and Sensors into Ambient Intelligence Systems," in *The Internet of Things*. Springer, 2010, pp. 367–373.

[12] J.-P. Calbimonte, S. Sarni, J. Eberle, and K. Aberer, "XGSN: An Open-source Semantic Sensing Middleware for the Web of Things," in *Proc. of the Intl. Workshop on Semantic Sensor Networks*, 2014, pp. 51–66.

[13] D. Massaguer, B. Hore, M. H. Diallo, S. Mehrotra, and N. Venkatasubramanian, "Middleware for Pervasive Spaces: Balancing Privacy and Utility," in *Intl. Conf. on Distributed Systems Platforms and Open Distributed Processing*. Springer, 2009, pp. 247–267.

[14] T. Teixeira, S. Hachem, V. Issarny, and N. Georgantas, "Service Oriented Middleware for the Internet of Things: A Perspective," in *European Conf. on a Service-Based Internet*. Springer, 2011, pp. 220–229.

[15] P. P. Ray, "A Survey of IoT Cloud Platforms," *Future Computing and Informatics Journal*, vol. 1, no. 1-2, pp. 35–46, 2016.

[16] N. Sinha, K. E. Pujitha, and J. S. R. Alex, "Xively Based Sensing and Monitoring System for IoT," in *Intl. Conf. on Computer Communication and Informatics (ICCCI)*. IEEE, 2015, pp. 1–6.

[17] H. Kim, E. Kang, E. A. Lee, and D. Broman, "A Toolkit for Construction of Authorization Service Infrastructure for the Internet of Things," in *Intl. Conf. on Internet-of-Things Design and Implementation (IoTDI)*, April 2017.

[18] Google Fit. [Online]. Available: https://developers.google.com/fit/

[19] IoT Analytics - ThingSpeak Internet of Things. [Online]. Available: https://thingspeak.com/

[20] P. Persson and O. Angelsmark, "Calvin–Merging Cloud and IoT," *Procedia Computer Science*, vol. 52, pp. 210–217, 2015.

[21] Node-RED: Flow-based Programming for the Internet of Things. [Online]. Available: https://nodered.org/

[22] MQTT. [Online]. Available: http://mqtt.org/

[23] Raspberry Pi - Teach, Learn, and Make with Raspberry Pi. [Online]. Available: https://www.raspberrypi.org/

[24] Arduino. [Online]. Available: https://www.arduino.cc/

[25] IoT: Intel Software. [Online]. Available: https://software.intel.com/en-us/iot/home

[26] Wireless and Smart Lighting by Philips: Meet Hue. [Online]. Available: https://www2.meethue.com/en-us

[27] Wemo - Home Automation. [Online]. Available: https://www.wemo.com/

[28] ecobee: Smart Home Technology. [Online]. Available: https://www.ecobee.com/