IJCRT.ORG ISSN : 2320-2882

INTERNATIONAL JOURNAL OF CREATIVE RESEARCH THOUGHTS (IJCRT)

An International Open Access, Peer-reviewed, Refereed Journal

A Trend toward Cloudlet for the Internet-of Things

Ph.D. Scholar Momin Mohammed Nadeem, Assistant Professor Dr. Gyanendra Kumar Gupta
IT&CS Department
Kalinga University Naya Raipur, Chhattisgarh, India

***Abstract*—The Internet of Things (IoT) is a key driver for smart city initiatives, making it necessary to have an IT infrastructure that can take advantage of the many benefits that IoT can provide. The Cloudlet is a new infrastructure model that offers cloud-computing capabilities at the edge of the mobile network. This environment is characterized by low latency and high bandwidth, constituting a novel ecosystem where network operators can open their network edge to third parties, allowing them to flexibly and rapidly deploy innovative applications and services towards mobile subscribers. In this paper, we present a cloudlet architecture that leverages edge computing to provide a platform for IoT devices on top of which many smart city applications can be deployed. We first provide an overview of existing challenges and requirements in IoT systems development. Next, we analyse existing cloudlet solutions. Finally, we present our cloudlet architecture for IoT, including design and a prototype solution. For our cloudlet prototype, we focused on a micro-scale emission model to calculate the CO2 emissions per individual trip of a vehicle, and implemented the functionality that allows us to read CO2 data from CO2 sensors. The location data is obtained from an Android smartphone and is processed in the cloudlet. Finally, we conclude with a performance evaluation.**

Keywords— Cloud Computing, Cloudlet Architecture, Server & Benefits, Internet of Things (IoT).

I. INTRODUCTION

Mobile devices have become ubiquitous and mobile communication is no longer exclusive to smartphones and tablets; almost any portable device can now be outfitted with electronics, software, network connectivity, and sensors, allowing it to provide pervasive services, i.e. communicate and share data with other devices. These new capabilities constitute one of the most innovative and disruptive technological scenarios of today, the Internet of Things (IoT).

The Internet of Things is becoming more and more widespread, and its potential for improving our overall quality of life is exceptional. Unfortunately, it inherits all of the many limitations that are intrinsic to mobility, regardless of the technology that is being used. Reduced computational resources, limited storage, and low battery life are, for example, particularly critical for resource-rich applications [1]. One very popular contemporary solution towards overcoming these impediments is to rely on cloud computing services. Finding an efficient strategy to leverage the cloud in mobile devices has been a recurrent topic across various research efforts, including the design of lightweight runtime environments, application partitioning, and application offloading, and bringing the cloud closer to mobile devices [2].

1. Problem statement

The specific problems that we are addressing are scalability, data management, and resiliency. First, IoT systems deal with heterogeneous data that come from an increasing number of devices, making scalability a fundamental requirement. Existing cloudlet solutions present certain issues regarding scalability. They make use of point-to-point communication protocols like HTTP, which has no built-in distribution features. Therefore, developers would have to create their own distribution mechanisms, making the process of sending and receiving sensor data more complex. Moreover, existing cloudlet architectures are intended to serve a small number of devices that run interactive applications. They use VMs as form of virtualization and they often have to run demanding algorithms used to power up technologies like augmented reality and speech recognition. In the case of a Cloudlet for IoT, a faster and lighter form of deployment that is able to serve a considerably bigger number of devices is required.

Second, IoT sensors generate massive amounts of data that has to be processed and stored somewhere. Unfortunately, existing cloudlet solutions are stateless, therefore all the data would still have to be send to the cloud after processing, which can saturate the network with traffic coming from IoT devices. Finally, applications that rely on data produced from real time environment monitoring using IoT sensors, temporary outages or increased response times cannot be accepted. Available solutions

use centralized cloud services that are off premises and out of the user control. This complete dependence on the cloud can deteriorate the response time or even make the system more susceptible to failures and attacks that intend to interrupt or suspend the services.

The use case of the project consists in the development of a cloudlet infrastructure that provides computing resources for smartphones and an IoT sensor system (Figure 1).

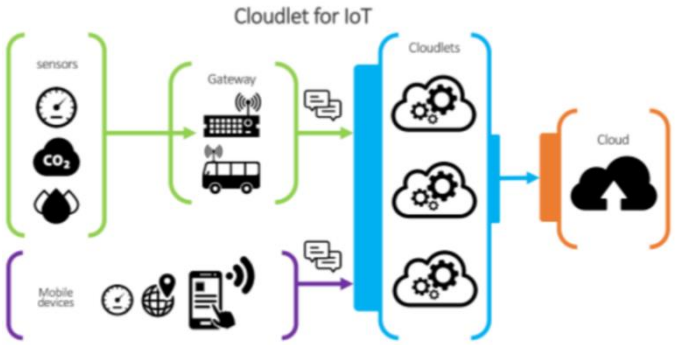

Fig.1 Cloudlet for IoT

The proposed architecture addresses the previously described challenges as follows:

Scalable communication and deployment. The cloudlet deployment is done using containers since they use less resources in comparison with the VMbased cloudlets that we analysed. A publish-subscribe messaging pattern using MQTT is also implemented. With this approach, the cloudlet can consume sensor data in parallel. Moreover, the MQTT protocol is design to work even in unreliable networks.

Data management support. The proposed architecture performs the data management directly on the edge, including both processing and caching. This approach reduces the system throughput to avoid saturating the internet bandwidth with traffic coming from IoT devices, since only historic data already processed and aggregated is stored in the cloud.

System Resiliency. we provide a decentralized system to process data on the edge, which is able to reduce end-to-end user perceived latency and increase the system resiliency1 by providing data redundancy, high availability, and a loosely coupled architecture.

The development of such a system leads to the following question:

What are the main requirements for a cloudlet to support the use case, and which strategies can we use to improve scalability and reliability, and reduce throughput?

2. Performance evaluation

The performance evaluation is based on measuring the response time of the system, from the moment the device agent is created and the client data is sent and processed, until the device agent is removed.

We start by creating a testing workbench – a fixed development environment that is reproducible and portable. This environment allows us to measure the performance of the cloudlet. One measuring cycle consists of measuring the response time given a pre-determined number of clients (sensor and smartphone clients) sending data We start off the measuring process with a ten-client cycle, and then gradually increase the number of clients for each further cycle. At the end of each cycle, we collect the desired measurements. After having performed all of the cycles, we will analyse the obtained data and the performance of our implemented cloudlet prototype.

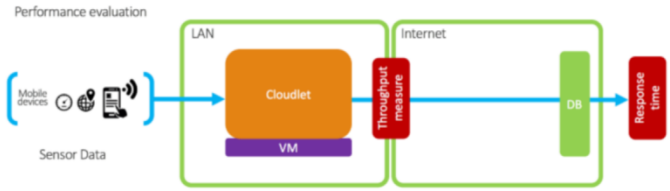

Fig.2 Performance Evaluation

3. Delimitations

The prototype is focused on a cloudlet that provides a data processing system where sensor data can be cached, processed, and aggregated before being ready to be stored in a cloud storage service (Figure 3). Given the time frame of this thesis project, integration with cloud services, inter-cloudlet communication, and multiple cloudlet integration have not been addressed. Additionally, the sensors involved are assumed to have sufficient communication capabilities to function without a gateway.

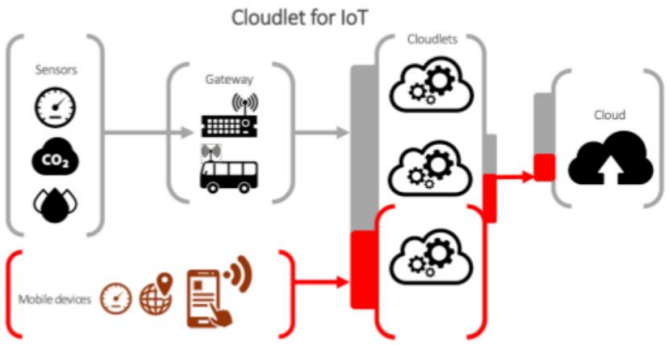

Fig.3 Prototype Delimitation

The client is a smartphone that sends sensor data to the cloudlet. The proposed cloudlet architecture includes a message broker that implements a publish/subscribe messaging model that facilitates one-to-many communication (Figure 4). This enables the development of loosely coupled applications, where components can be modified and replaced with alternative implementations

without interfering with other elements of the system. At the same time, the client applications (IoT devices) are less constrained to be same developed on the same programming language, operating system, or any environment.

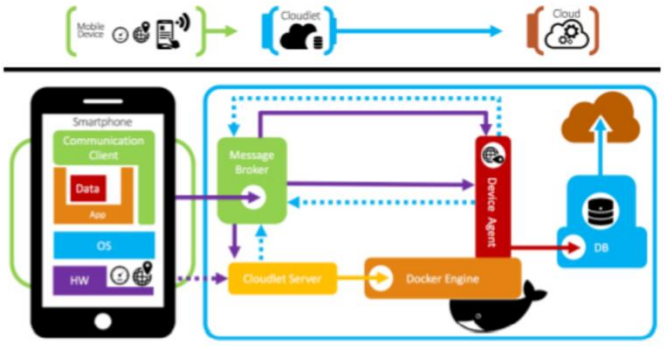

Fig.4 Prototype Design

II. Cloudlet benefits

A cloudlet is a new architectural element that represents the middle layer of a 3-tier hierarchy (Figure 5). The original motivation for this architecture is to reduce the latency of MCC applications by using a singlehop network and also potentially lower battery consumption by avoiding the use of broadband wireless networks, which normally consume more energy. The characteristics of cloudlets bring additional benefits [8]:

Easy deployment. The fact that cloudlet servers are stateless simplifies management; adding or replacing a cloudlet only requires a moderate setup and configuration effort.

Security improvement. The proximity of the cloudlet to mobile devices can make the architecture more resilient against DoS attacks2. It can also reduce information leakage due to traffic analysis, since restricting the range of end-toend communication prevents distant snoopers from accessing traffic information.

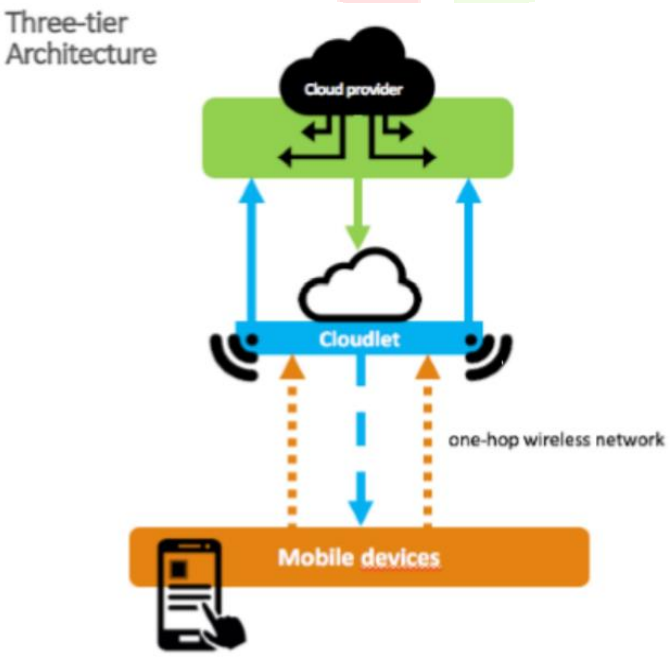

Fig.5 Cloudlet Hierarchy

Resilience. A cloudlet collection can offer reliable cloud computing services even with a fragile connectivity to a distant cloud provider.

III. Cloudlet architectures

During the past few years, several cloudlet architectures have been proposed, many of which are based in VMs deployed in elastic cloud computing platforms, such as OpenStack. There exist other designs that distinguish between elastic cloudlets and ad-hoc cloudlets with fixed resources, and between centralized and decentralized cloudlet management.

1. VM-based cloudlet architecture

Satyanarayanan et al. [1] [8] [9] [10] have developed a cloudlet reference architecture based in VM overlay application offloading. The cloudlet system consists in two type of elements, the cloudlet host and the mobile client (Figure 6). Mobile devices can offload computations to a single mirrored device clone in the form of a VM or to a set of VM that handle specific computations.

The cloudlet host manages a discovery service that broadcasts the cloudlet IP address and port, a storage system for the base VM images, a cloudlet server that handles application overlays for code offloading and the life cycle of the VMs, and it also contains a VM manager that hosts all guest VM instances corresponding to each mobile app.

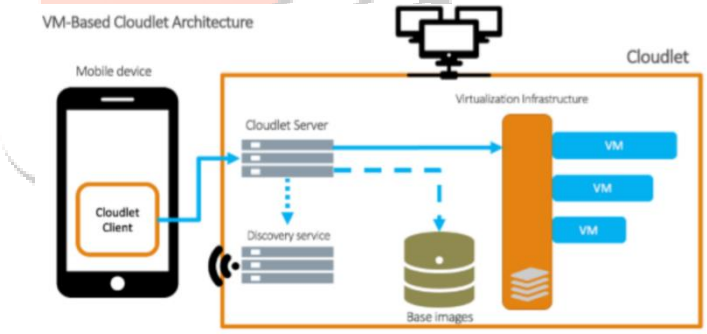

Fig.6 VM-based Cloudlet Architecture

2. Ad hoc cloudlet vs elastic cloudlet

Instead of managing VMs for the deployment of a cloudlet system, Verbelen et al. [11] propose a finer-grained cloudlet concept that offloads applications on the component level, without the need of sending a VM overlay. They also suggest that Cloudlets can be formed dynamically with any device in the LAN network that has available computing resources. Their architecture consists of three layers (Figure 7): the cloudlet level, the node level, and the component level.

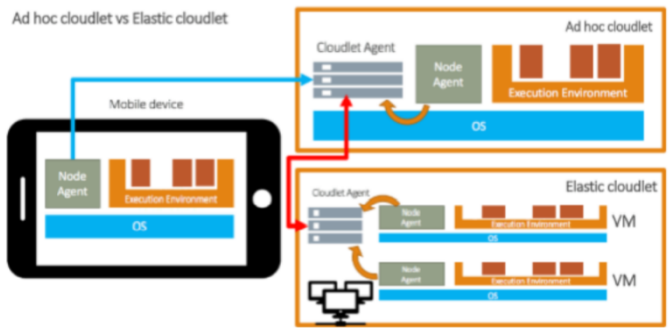

Fig.7 Dynamic Cloudlet

A component is the unit of deployment managed by an Execution Environment (EE). To support distributed execution, the components on different EEs can communicate using remote procedure calls (RPCs). The EE and the OS together form a node that is managed by a Node Agent (NA). The cloudlet is managed by a Cloudlet Agent (CA), that communicates with all underlying Node Agents. CA of different cloudlets can also communicate with each other. Abolfazli et al [13] also propose a dynamic cloudlet architecture consisting only of ad hoc cloudlet nodes, all of which are administered by a central service governor, a replicated supervisory entity that monitors and supervises computing augmentation entities (Figure 8).

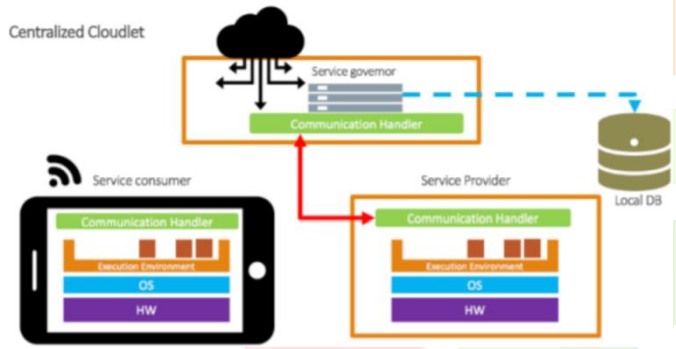

Fig.8 Centralized Cloudlet

IV. Cloudlet for the IoT

IoT applications in general, and sensor and crowd-sensing based applications in particular, have to be able to deal with a large amount of highly dynamic data coming from the real world. To support this scenario, it is necessary to combine computational resources at the edge with cloud computing services.

Cloudlets (As described by Satyanarayanan et. al [37]) were originally ideated to reduce end-to-end latency in interactive applications while addressing other concerns, such as limited processing capability and limited battery capacity of mobile devices. One example of such interactive systems is Gabriel [38], a wearable cognitive assistant for users in cognitive decline. It combines the image capture and sensing capabilities of Google Glass devices with cloudlet processing to perform real-time scene interpretation. This system is layered on top of an OpenStack extension for use in cloudlet environments.

As we can see in Figure 9, this system includes an ensemble of VMs, each encapsulating a different cognitive task. A single control VM is responsible for all interactions with the mobile device, and a Pub/Sub mechanism distributes sensor streams to cognitive VMs. This kind of architecture is intended to be used in an environment were only a handful of devices are expected to be operating at the same time.

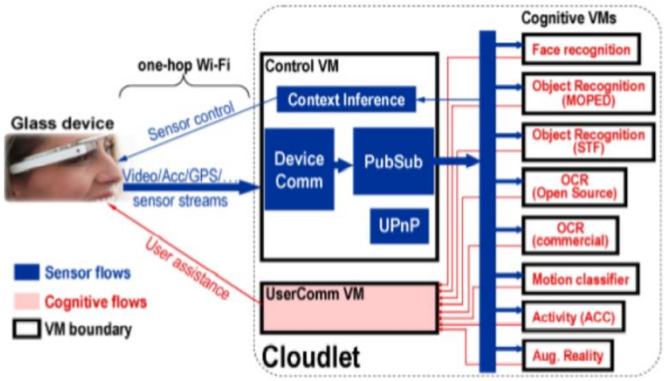

Fig.9 Cognitive Assistant Architecture

A VM-Based cloudlet architecture for offloading computations faces several drawbacks if it is intended to be used for IoT. First, higher synchronization efforts are necessary to preserve a consistent state between the mobile device and the VMs that handle the computations that are being offloaded [23]. Second, as the number of devices that leverage the cloudlet increases (as in the case of sensor networks) problems related to scalability become evident, and the deployment of numerous heavyweight virtual machine images grows unpractical.

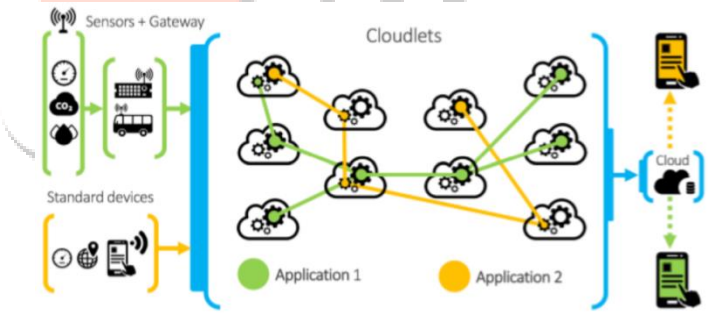

Fig.10 Cloudlet Applications for IoT

1. Cloudlet design and architecture overview

We assume that various devices such as small servers, personal computers, or onpremises small data centers, are available to run cloudlet instances. These devices are physically spread throughout a geographic area and are close to the network infrastructure (Figure 11).

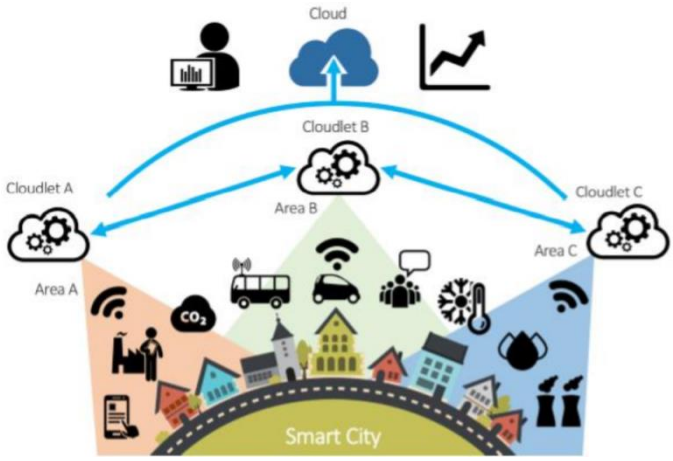

Fig.11 Geographical Distribution of Cloudlets

The cloudlet architecture that we propose consists of four main components (Figure 12): the Cloudlet server, Device Agents, Message Broker, and Client Application. The interaction between a mobile device and the cloudlet starts when the client application requests the deployment of a device agent to the cloudlet server. Each agent serves an individual device and performs specific tasks such as collecting and processing sensor data. The device agent and the client application can communicate using the message broker. A distributed application in our cloudlet model consists of a set of device agents distributed across one or various cloudlets.

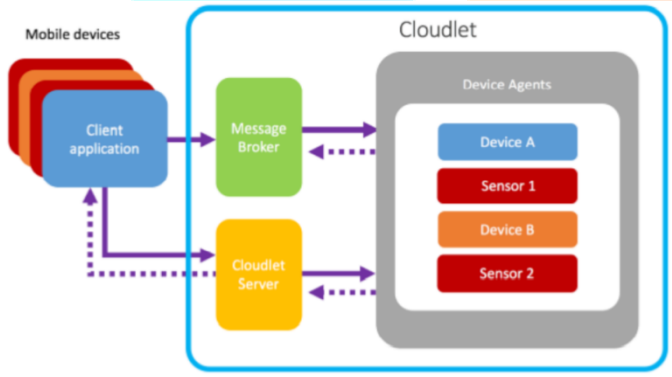

Fig.12 Cloudlet for IoT Main Components

1.1 Cloudlet communication

Our cloudlet architecture uses a combination of two approaches to communicate, synchronous HTTP requests and asynchronous messaging. Representational State Transfer (REST) is used to enable the communication between the client application and the cloudlet server to manage the lifecycle of the device agents. REST uses a request/response messaging model; if the desired response is not available, the application needs to execute the call again. Generally speaking, REST is a powerful messaging exchange pattern, but the clients always need to know the host and port of the server to which they want to communicate.

MQTT is a lightweight protocol that implements a publish/subscribe architecture, minimizing network bandwidth and device resource requirements. It is open-source and a single MQTT server is capable of supporting thousands of remote clients. These characteristics are particularly well suited for an IoT environment [39]. furthermore, there exists several implementations and client libraries on the majority of programming languages, Thereby, MQTT was chosen as the communication protocol between clients and device agents. The MQTT protocol includes the following benefits [39]:

- It can easily be adapted to a wide variety of operating systems, devices, and platforms.
- It is very well suited for constrained networks that deal with high latency, low bandwidth, and fragile connections.
- It is designed specifically for devices with little memory or processing power.
- It enables massive scalability of deployment.
- It implements a publish/subscribe messaging model that facilitates one-to-many distribution
- The sender applications or devices do not need to know anything about the receiver, not even their address.
- It delivers relevant data to any component that can use it.

In order to have a better understanding of the differences between the two protocols that we are using, we present a comparison between MQTT and HTTP in Table 1.

	MQTT	*HTTP*
Design orientation	Data-centric	Document-centric
Pattern	Publish/subscribe	Request/response
Complexity	Simple	More complex
Message size	Small	Large
Service levels	Three quality of service settings	All messages get the same level of service
Extra libraries	Less than 100 KB	Typically not small
Data distribution	Supports 1 to zero, 1 to 1, and 1 to n.	1 to 1 only

Table 1. MQTT vs. http

1.2 Cloudlet: VM vs Containers

The use of an intermediate software layer to provide virtual resources on top of an underlying system is known as resource virtualization. In general, the virtualized resources can be seen as isolated execution contexts, most commonly virtual machines (VM). Instead of using VMs as a method of virtualization, we opted for a container-based virtualization alternative for our cloudlet development (Figure 13).

In the case of hypervisor-based virtualization, each VM has its own operating system, allowing a single host to execute multiple operating systems. Alternatively, in container-based virtualization all virtual instances share a single operating system kernel. However, from the point of view of the users, each container looks and executes like a stand-alone OS. This makes containers a lightweight virtualization option [40]. Furthermore, the use of Linux containers results in equal or better performance than VMs when evaluating benchmarks that stress different aspects such as computation, memory bandwidth, memory latency, network bandwidth, and I/O bandwidth [41].

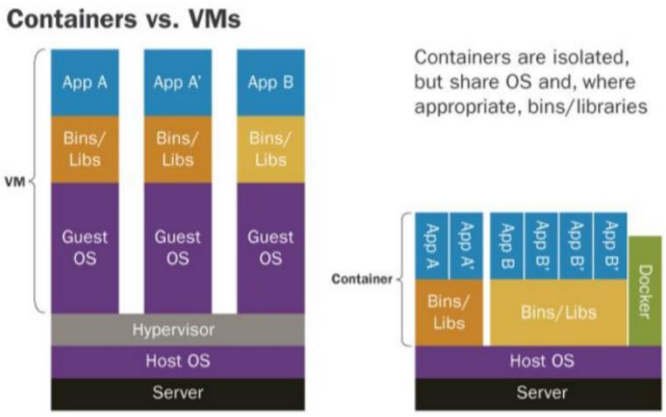

Fig.13 Containers vs. VMs

Docker is a widely adopted platform for deploying containerized applications. Linux Containers and LXC (a user-space control package for Linux Containers) constitute the core of Docker. LXC ensures that the container's root user does not have root privileges on the host. Furthermore, Docker can limit the resources being consumed by a container, such as memory, disk space and I/O. In addition, Docker uses a file system for containers. that allows Docker to use a single image as the basis for many different containers, which results in savings of storage and memory, as well as their faster deployment. All this characteristics, including the fact that is open-source, motivated our decision to implement our architecture using Docker.

2. Cloudlet prototype development

The transportation sector is a one of the main sources of CO2 emissions, and a major contributor to environmental pollution [42] [43]. In order to have an efficient strategy to reduce pollution produced by vehicles, it is necessary to have a model that can estimate the CO2 emissions accurately. The use case of our cloudlet prototype consists of the development of a distributed application to calculate CO2 emissions from vehicles using mobile devices, and also monitor and process data from CO2 sensors.

The most common models used for calculating greenhouse gas emissions like CO2 are known as macro-scale models. In these models, the estimation is done through area-wide driving cycles. For our cloudlet prototype, we focused on a micro-scale emission model to calculate the CO2 emissions per individual trip of a vehicle. The location data is obtained from an Android smartphone and is processed in the cloudlet. We also added a functionality that allows us to read CO2 data from CO2 sensors (Figure 14).

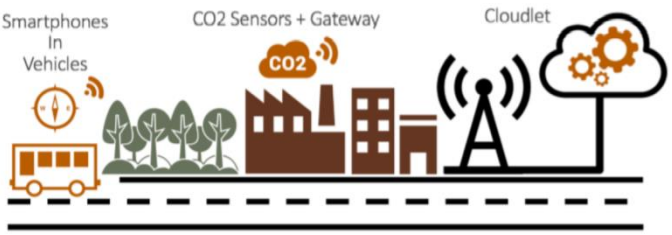

Fig.14 Use Case: CO2 Monitoring

3. Client application

We have developed two kinds of client applications, one for Android smartphones and one for CO2 sensors. Both clients make use of the cloudlet server REST API (Figure 15) to request computing resources to the cloudlet in the form of docker containers (device agents). Once the container is ready, the mobile client starts publishing data via a message broker that implements the MQTT protocol (Figure 16). The client application needs to send a POST request a specific cloudlet URI that handles the desired type of device to create a device agent.

POST http://<cloudletserver>:<port>/<device type>/ JSON

POST http//localhost:3000/smartphone/ {deviceId: "0001", type: "smartphone"}

Similarly, the PUT method is used to stop and start the container. Additionally, once the client application has completed the data transfer, the device agent can be removed by sending a DELETE request.

PUT/DELETE http://<cloudletserver>:<port>/<device type>/<device ID>/ JSON

PUT/DELETE http//localhost:3000/smartphone/0001/ {deviceId: "0001", stop: true}

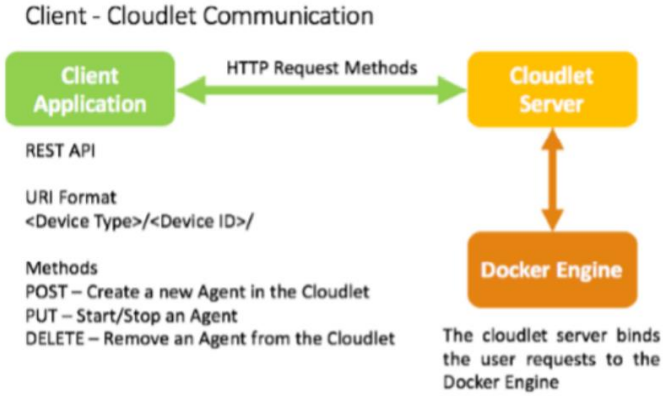

Fig.15 Client-Cloudlet Communication

The communication between client applications and device agents is done via MQTT message topics. There are four different topics that are used: one for the data that is being send from the client, one to send a notification to the agent to save the data, one to send messages on behalf of the client in case that it has being disconnected, and one to publish the results of the processing that is being done by the device agent.

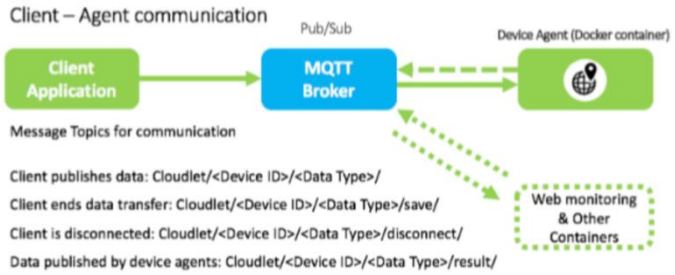

Fig.16 Client-Device Agent Communication

3.1 Smartphone client application

The smartphone client application was designed for Android devices. The first step in using the application consists of typing the address of the cloudlet server and requesting the creation of a device agent for the smartphone (Figure 17a). After the agent is created, we will see the option to start the GPS data transfer (Figure 17b). Finally, the device agent starts the calculation of CO2 production as a function of speed and acceleration of the vehicle (Figure 17c). For testing purposes, a speed simulator is also included. The application was developed using Android Studio; for the asynchronous message communication, we used a client implementation of MQTT provided by the Eclipse Paho3 project which consists of various open-source MQTT clients for many programming languages.

Fig.17 Smartphone Client

The device used to test the client was a Samsung Galaxy S5 Mini, Table 1 presents the main characteristics of this device4.

Galaxy S5 mini specifications	
Processor	CPU: Quad-Core 1.4GHz
Display	Size: 4.5" (113.4mm) 720 x 1280 (HD)
Camera	Main Camera: 8.0 MP, Front Camera: 2.1 MP
Memory	RAM Size: 1.5 GB, ROM Size: 16 GB
Network/Bearer	3G, 4G
Connectivity	GPS, Glonass, Wi-Fi 802.11 a/b/g/n 2.4+5GHz, Bluetooth v4.0
Sensors	Accelerometer, Fingerprint Sensor, Gyro Sensor, Geomagnetic Sensor, Light Sensor, Proximity Sensor
Physical specification	Dimension (mm) 131.1 x 64.8 x 9.1, Weight (g) 120

Table 2. Smartphone Specifications

3.2 Sensor client application

The sensor client application is responsible for the publication of the sensor data in the cloudlet. It consists of a Java application that includes a MQTT client that sends the data from the CO2 sensor to the message broker, and a HTTP client to handle the device agent.

Figure 18 shows the user interface of the application. First, we need to input the details of the port where the sensor is connected and the address were the MQTT server and the Cloudlet server are hosted (Figure 18 a & b). Afterwards, we can start and stop the readings from the CO2 sensor (Figure 18 c & d). Finally, we start the MQTT client to initiate communication with the device agent (Figure 18 e & f).

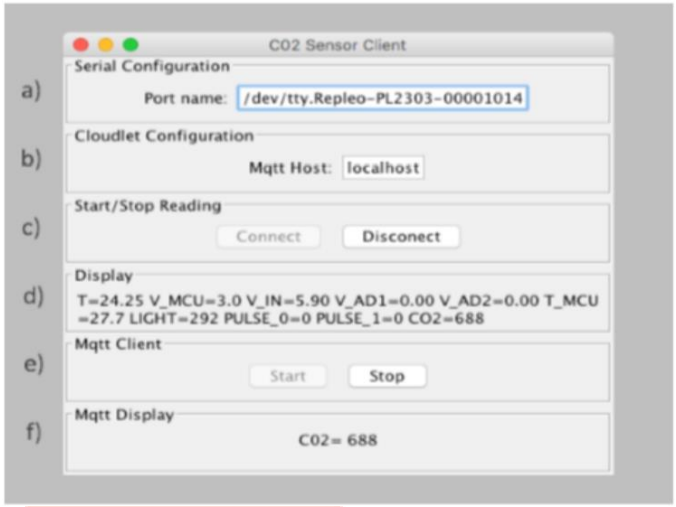

Fig.18 CO2 Sensor Client

The sensor used for the prototype was a CO2 Engine K30 sensor produced by SenseAir, with Table 3 showing its main specifications5.

CO2 Engine K30 sensor specifications	
Operating Principle	Non-dispersive infrared
Measured gas	Carbon dioxide (CO2)
Measurement range CO2	0 to 5000 ppm / 0 to 3%vol
Accuracy	±30 ppm ±3% of reading
Dimensions	57 mm x 51 mm x 14 mm
Maintenance	Maintenance-free*
Life Expectancy	> 15 years
Operation temperature range	0 to 50 °C
Operation humidity range	0 to 95%
Power supply	4.5 to 14.0 V DC
Response time(T1/e)	20 sec diffusion time
Warm up time	1 min
Communication	Uart (Modbus)

Table 3. CO2 Sensor Specifications

V. Cloudlet Server

The cloudlet server is the main element of the architecture. It implements a REST API to handle requests from client applications, binds all the requests related to the device agent lifecycle to the Docker engine, and provides a user interface to monitor all connected devices in real time.

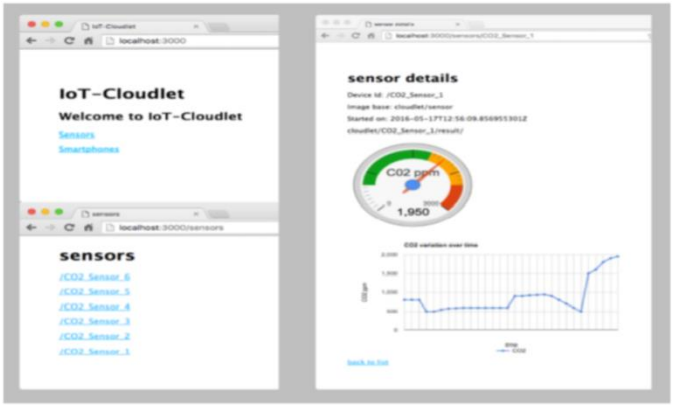

Fig.19 Cloudlet Server UI

The cloudlet server was implemented using Node.js, an open source back-end JavaScript environment that supports long-running server processes and is focused on low memory consumption. Node.js is based on Google's V8, a JavaScript runtime implementation. Unlike server-side environments like Java, that rely on multithreading to support the concurrent execution of business logic, Node implements a non-blocking asynchronous I/O event-driven model (Figure 20).

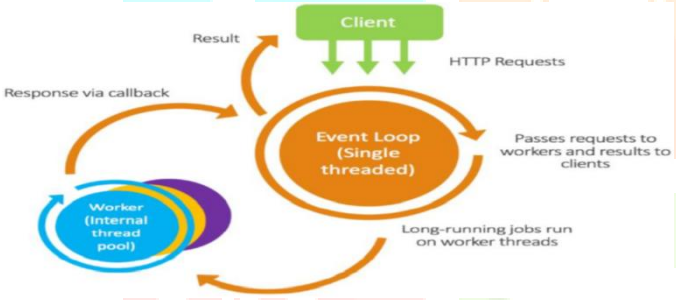

Fig.20 Node.js Process Model

1. Message Broker

The Message Broker is an intermediary module that mediates the communication between different entities within the cloudlet architecture. Its main purpose is to implement decoupling, i.e. to make sure that applications are mutually aware of each other only to the minimal extent necessary for them to be able to exchange messages.

As our message broker, we decided to use Mosquitto6, a lightweight MQTT server implementation. In order to run on machines of low capacity, Mosquitto is written in C. The current implementation consumes around 3MB RAM with 1000 clients and it has been successfully tested with up to 100,000 connected clients7. In order to offer scalability, Mosquitto has bridge capabilities that allow it to connect to other MQTT servers and not only Mosquitto instances, which, in turn, allows for the constructions of MQTT server networks, passing MQTT messages from any location in the network to any other. It also provides Websocket support so that one can implement MQTT clients directly on top of the web browser.

2. Device Agents

The data produced by the mobile devices and sensors is processed by the device agents. They are subscribed to specific topics via the Mosquitto message broker and are intended to be very flexible.

There are two different types of device agents: smartphone agents and sensor agents. Both types of agents are based on the Iron microcontainer image for Node.js, instead of using the official Docker Image repository for Node.js. This is mainly because the size of the official Node image is ~600 MB, which is exceptionally demanding, considering the amount of agents that we need to deploy.

Iron microcontainers are much smaller, they only include the application itself and the OS libraries and language dependencies required to run it. In the case of the Iron Node image, the size is ~20 MB. The main benefits8 of using microcontainers are: easier distribution (since the image is much smaller, different cloudlets can be distributed much quicker) and improved security (since less code is allocated in the container, we have a smaller attack surface).

The sensor agent collects CO2 data from a sensor and when the client sends a save request, it creates a JSON object that includes the mean value of the CO2 readings, the device Id, and a timestamp. This object can be saved on a database or published on a different topic to be used by another agent. The smartphone agent collects the location data and speed from the mobile device and uses it to calculate the CO2 production (grams per second) as a function of speed and acceleration [49]. When the agent receives a save request, it returns a JSON object containing the total CO2 produced during the whole trip, including the duration of the trip and the coordinates of the start and end.

Fig.21 Device Agents Output

3. Performance evaluation

The performance evaluation is based on measuring the response time of the system, from the moment the device agent is created and the client data is sent and processed, until the device agent is removed. We start by creating a testing workbench, which is a fixed development environment that is reproducible and portable. This environment allows us to measure the performance on the cloudlet. One measuring cycle consists of measuring the response

time given a predetermined number of sensor and smartphone clients that are sending data. We start off the measuring process with a N-client cycle, and then gradually increase the number of clients for each further cycle. The cloudlet prototype and the testing workbench were performed using an Apple MacBook Pro, whose hardware and software specifications are provided in Table 4.

MacBook Pro	
Processor	Intel Core i5 2.6 GHz
Memory	8 GB
SSD	120 GB
System Version	OS X 10.11.5 (15F34)
Kernel Version	Darwin 15.5.0

Table 4. Computer Hardware

4. Testing workbench

Because the Docker daemon uses Linux-specific features, is not possible to run Docker natively in OS X or Windows machines. Therefore, is necessary to use a Linux VM that can host Docker. In a Linux installation, the Docker daemon, Docker client, and any container run directly on the physical machine. In the case of our OS X installation, we used Docker Machine to install and run Docker inside a lightweight Linux VM made specifically for the running of the Docker daemon on Mac OS X (Table 5).

Linux VM for Docker on OS X	
Cores	1
Memory	2 GB
Disk size	20 GB
Docker version	1.10.3, build 20f81dd

Table 5. VM Specifications

In OS X, the Docker host address is the address of the Linux VM. When we start a device agent (container), the ports map to ports on the VM, which means that the user can address ports on a device agent using the VM addressing.

The Docker Machine is currently the only way to run Docker on Mac OS X or Windows. It can also be used for provisioning and management of multiple remote Docker hosts, as well as provision of Swarm clusters.

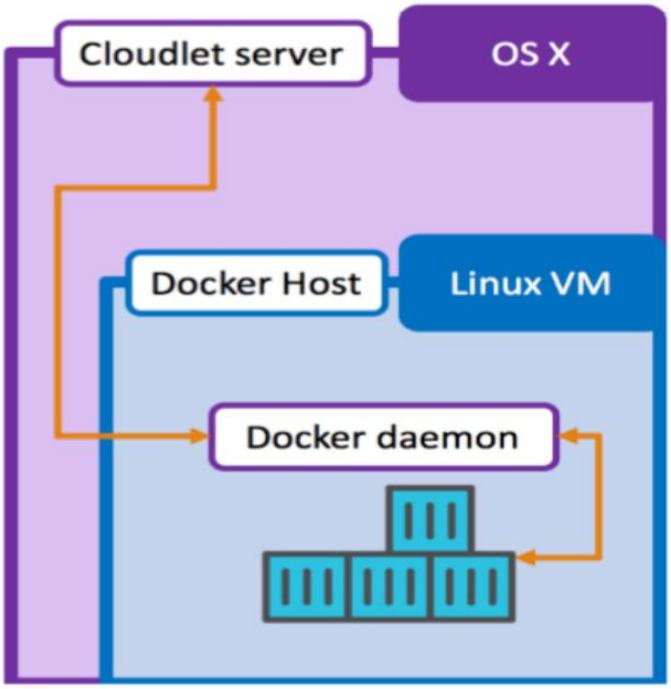

Fig.22 Docker OS X Installation

5. Device Agent Simulators

The device agent simulator is a Java-based application that uses threads to simulate multiple mobile devices that send http requests to create, stop, and remove device agents. When we type in the number of devices and press create, each of these devices will send a create request at a random time that ranges from zero to ten seconds. Then we can choose to either start sending data or to stop and remove the agent. If we decide to start a data simulation, we can introduce the duration in seconds. In the case of the sensor simulator, each simulated device sends a set of messages containing a JSON object with a randomly generated CO2 sensor reading. The smartphone simulator sends instead a speed value and a set of coordinates.

Fig.23 Device Agent Simulator

6. Hard disk and memory usage

Figure 24 presents a comparison of the size between different Docker images. The largest images that we compared were the official Node.js image and the official Java language image, both of which are over 500 MB in size.Our device agent images for both sensor and smartphone devices, which include all the required files and software ready to be used, fits in only 34.7 MB of space. With respect to memory usage, each agent requires ~18 MB of RAM and ~0.30% of CPU utilization to run and process the data from the mobile client.

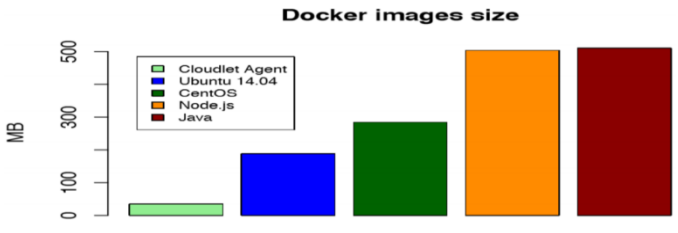

Fig.24 Docker Images Comparison

In the case of smartphone agents, we can see that after increasing the number of devices above 125, the disk usage is increasing at a lower rate, which is due to the fact that many of the smartphone agents that were deployed started to fail because of the lack of available resources in the VM and, therefore, they were unable to process and cache any data.

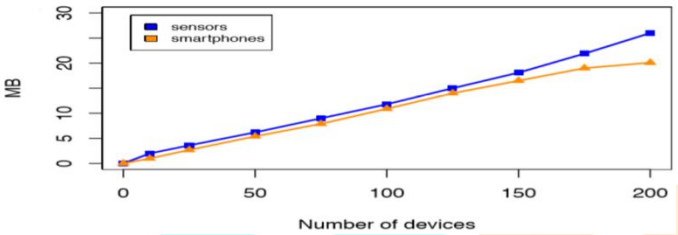

Fig.25 Hard Disk Usage

Figure 26 presents the memory usage evaluation. After exceeding 125 devices, the memory usage remained to be around 1.7 GB, which was the real maximum amount that could be used, since the remaining 0.3 GB is used by other activities of the operating system.

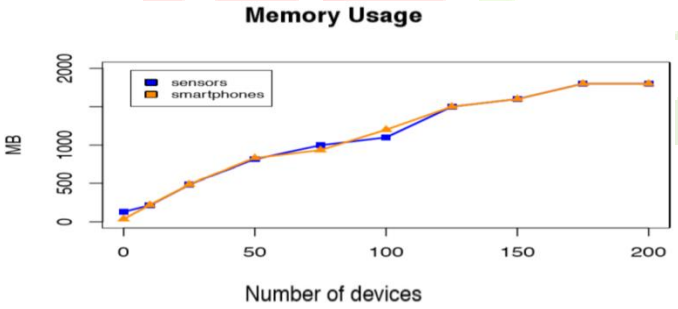

Fig.26 Memory Usage

7. Cloudlet response time

The Cloudlet response time evaluation consisted of a set of REST requests to deploy, stop, and remove a number of device agents during a fixed period of time. For this purpose, we measured the time between the client requests coming into the Cloudlet server and when the response headers are written, in milliseconds. Figure 27 and Figure 28 show the response time from Post and Put requests respectively. We observe that in both cases, after having more than 75 devices sending requests in parallel the response time increases dramatically.

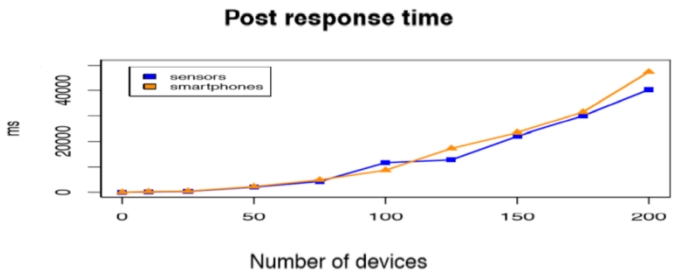

Fig.27 Post Response Time

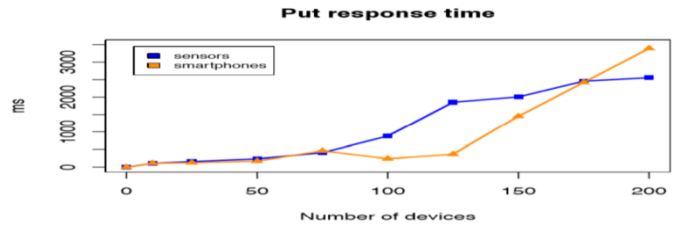

Fig.28 Put Response Time

On the contrary, the response time from Delete requests remained between 10 to 15 milliseconds, regardless of the number of parallel requests that were sent.

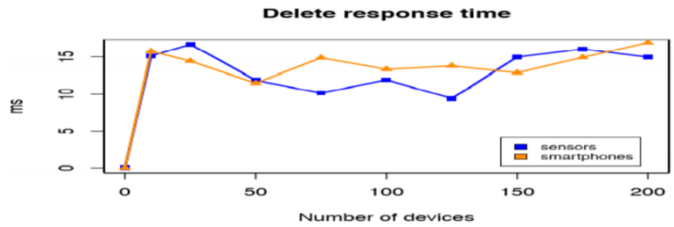

Fig.29 Delete Response Time

VI. Conclusions

We are still in an early stage of the development of mobile edge computing solutions, and there are still numerous efforts focused on the standardization of these technologies. This is of paramount importance, as otherwise there will exist multiple and not necessarily compatible solutions; this can lead to a fragmented marketplace that would fail to grow. It is also important that key industry players, such as network providers understand what are the benefits of opening their infrastructure to allow third parties to provide new services at the edge. In the specific case of our cloudlet architecture, its ability of operating on any commodity hardware is a very attractive feature, allowing mobile developers to deploy highly scalable applications in a secure and highly customizable environment. Finally, the user case itself is a very interesting tool that can be helpful to understand how the driving styles can affect the CO2 emissions from vehicles.

REFERENCES:

[1] M. Satyanarayanan, P. Bahl, R. Caceres and N. Davies, "The case for VMbased cloudlets in mobile computing," Pervasive Computing, vol. 8, pp. 1423, 2009.

[2] E. Ahmed, A. Gani, M. Sookhak, S. Hafizah Ab Hamid and F. Xia, "Application optimization in mobile cloud computing: Motivation, taxonomies, and open challenges," Journal of Network and Computer Applications, vol. 52, pp. 52-68, June 2015.

[3] F. Bonomi, R. Milito, J. Zhu and S. Addepalli, "Fog computing and its role in the internet of things," in Proceedings of the first edition of the MCC workshop on Mobile cloud computing, 2012.

[4] V. Gaziz, M. Goertz, M. Huber, A. Leonardi, K. Mathioudakis, A. Wiesmaier and F. Zeiger, "Short Paper: IoT: Challenges, Projects, Architectures," in Intelligence in Next Generation Networks (ICIN), 2015 18th International Conference on, 2015.

[5] I. Lee and K. Lee, "The Internet of Things (IoT): Applications, investments, and challenges for enterprises," in Business Horizons. July 2015, 2015.

[6] E. Borgia, "The Internet of Things vision: Key features, applications and open issues," in Computer Communications, 2014.

[7] B. Zhang, N. Mor, J. Kolb, D. S. Chan, N. Goyal, K. Lutz, E. Allman, J. Wawrzynek, E. Lee and J. Kubiatowicz, "The cloud is not enough: saving IoT from the cloud," in Proceedings of the 7th USENIX Workshop on Hot Topics in Cloud Computing, 2015.

[8] M. Satyanarayanan, G. Lewis, E. Morris, S. Simanta, J. Boleng and K. Ha, "The role of cloudlets in hostile environments," Pervasive Computing, vol. 4, pp. 40-49, 2013.

[9] S. Simanta, G. Lewis, E. Morris, K. Ha and M. Satyanarayanan, "Cloud computing at the tactical edge," 2012.

[10] S. Simanta, K. Ha, G. Lewis, E. Morris and M. Satyanarayanan, "A reference architecture for mobile code offload in hostile environment," Mobile Computing, Applications, and Services, pp. 274-293, 2013.

[11] T. Verbelen, P. Simoens, F. De Turck and B. Dhoedt, "Cloudlets: bringing the cloud to the mobile user," in Proceedings of the third ACM workshop on Mobile cloud computing and services, 2012.

[12] M. Shiraz and A. Gani, "Mobile cloud computing: critical analysis of application deployment in virtual machines," in International Proceedings of Computer Science & Information Tech, 2012.

[13] S. Abolfazli, Z. Sanaei, A. Gani, F. Xia and W.-M. Lin, "RMCC: Restful Mobile Cloud Computing Framework for Exploiting Adjacent Service-Based Mobile Cloudlets," 2014.

[14] D. Fesehaye, Y. Gao, K. Nahrstedt and G. Wang, "Impact of Cloudlets on Interactive Mobile Cloud Applications," 2012.

[15] Y. Shi, S. Abhilash and K. Hwang, "Cloudlet Mesh for Securing Mobile Clouds from Intrusions and Network Attacks," 2015.

[16] "GreenIoT: An Energy-Efficient Internet-of-Things Platform for Open Data and Sustainable Development," [Online]. Available: http://user.it.uu.se/~eding810/GreenIoT.htm. [Accessed 04 February 2016].

[17] V. Bahl, "Cloudlets for Mobile Computing," in MSR Summer School on Advances in Wireless Networking, Bangalore.

[18] E. Cuervo, A. Balasubramanian, D.-k. Cho, A. Wolman, S. Saroiu, R. Chandra and P. Bahl, "MAUI: making smartphones last longer with code offload," in Proceedings of the 8th international conference on Mobile systems, applications, and services, 2010.

[19] B.-G. Chun, S. Ihm, P. Maniatis, M. Naik and A. Patti, "Clonecloud: elastic execution between mobile device and cloud," in Proceedings of the sixth conference on Computer systems, 2011.

[20] M.-R. Ra, A. Sheth, L. Mummert, P. Pillai, D. Wetherall and R. Govindan, "Odessa: enabling interactive perception applications on mobile devices," in Proceedings of the 9th international conference on Mobile systems, applications, and services, 2011.

[21] S. Kosta, A. Aucinas, P. Hui, R. Mortier and X. Zhang, "Thinkair: Dynamic resource allocation and parallel execution in the cloud for mobile code offloading," in INFOCOM Proceedings.

[22] K. Hong, D. Lillethun, U. Ramachandran, B. Ottenwälder and B. Koldehofe, "Mobile fog: a programming model for large-scale applications on the internet of things," in MCC '13 Proceedings of the second ACM SIGCOMM workshop on Mobile cloud computing, 2013.

[23] G. C. Fox, S. Kamburugamuve and R. Hartman, "Architecture and Measured Characteristics of a Cloud Based Internet of Things API," in Collaboration Technologies and Systems (CTS), 2012 International Conference on, 2013.

[24] A. M. M. Ali, N. M. Ahmad and A. H. M. Amin, "Cloudlet-based cyber foraging framework for distributed video surveillance provisioning," in Fourth World Congress on Information and Communication Technologies (WICT), 2014.

[25] M. Satyanarayanan, R. Schuster, M. Ebling, G. Fettweis, H. Flinck, K. Joshi and K. Sabnani, "An open ecosystem for mobile-cloud convergence," Communications Magazine, vol. 53, no. 3, p. 63–70, 2015.