Android-based Implementation of a Fog Computing and Networking Environment

Android-based Implementation of a Fog Computing and Networking Environment

Daniele Tarchi, Stefano Grandi and Walter Cerroni
Department of Electrical, Electronic and Information Engineering
University of Bologna, Bologna, Italy
email:daniele.tarchi@unibo.it, stefano.grandi8@studio.unibo.it, walter.cerroni@unibo.it

***Abstract*—The increasing number of devices and applications requesting external processing and storage facilities with reduced access latency has led to the introduction of edge computing solutions. Among others, Fog Computing can be considered as an edge computing solution enabling the edge devices to offer general-purpose processing and storage capabilities. Despite a huge research effort for proposing efficient Fog Computing solutions, their implementability is still under study. By resorting to the Cloud Computing service models, we propose here an Android-based proof-of-concept solution allowing to implement different Fog services in an edge scenario by using off-the-shelf end-user devices.**

I. Introduction

The Fog Computing paradigm was recently introduced to implement a distributed computing environment by taking advantage of the computational resources available at the network edge. Despite the increasing number of high-performance cloud computing infrastructures, they are typically located far from the edge of the network, where the interaction between user devices and data sources often takes place. Fog Computing fits this scenario by leveraging on the presence of a multitude of devices available at the edge that offer general purpose processing capabilities suitable to execute a wide range of applications [1], [2]. Moreover, exploiting processing resources at the edge allows to reduce the latency experienced by the end user when accessing remotely-executed services, thus complying with one of the most important requirements of the upcoming 5G communication standards [3].

Such a highly distributed environment not only involves the optimization of the processing resources deployment, but also requires to rethink the provisioning and management of networking resources, given the necessity of taking into account the characteristics of the communications links interconnecting the different devices involved. This is what is usually referred to as Fog Networking [4].

The importance of Fog Computing and Networking, and the increasing interest from both academia and industry in this subject, became clear since the beginning, when the OpenFog consortium was established to foster the development of an open Fog ecosystem [5]. The outcome of such as a joint effort of both industry and research centers was recently included also as a part of a new work-in-progress IEEE Standard [6]. Although under a different name, a similar concept has been also considered in the European context by the ETSI Multi-access Edge Computing (MEC) working group, aimed at studying the scenario where multiple nodes with computing capabilities and heterogeneous communication technologies are disseminated at the edge of the network [7]. Despite MEC and Fog Computing have lots of similarities, the MEC concept is more devoted to an architecture bringing a reduced number of computing nodes at the network edge, whereas Fog Computing turns toward a multi-layer architecture where different types of nodes from the lower-end device to the high-performance computing node cooperate for defining a distributed architecture able to cope with the user requirements while limiting the processing at the edge.

The interest in Fog Computing and Networking solutions has been rapidly raising in the last few years. In [4], an overview of the advantages of using the Fog architecture in Internet of Things (IoT) scenarios is introduced. In [8], an optimization framework for scheduling applications in a Fog Computing scenario is considered.Nonetheless, despite the large interest raised by the ideas behind Fog Computing and Networking, the availability of working implementations of an open Fog environment is still very limited [9], [10]. This is a significant gap that should be filled in order to prove the feasibility of the Fog ecosystem, as well as to quantify in practical terms the potential benefits that it can bring to both network service providers and customers.

This is the main motivation behind the contribution we provide in this paper, which presents a possible implementation of generic Fog Computing and Networking services assuming the presence of end-user devices available to share their processing capacity. Although the ultimate goal of our implementation is to enable a platform-independent system able to cope with different on-demand services, in this work we consider as a starting point a deployment based on the Android mobile operating system, due to its wide adoption by end-user devices and simple implementation. In particular, we present the design and development of a Fog Networking layer upon which generic Fog Computing services could be built. As a proof-of-concept implementation, we consider a computation offloading service [11]–[13] and test it through

This work has been partially supported by the project "GAUChO - A Green Adaptive Fog Computing and Networking Architecture" funded by the MIUR Progetti di Ricerca di Rilevante Interesse Nazionale (PRIN) Bando 2015 - grant 2015YPXH4W_004.

four different applications, having different characteristics in terms of requested amount of processing and data exchanged.

The paper is organized as follow. In section II we define the possible service models assumed in our distributed Fog Computing scenario. Then in section III we present the methodology we followed for the design of our Fog Networking layer. We report the testing results of our proof-of-concept implementation in section IV and, finally, we conclude the work in section V.

II. Service Models for a Distributed Fog Computing Scenario

The general scenario considered here in order to deploy a Fog Computing environment is composed by a number of connected devices, both fixed and mobile, able to interact among each other and create a localized Fog Networking infrastructure. The set of Fog-enabled devices is assumed to be heterogeneous and providing different computing, storage and communication capabilities, including for instance wireless access points, edge cloud nodes, portable devices, tablets, smart objects, etc. Under this assumption, the ultimate goal of our Fog Networking layer proof-of-concept implementation is to demonstrate the feasibility of the Fog concept by taking advantage of off-the-shelf equipment, such as System-on-Chip (SoC), System-on-Module (SoM) and Android-based devices. In the preliminary work presented in this paper, we limit our implementation to Android-based devices to show how is it possible to create an infrastructure capable of setting up a Fog Computing and Networking environment using off-the-shelf smartphones and tablets.

To this end, first we need to define the different service models that can be implemented in a Fog environment, and then discuss about their implementability on an Android-based platform.

A Fog Computing environment can be considered as an extension toward the edge of the well-known cloud computing environment. Therefore, following the classical cloud computing XaaS service model taxonomy, we decided to classify the Fog Computing applications into Software-as-a-Service (SaaS), Platform-as-a-Service (PaaS) and Infrastructure-as-a-Service (IaaS) categories [14].

In a cloud computing environment, a SaaS model allows to access a software application instance running on a remote cloud server through a thin client interface. This principle, when extended to a Fog environment, allows to implement a service where a server node provides a *specific* Fog application through a *specific* interface to another node, which runs a thin client application. This means that both client and server nodes should implement mutually compatible applications. The amount of exchanged data is limited to the input parameters/output results from one node to another.

In a cloud computing environment, a PaaS model allows to access a remote platform or operating system instance for developing custom applications, using programming languages, libraries and tools provided by the cloud. This principle, when extended to a Fog environment, allows to implement a service where a server node provides a *generic* Fog application through a *specific* library/platform. Within this scenario a client Fog node can send an application's source code to be compiled and executed in the server Fog node, if the latter provides a compatible programming environment. In this case the amount of exchanged data includes also the code/application to be executed.

Finally, in a cloud computing environment, an IaaS model allows to access virtualized computing, storage and networking resources for installing and running a custom system, including the whole processing stack. This principle, if extended to the Fog environment, allows to implement a model where a server node provides a *generic* Fog application on a *generic* platform through a programmable infrastructure (e.g., the virtualization environment). In this case a complete instance of a virtual machine, implementing the required application, is executed and the server node needs to support the virtualization system (i.e., the hypervisor). Due to the limited resources available in a Fog Computing node, lightweight virtualization paradigms must be considered, such as containers and unikernels. In this scenario the process becomes more complex since a direct offload of the whole virtualized environment from the requesting node to the server node can be unfeasible. Therefore, a repository should be foreseen for storing the available virtualized OS implementations to be used upon request from a Fog node. Indeed, in this case the amount of exchanged data includes the whole virtualized instance image.

It is quite clear that the implementation of the different cloud service models on a Fog environment offers similar characteristics: when going from the SaaS to the IaaS model through the PaaS, we move from the lowest to the highest grade of generality and flexibility, but also from the fastest to the slowest service delivery speed. In the following we will focus on an Android-based implementation of the SaaS and PaaS models. Indeed, due to limited resources typically available on Android-based devices in terms of virtualization, the implementation of an IaaS solution seems unfeasible with this platform, and we leave it to future extensions of our work. However, the proposed SaaS/PaaS implementation, despite being less flexible than the IaaS model, provides a simple and lightweight solution to be used in distributed environments, even without the presence of a fixed infrastructure, giving some interest to be used in some specific scenarios, such as isolated and emergency situations.

III. Android Fog Node Implementation

In this section, the main points related to the Android implementation of a Fog Service module are described. In order to avoid any misunderstanding, in the following we will refer to *Fog Service* as the generic service that can be delivered through a Fog network, and to *Android Service* as the Android-based implementation of the service module, as defined in the Android fundamental architecture [15].

The developed module is composed of two main parts. On one hand, an Android Service module implements the

necessary methods allowing the exchange of data among the nodes. On the other hand, an Android app, composed of different application types, is in charge of implementing different Fog Services by exploiting the Android Service. In particular we focused on the possibility of building a generic Android Service allowing to implement the previously introduced SaaS and PaaS models, to be used by the Android apps for realizing the desired Fog Service.

The Android Service, named *FogNetworkService*, implements two interfaces:

- *IFogNetworkService*, used to execute the Fog Services by exchanging the messages among the Fog Nodes;
- *INetwork*, allowing to create and manage a Fog Network.

A. Android Fog Service Interface

The IFogNetworkService is implemented by using the Android Interface Definition Language (AIDL) and implements two methods, *invokeCode()* and *invokeInterface()*, used for implementing the PaaS and the SaaS models, respectively. Both methods require two parameters as arguments, named *FogParams* and *FogResults*, that are two objects used for exchanging the input parameters and the output of the executed process at the computing Fog node. This implementation allows to have multiple instances of the *IFogNetworkInterface* so that each Fog node can implement different Fog services at the same time by using different *IFogNetworkInterface* instances.

The *invokeInterface()* method allows to execute a remote application, by exchanging the application-dependent input parameters. To this aim, the node implementing the Fog application is designed to execute a certain task on the received parameters by using its libraries. This can be seen as a SaaS model implementation on Android-based Fog Computing nodes. While this method can gain in terms of execution efficiency of the requested application, it gives lower flexibility limiting the Fog interaction only to the specific parameters defined as input for the considered application.

The *invokeCode()* method, instead, allows to remotely execute an application by sending the code to be executed at the remote fog node. This allows to implement the service with higher flexibility by resorting to the PaaS model. The *invokeCode()* method is implemented by having as input the code to be executed and as an output the result of the executed code. This method allows to implement a much more flexible service, while requiring more time for the code execution due to the indirect implementation of the requested Fog application as an Android library. In particular, by gaining from the fact that Android apps can be implementation by coding in Java, we resorted to BeanShell [16], a lightweight scripting language able to remotely execute Java programs. Beanshell is implemented in any of the nodes in the network able to receive some code from the others to be executed through the *invokeCode()* method.

In order to implement any of the possible interfaces, a specific method, called *addInterface()*, is used, giving the possibility to a node to offer such interface to the other nodes; similarly, *removeInterface()* is used for removing such an interface and avoiding other nodes to use it.

B. The Android Fog Network Interface

The *INetwork* interface is used for exchanging the messages needed to create the Fog Network by implementing a network discovery protocol; to this aim, aiming at simple and reliable solution, we resorted to a broadcast message sent by each Fog node to inform the other nodes about its presence and the Fog services it offers. In particular, a distributed approach is considered. Each Fog node entering in a network will send the discovery message on a specific socket. Every other node present in the same network will reply to notify its presence and the offered Fog services. This handshaking mechanism allows each node to store the list of available nodes and the related offered Fog services, without the need of a centralized infrastructure.

After the Fog Network creation, in order to implement the Fog Service a suitable protocol has been implemented. The Fog Network works by using two sockets. A UDP socket working on a predefined port number allows to exchange the control information of the Fog service, while a TCP socket, activated upon a new request between two peers should be established, allows to exchange the data by using one of the two models, i.e., PaaS or SaaS, by using one of the two methods previously introduced, i.e., *invokeInterface()* or *invokeCode()*. To this aim both server and client sides have been considered, where the server side takes into account the interface exposition and the computation, while the client side takes into account the service request and the computation request. Within the *FogNetworkService* class there are two listening threads:

- *ServiceListeningSocketThread* allows to create a UDP socket always listening on a predefined port number. Every message received by this handler is then passed to the HandlerThread
- *HandlerThread* manages the received message by creating a separate execution thread for each received request.

The proposed solution allows to manage different requests at the same time, even related to different services by each node, by exposing more interfaces. The incoming requests are queued at the handler.

At the client side, an AsyncTask instance manages the service request to be executed. AsyncTask is an Android class enabling to perform background operations and publish results on the User Interface thread in an asynchronous way, avoiding to halt any other threads and/or handlers. Therefore, the result of the remote operation can be waited without blocking the requesting node operations.

In Fig. 1 the whole set of protocol messages is reported, where the main messages are highlighted with numbers:

1) The client node creates a *ServerSocket()*, that is a TCP socket endpoint.
2) The client node sends a UDP message to the server, including the request type, and the port number of its ServerSocket.

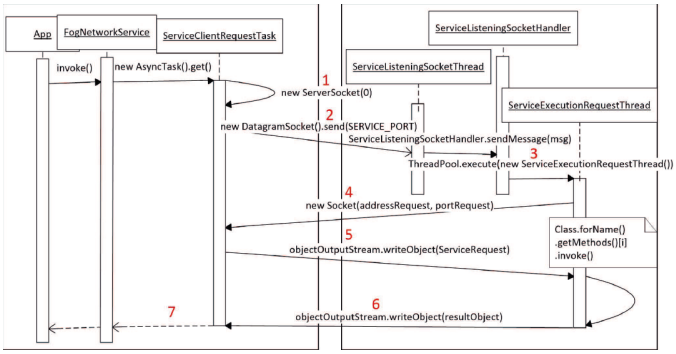


Fig. 1. Protocol messages exchanged between client and server for setting up a Fog Service.

TABLE I
TESTBED COMPONENTS

Device type	Number	Characteristics	Usage
Rasberry Pi3	1	CPU 4×ARM Cortex-A53 1.2GHz; RAM 1 GB; 10/100 Ethernet, Wi-Fi 2.4GHz 802.11n, Bluetooth 4.1 & LE; microSD; GPIO 40-pin.	Fog Access Point
Asus ZE520KL	1	Android 8.0; CPU Octa-core 2.0 GHz Cortex-A53; RAM 4GB; Wi-Fi 802.11 a/b/g/n/ac	Fog Node
Samsung SM-T705	1	Android 6.0; CPU Quad-core 2.3 GHz Krait 400; RAM 3GB; Wi-Fi 802.11 a/b/g/n	Fog Node
LG Nexus 5	1	Android 6.0; CPU Quad-core 2.3 GHz Krait 400; RAM 2GB; Wi-Fi 802.11 a/b/g/n	Fog Node

3) The server node receives the request, and creates a thread for elaborating the request.
4) During the thread execution the server connects to the TCP client socket.
5) After creating the TCP link between the nodes, the client sends to the server the information for starting the service (interface name and operation) and the parameters (values or code to be executed)
6) The server executes the request and sends back the result on the same TCP connection
7) The client terminates the execution when the task sends the received result to the app.

IV. COMPUTATION OFFLOADING PROOF-OF-CONCEPT SOLUTION

The proof-of-concept solution has been implemented by using different device types, each one with different characteristics. The devices used in the testbed are listed in Table I; in particular three devices have been used as Fog nodes, while the Raspberry Pi3 has been used as a WiFi AP.

The solution copes with three main phases able to provide and execute Fog services:

1) Network Discovery: when a node is turned on, it should start a discovery phase for searching other nodes nearby capable of building any of the Fog service models;
2) Service Discovery: after having discovered any node, the incumbent node should be able to discover which services are implemented by the nodes in the area;
3) Service Delivery: the Fog service implementation is performed by following one of the previously introduced models depending on the requested service.

In the first phase each Fog node, when entering in a network, sends a broadcast message on a predefined socket. If any other Fog node is present in the network, an acknowledgement message is sent, including the list of the available Fog services. This message is implementing the second phase. In this paper we will focus mainly on the third phase by assuming to have an already established network with a given number of Fog nodes, each one aware of the services that can be delivered by the other nodes.

The components were used to set up a testing environment for computation offloading in a Fog Computing environment. To this aim four different types of application were tested by leveraging on a suitable Android implementation of the Fog Network primitives, and based on SaaS and PaaS models:

- Calculator: A calculator applications that allows to exploit a remote Android device for executing some simple algebraic operations;
- Integral: A remote integral solver;
- Image Compression: A remote JPEG compression service;
- Hash Calculator: A remote brute force hash calculator.

The implemented apps exploit the underlying Android Services described in the previous section for implementing the related services by using different models.

The Calculator has been implemented in both SaaS and PaaS models. The SaaS implementation has been performed with a limited subset of operations. This is due to the fact that when designing the services to be executed as a SaaS even the operation is one of the parameters to be exchanged between the nodes, and, hence, the possible operations should be a predefined set. On the other side, in the PaaS implementation a generic Java code could be sent remotely to be executed on the other side. In this way any possible algebraic operation can be executed, by exploiting the related mathematical Java class. The flexibility is evident by noticing that in this case we can execute any complex set of different mathematical operations.

The Integral solver app allows to solve a definite integral. To this aim we selected to solve the integral of the cosine function between two bounds. In this case we have instead considered three comparisons. On one side a SaaS implementation exploiting a suitable Android mathematical library. On the other side the same solver was implemented locally allowing to compare the remote and the local solutions in terms of delay. Finally, we implemented a partial offloading mechanism by dividing the definite integral in two equally bounded definite integrals and solving one half locally and the other remotely.

The image compressor implements both local and SaaS JPEG compression of an uncompressed bitmap image. Due

TABLE II
ALGEBRAIC OPERATIONS APP DELAYS

Device	SaaS [ms]	PaaS [ms]
ASUS ZE520KL	178.8 (± 78.8)	216.5 (± 57.6)
Samsung SM-T705	233.9 (± 88.2)	248.7 (± 99.7)

to the complexity of the code only the SaaS model has been implemented by comparing it with the local compression. In this case the test allows to consider the case in which the exchanged data is relevant; we have considered a 256 color bitmap version of the Lena sample image with resolution 512×512 and size 222 kB. The compression has been implemented by resorting to a suitable image processing library in Android.

Finally, the fourth app implements the iterative execution of a hashing operation through the HMAC algorithm [17], to discover the key given a string and its Hash. This operation has been executed locally and remotely by exploiting both SaaS and PaaS models. The SaaS model has been implemented through the exchange of the string between two different Android nodes. Differently, the PaaS model has been implemented by exploiting the BeanShell scripting installed on Android devices. In this case the requesting node is sending the entire Java code that is executed remotely. The results are also compared with a local execution.

In the following the time required to completely deliver different services has been reported. Unless otherwise stated, in the following tests we have considered that the LG Nexus 5 device is the client device while the other devices are selected as servers. The measured values are averaged over 10 trials, by indicating also the standard deviation. Those values represent the whole remote execution, including both transmission/reception and processing phases. The LG Nexus 5 has been selected being the less powerful device, even if the goal of the tests more than evaluating the numerical values aims at demonstrating the correctness of the implemented solution.

In Table II, the results for the remote algebraic operations are reported, where each device represents the selected remote node performing the calculation. In particular we selected a simple addition of two integers. It is interesting to notice that the processing time is lower when exploiting a higher performing device. Moreover SaaS allows to have a lower latency with respect to PaaS, despite a lower flexibility. It is worth to be noticed that in this case the additional delay introduced by the PaaS library is reduced mainly due to the simple operation that is performed. It is worth to be noticed that in case of PaaS more complex operations could be performed by sending a user defined Java code to the processing node.

In Table III, the results for the definite integral calculation are reported by comparing the local integration with the SaaS implementation and the partial local and SaaS implementation. In particular we resorted to the integration of the cosine function between -10000 and 10000 in order

TABLE III
DEFINITE INTEGRAL ESTIMATION DELAYS

Device	Local [ms]	SaaS [ms]	Local&SaaS [ms]
LG Nexus 5	1816.8 (± 21)	-	-
ASUS ZE520KL	61.6(± 8.4)	461.7(± 163.4)	920.5(± 27.3) & 293(± 88.3)
Samsung SM-T705	1439.5(± 80.7)	1723.7 (± 81.4)	907(± 14.3) & 1100.5(± 120.4)

TABLE IV
JPEG COMPRESSION DELAY

Device	Local [ms]	SaaS [ms]
LG Nexus 5	51.6 (± 12.6)	-
ASUS ZE520KL	39.6(± 16.5)	536 (± 68.1)
Samsung SM-T705	60.6 (± 13.2)	582 (± 116.7)

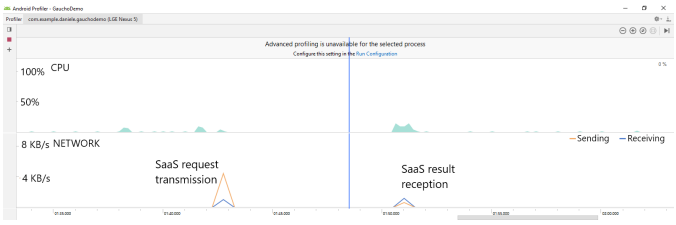
to perform a high number of operations. Both local and SaaS are implemented by using Romberg's method [18] for estimating the definite integral, available trough the Java package org.apache.commons.math.analysis.integration. In case of partial offloading we divide the definite integral in two equal parts; the first from -10000 to 0 computed locally while the rest offloaded. By analyzing the time needed for local computation it is possible to see the impressive improvement when using a higher performance device (i.e, the ASUS ZE520KL) when performing heavy processing. To this aim it is interesting to see the second column, reporting the time needed for performing the processing by the device in the related row when requested by the LG Nexus 5. Here it is clear the advantage of the SaaS approach, in particular by looking to the ASUS ZE520KL that allows to reduce the time. The additional time with respect to the local computation at the same device is due to the transmission/reception time and additional Android operations. Even more interesting the results of the partial offloading where half of the processing is performed locally and half is offloaded. In this case it is possible to notice that both SaaS and local are reduced to about one half, as expected. Since the local and SaaS processing can be done in parallel, this incurs in a remarkable gain when using the Samsung device, allowing to reduce to about one half the overall processing time.

In Table IV, the results for the image compression are reported, by comparing the local compression and the SaaS compression implemented through the ImageIO Java class. In this case the local compression is always performed with the lowest amount of time, while offloading does not have a remarkable impact on using one of the two devices. This is mainly due to the fact that in this case most of the time is spent for exchanging the image. However, even if with a higher delay, it is possible to see that in case one node cannot compress an image, this facility can be easily implemented remotely.

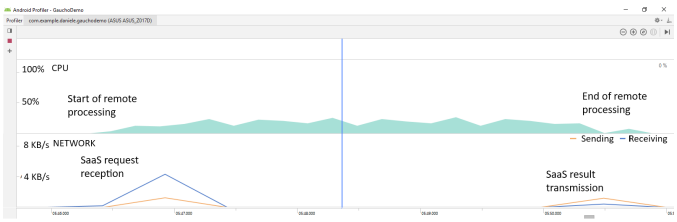
In Table V, the results for the iterative execution of a hashing operation through the HMAC algorithm are represented. It is possible to notice that the SaaS execution delay is higher than the local execution but the increase is not remarkable,

TABLE V
ITERATIVE OPERATION DELAYS

Device	Local [ms]	SaaS [ms]	PaaS [ms]
LG Nexus 5	76 (± 8.3)	-	-
ASUS ZE520KL	35 (± 20)	247.5 (± 52.8)	12078.1 (± 663)
Samsung SM-T705	51 (± 3.9)	255.4 (± 75.8)	15933.7 (± 427.3)



(a) Client Profiling (LG Nexus 5)



(b) Server Profiling (ASUS ZE520KL)

Fig. 2. Definite Integral SaaS Profiling

while the PaaS execution requires much more time. This is due to the fact that the Android devices should execute a given code on an external library requiring much more processing effort. It is also possible to notice that the device processing capabilities do not affect the SaaS execution time, while PaaS is affected by the device capabilities. Also in this case it is worth to be noticed that the PaaS case, despite a higher processing time, provides higher flexibility allowing the execution of any user-defined Java code on the target device.

Finally, we have used the Android Studio profiler for taking a look to the time spent by each device during one of the previous operations. Due to space limit we are showing here the results for the remote Integral operation.

In particular in Fig. 2a, it is possible to notice that the LG Nexus 5 is using the network interface two times, when sending and receiving the result of the request, while between the two network usage the CPU is not used. This is the reverse in Fig. 2b where instead the ASUS is mainly receiving then the CPU is used and finally the result is sent back.

In Fig. 3, we are showing the profiling for the client side when partial offloading is performed. It is possible to notice that the client is sending and receiving the SaaS result while at the same time the CPU is occupied for the local processing.

V. CONCLUSION

Fog Computing enables the usage of edge devices for implementing a computational sharing environment. The aim of this work is to show a proof-of-concept implementation of PaaS and SaaS models on a Fog Computing environment by using Android-based devices. The correctness of the solution has been proved by resorting to four possible applications implementing computation offloading with different schemes.

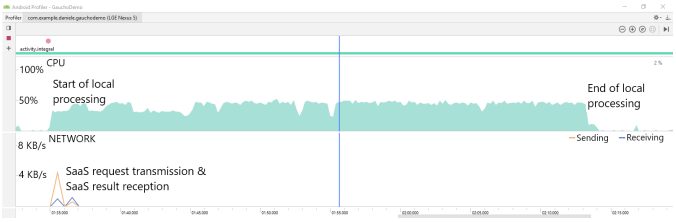


Fig. 3. Partial Definite Integral SaaS Client Profiling (LG Nexus 5)

REFERENCES

[1] M. Mukherjee, L. Shu, and D. Wang, "Survey of fog computing: Fundamental, network applications, and research challenges," *IEEE Commun. Surveys Tuts.*, vol. 20, no. 3, pp. 1826–1857, Third Quarter 2018.

[2] C. Mouradian, D. Naboulsi, S. Yangui, R. H. Glitho, M. J. Morrow, and P. A. Polakos, "A comprehensive survey on fog computing: State-of-the-art and research challenges," *IEEE Commun. Surveys Tuts.*, vol. 20, no. 1, pp. 416–464, First Quarter 2018.

[3] Y. Ku, D. Lin, C. Lee, P. Hsieh, H. Wei, C. Chou, and A. Pang, "5G radio access network design with the fog paradigm: Confluence of communications and computing," *IEEE Commun. Mag.*, vol. 55, no. 4, pp. 46–52, Apr. 2017.

[4] M. Chiang and T. Zhang, "Fog and IoT: An overview of research opportunities," *IEEE Internet Things J.*, vol. 3, no. 6, pp. 854–864, Dec. 2016.

[5] (2018, Oct.) OpenFog Consortium. [Online]. Available: https://www.openfogconsortium.org

[6] *IEEE Standard for Adoption of OpenFog Reference Architecture for Fog Computing*, IEEE Std. 1934-2018, Jun. 2018.

[7] (2018, Oct.) Multi-access Edge Computing. [Online]. Available: https://www.etsi.org/technologies-clusters/technologies/multi-access-edge-computing

[8] L. F. Bittencourt, J. Diaz-Montes, R. Buyya, O. F. Rana, and M. Parashar, "Mobility-aware application scheduling in fog computing," *IEEE Cloud Computing*, vol. 4, no. 2, pp. 26–35, Mar. 2017.

[9] R. Craciunescu, A. Mihovska, M. Mihaylov, S. Kyriazakos, R. Prasad, and S. Halunga, "Implementation of fog computing for reliable e-health applications," in *2015 49th Asilomar Conference on Signals, Systems and Computers*, Pacific Grove, CA, USA, Nov. 2015, pp. 459–463.

[10] A. Brogi, S. Forti, A. Ibrahim, and L. Rinaldi, "Bonsai in the fog: An active learning lab with fog computing," in *2018 Third International Conference on Fog and Mobile Edge Computing (FMEC)*, Barcelona, Spain, Apr. 2018, pp. 79–86.

[11] D. Mazza, D. Tarchi, and G. E. Corazza, "A unified urban mobile cloud computing offloading mechanism for smart cities," *IEEE Commun. Mag.*, vol. 55, no. 3, pp. 30–37, Mar. 2017.

[12] A. Bozorgchenani, D. Tarchi, and G. E. Corazza, "Centralized and distributed architectures for energy and delay efficient fog network based edge computing services," *IEEE Trans. Green Commun. and Netw.*, Dec. 2018, early view.

[13] L. Liu, Z. Chang, X. Guo, S. Mao, and T. Ristaniemi, "Multiobjective optimization for computation offloading in fog computing," *IEEE Internet Things J.*, vol. 5, no. 1, pp. 283–294, Feb. 2018.

[14] P. Mell and T. Grance, "The NIST Definition of Cloud Computing," The National Institute of Standards and Technology, SP 800-145, Sep. 2011. [Online]. Available: https://doi.org/10.6028/NIST.SP.800-145

[15] (2018, Oct.) Services overview | Android Developers. [Online]. Available: https://developer.android.com/guide/components/services

[16] (2016, Feb.) Beanshell - lightweight scripting for Java. [Online]. Available: http://www.beanshell.org/

[17] "HMAC: Keyed-hashing for message authentication," RFC 2104, Feb. 1997. [Online]. Available: https://tools.ietf.org/html/rfc2104

[18] W. Romberg, "Vereinfachte numerische integration," *Det Kongelige Norske Videnskabers Selskab Forhandlinger*, vol. 28, no. 7, pp. 30–36, 1955.