SOSE: Smart Offloading Scheme using Computing Resources of Nearby Wireless Devices for Edge Computing Services

Ali Al-ameri and Ihsan Alshahib Lami

School of Computing, The University of Buckingham
Buckingham MK18 1EG, UK

`first.last@buckingham.ac.uk`

Abstract. Offloading of all or part of any cloud service computation, when running processing-intensive mobile cloud computing services (MCCS), to servers in the cloud introduces time delay and communication overhead. Edge computing has emerged to resolve these issues, by shifting part of the service computation from the cloud to Edge servers near the end-devices. An innovative Smart Cooperative Computation Offloading Framework (SCCOF) to leverage computation offloading to the cloud has been previously published by us [1]. This paper proposes SOSE; a solution to offload sub-tasks to nearby devices, on-the-go, that will form an "edge computing resource, we call SOSE_EDGE" so to enable the execution of the MCCS on any end-device. This is achieved by using short-range wireless connectivity to network between available cooperative end-devices. SOSE can partition the MCCS workload to execute among a pool of Offloadees (nearby end-devises such as Smartphones, tablets, and PC's) to achieve minimum latency and improve performance while reducing energy consumption of the Offloader (end-device that is running the MCCS). SOSE established the Edge computing resource by: (1) Profiling and partitioning the service workload to subtasks based on a complexity relationship we developed, (2) establishing peer2peer remote connection with the available cooperative nearby Offloadees based on SOSE assessment criteria, (3) migrating the subtasks to the target edge devices in parallel and retrieve results. Scenarios and experiments to evaluate SOSE show that a significant improvement in terms of execution time (>40%) and energy consumption (>28%) has been achieved when compared with cloud offloading solutions.

Keywords: Offloading, Edge Computing, Cooperative, Mobile Cloud Computing

1 Introduction

The Smartphone (SP) is continually being improved to have more and more computational resources and connectivity, amongst many others such as memory, display, sensors, battery, etc. Nevertheless, SP's are still lacking behind in terms of performance and battery capacity, which are the main desired features for SP subscribers [2]. SP's are now being used for running resource intensive MCCS, such as Tracking humans or animals in crowd sensing scenarios or "manipulating blind persons" via IoT Sensors [3]. Some of these MCCS require machine learning and AI algorithms to be executing live. Current SP's will run out of puff processing, and the battery will run flat when running such MCCS.

We believe that there will always be a big gap between SP resource offerings and developers of intensive processing MCCS. To fill this gap, many offloading solutions exist that ships the processing of such MCCS to a central server in the cloud. This intern will create large traffic in an already crowded spectrum. I.e. offloading the computation to servers in the cloud introduces time delay and communication overhead cost. Edge computing has been emerged to resolve these issues by shifting the computation from servers in the cloud to servers near the edge to reduce both delay and communication cost. However, edge computing servers normally are planned as part of the infrastructure of the cloud in the vicinity. SOSE overcomes this limitation! SOSE a scheme that forms an edge computing resource to execute such MCCS on the go from cooperative nearby edge devices. SOSE offloads sub-tasks for computation from the host device/SP of the MCCS to a network of nearby SP's/devices. Figure 1 shows SOSE end2end scheme. It shows that, the cloud server is used to host SOSE_INTELLIGENT, an intelligent engine to recruit cooperative end devices and authenticate their availability when needed. Also, SOSE_INTELLIGENT engine provides the end-device with decisions on the best scenario to partition and offload to achieve a low execution time and reduce the energy consumption. It also shows the newly formed SOSE_EDGE computing resource network (dotted circle in the diagram).

The Offloader will ask SOSE_INTELLIGENT engine for decisions of nearest device that has the lowest load and the highest resources of processing and battery capacity, as well as the best network connectivity to use. Then the Offloader will generate VMs (bundle them as APKs and JAR files) of all the partitioned subtasks and will establish connectivity with all available Offloadees as advised by SOSE-INTELLIGENT engine. Finally, the Offloader will offload the VM's to the Offloadees and retrieve the results.
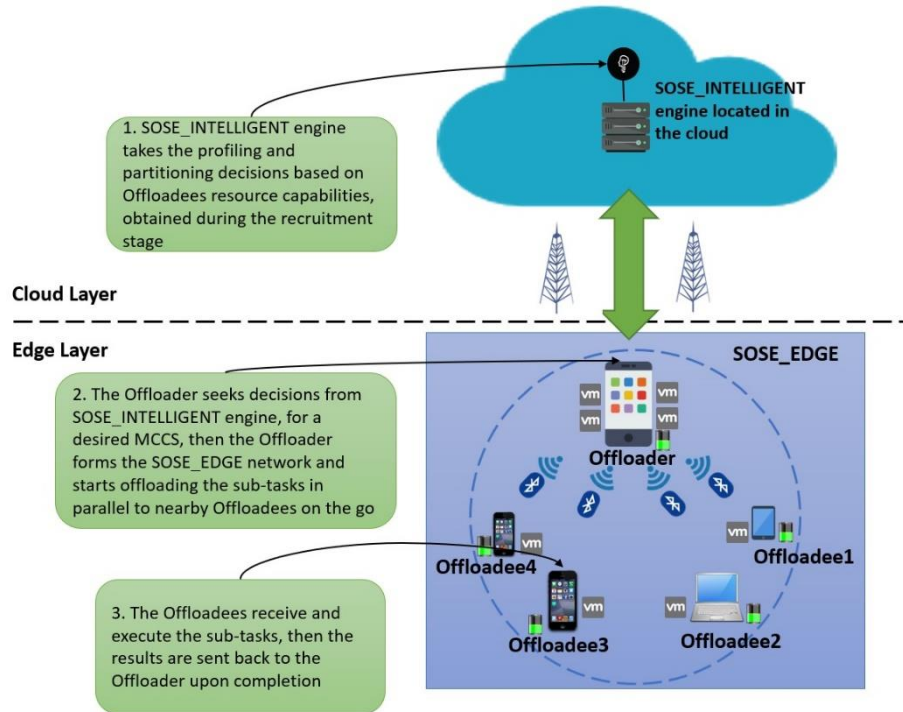
Fig. 1. SOSE Scheme

The novelty contributions of this paper are:

- Introduces SOSE, a unique scheme that forms the edge computing resource, on the go, from nearby devices and share the execution of the MCCS in parallel among them via short range wireless connectivity.
- The offloading between the devices on SOSE_EDGE is done intelligently by an SOSE_INTELLIGENT engine based in the cloud. SOSE_INTELLIGENT engine recruits cooperative device resources and monitors (processing capability, battery status, and availability) and authenticates (access, session keys and engagement status) them so to advice on available device nearby when the Offloader needs to form the SOSE_EDGE.

The rest of this paper includes: Section 2 that summarizes the recent literature on edge computing implementations, while Section 3 presents the development of SOSE. Section 4 presents the experiments, results and analysis. Conclusions and further work are presented in section 5.

2 Recent literature on edge computing implementations

Review of solutions that perform offloading to a centralised server in the cloud have been published in in our previous paper [1]. This review focuses on implementations/solutions that consider IoT intensive applications which offloads to nearby pre-setup infrastructure of edge servers. SOSE proposed to deploy the SOSE_EDGE solution on the go when needed. This is achieved by recruiting a group of available processing resources/devices nearby in a local network to form a cooperative sharing environment using SOSE_INTELLIGENT engine.

IoT deployments have increased the amount of data generated to the cloud; the amount of data hosted in 2018 is equal to the data gathered in all prior years [4]. This has necessitated that data-handling tasks are shifted to the edge nearer to the IoT sensors network, and so typical existing solutions focuses on offloading between the edge servers and the cloud center. Running these IoT services on cloud servers can have a negative impact on the offloading process due to network cost and bandwidth traffic. Therefore, an advantage of Edge computing is to provide resources near end users/nodes so to reduce long execution time and end-node battery consumption. A solution that facilitates offloading of complex services from a mobile node to an edge computing server has introduced a model that provides the use of virtual resources in the edge servers [5]. It achieves this by shifting the service execution from a single mobile node to the edge servers automatically, by dividing a single task to 5 subtasks using 0-1 integer liner programming method. It marks the tasks with a value of (0,1) where "0" stands for tasks to run locally on the mobile node, such tasks that access mobile local features or input and output tasks, while "1" stands for tasks to run on edge server which has multiple virtual resources to handle the execution of subtasks. This then followed by a "decision solver" engine to decide on which virtual resource to select for the incoming 5 subtasks, based on the virtual resource "current queue and completion time". Experiments have affirmed that performing the execution at the edge servers can reduce the network cost and internet traffic. However, this model requires pre-setup infrastructure which is difficult to predict for IoT network type computation and so we believe a more dynamic model/solution that forms the edge computing resources on the go is needed so to achieve faster execution time.

Tracking humans or animals with drones in crowd sensing scenarios like volcanos or disasters are examples of nowadays IoT applications. These applications require machine learning and AI algorithms engines to analyze streams of audio, video and image data coming from many sensors. Such intelligent algorithms require a significant computational/processing resources that are not typically available at the edge, but rather available in large data centers in the cloud. An offloading solution that balance the computational workload between the available cloud and the edge resources has been proposed in [6]. It achieves this by shifting the training and testing phases of the workload to the cloud. I.e. the end-node/device will upload data, which are then labelled and tested by multiple ML algorithms, then, based on the chosen decision, the model is retrieved, sterilized and packed in a shared repository. Only the AI inference engine is positioned at the edge as a micro service that can be accessed through the shared repository. This model is impressive in that it sends less data to the cloud,

which reduces network cost and bandwidth traffic. However, it lacks a dynamic partitioning algorithm that decides if a task is executed in the cloud/edge servers, but rather depends on a pre-processing developer analysis to decide where to execute every task. We believe that the concept of letting the cloud be responsible of the overall decision making in splitting the computation workload between the edge and the cloud is commendable. We shall deploy a similar concept in SOSE, we used AWS services to perform the creation of the DB and recognition using AWS rekognition service [7], only the recognition results of the extracted faces are saved in a local DB shared repository using SQLite.

A solution that enhances the above described offloading model by including a dynamic partitioning algorithm of tasks moved between the cloud and the edge is achieved by including an "optimal virtual machine selection technique" and a "dynamic task partitioning algorithm" [8]. These two algorithms offload the intensive tasks from end-device to the edge server and/or the cloud server. It achieves this by (1) sort algorithm that topologically analyze a "task graph" to partition the tasks between edge and cloud servers to achieve a low computational complexity. (2) Then it ranks the available virtual machines based on the time it takes to execute in each virtual machine, (3) it selects the appropriate virtual machine and utilize the dynamic task partitioning algorithm to compute the minimum completion time for the executed task. However, it only considers execution time as a metric to evaluate the proposed model. We deploy a similar concept to execute the subtasks in parallel on the nearby edge devices so faster execution time can be achieved. We also developed and calculated other metrics like computational complexity, battery consumption and communication overhead.

Some IoT apps required DL algorithms to extract accurate information for classification, especially for IoT devices deployed in complex environments such as vehicle tracking animals, drones tracking surveillance, e-health apps, DL-driven smart video surveillance and smart homes. (i.e. DL can predict accurately home electricity power consumption using data collected from smart meters). Nevertheless, DL algorithms require a significant amount of processing because each DL extra layer can bring extra processing among its multilayer structure. Therefore, Efficient scheduling mechanisms are needed to decide on how many DL layers can run on the edge servers. A solution that facilities offloading strategy to optimize the performance of DL for IoT at the edge has introduced a model that provides offline and online scheduling mechanism [9]. It achieves this by checking each server capacity to decide how many layers each server can handle. I.e. the first input layers are consisting of many processing compute layers, therefore it is more beneficial to run in the cloud server. Then, when the dimension of the DL network is reduced, and the size of the intermediate layers becomes smaller than the input layer. This allows moving the processing of these lower layers to the edge server. This proposed model uses AlexNet DL model which consists of 8 layers, the first 5 layers are deployed in the cloud server and the last 3 layers are deployed in the edge server. This model is unique in that it can generates less data transfer and reduces the response latency. This inspired us to form SOSE by forming a network of resource from end-nodes and schedule the subtasks among them. I.e. SOSE_INTELLIGENT engine schedules the subtasks and selects

the device with the lowest load and has the highest resources in terms of processing power and battery level.

Offloading the intensive processing tasks and sharing the end-user data to the cloud or edge servers lead to an unsecure deployment inviting malicious activities. A solution that proposes to secure the offloading process has introduced a model that secures the data being shared between the edge servers during offloading [10]. It achieves this by (1) it segments and offloads the tasks to the edge server in a sequence order. (2) It syncs to the edge server through a middleware that handles the communication. (3) It provides a security manager interface to encrypt, exchange security keys and verify the data before offloading, it is responsible to monitor the offloading process and generate alerts if a breach occurred by observing all the edge devices. Despite the fact that, to the best of our knowledge this proposed model is the first to addresses security issues when offloading IoT apps to the edge server. Nevertheless, it lacks details of the used mechanism nor experiments to approve the novelty. Being said that, SOSE introduces (1) a SOSE_INTELLIGENT engine based in the cloud server that (monitors and approve) the nearby end-devices for qualifying as being secure and fit before offloading. (2) Partitions the tasks and distribute the subtasks among a variety of nearby edge devices, so the shared data cannot be retrieved or invoked as a package, and so stealing the subtask will not impact the overall security of the offloading. (3), We are using AWS rekognition service, which is a highly secure service that uses access and secret keys to authenticate the nearby devices. (4) We used nearby peer2peer API protocol [11] to communicate the nearby devices, which is a secure middleware that provides fully encrypted P2P data transfer between nearby edge devices.

3　SOSE Architecture

There are two distinct engines that make SOSE function. The SOSE_INTELLIGENT engine and the SOSE_EDGE.

3.1　SOSE_INTELLIGENT engine

This engine is meant to be active at all times. It is based in the cloud. Its main functions are:

1. Identify and recruit suitable devices that can be used when needed by SOSE_EDGE. This process is continuous, and we envisage that such devices, as a principle, are SP's that are willing to contribute to help other SP's when running demanding MCCS. We propose that such devices are assigned certain credits that they will be able to use when they are the one running the MCCS. A suitable arrangement for controlling this will need to be in place, but out of the scope of this paper. Therefore, this engine will have a database of such devices, their local localisation, their resources, typical usage, availability and current load.
2. When contacted by the SOSE_EDGE Offloader, this engine will: a) Perform profiling and partitioning of the MCCS, if not already done in a previous request. B)

try and establish if such MCCS has been run elsewhere to learn from that experience (resource required, time to execute, and dependency between tasks), c) provide a list of potential available SPs/devices near the location of the Offloader together with their capability, d) advice the Offloader with the MCCS profiling and partitioning process. This information will help the Offloader to generate the Virtual Machines (VMs) that will form the sub-tasks to be offloaded to nearby devices.

3.2   SOSE_EDGE

This engine performs various stages resulting in forming the edge computing resource that will execute the MCCS and is led by the SP that is hosting the MCCS (Called the Offloader here). Any participating device in helping to run the sub-tasks are called the Offloadee. The process of SOSE_EDGE is as follows:

1. The Offloader will generate VMs (bundle them as APKs and JAR files) of all the partitioned subtasks, based on the instructions provided by the SOSE_INTELLIGENT engine. Note that the choice of having the profiling and partitioning tasks of the MCCS in the cloud was to save battery of the Offloader, and source knowledge of the MCCS provided by the developer is more accessible to the cloud.
2. The Offloader will establish connectivity with all available Offloadees as advised by the SOSE_INTELLIGENT engine. Note that the connectivity will be wireless, and that SOSE_INTELLIGENT engine will advise on the best wireless technology to use (e.g. Wi-Fi or BT, or Cellular) for each Offloadee device.
3. The Offloader will offload the VM's to the Offloadees and communicate the results from this process appropriately, including the termination of the contact.
4. The Offloader will also be executing its own share of the sub-tasks as and when it is not busy with the other tasks.
5. When the MCCS run is completed, a summary record of this experience is feedback to SOSE_INTELLIGENT engine to train and update it for future execution if needed by any other Offloader.

Details of each of these steps will be detailed as part of the experiments we have done to prove the concept of SOSE. For example, all wireless connectivity is done on a peer2peer protocol, etc.

4   Experiments, Results & Analysis

The following experiment scenarios is used to prove that SOSE_EDGE can provide an on-the-go (dynamic) edge resource from available nearby devices, and will perform as good as, or better than, a structured pre-setup edge computing engine. The details of the implementation of SOSE_INTELLIGENT engine and the automation of the process will be documented elsewhere as being not the focus of this paper.

4.1   MCCS choice: Face Detection Service (FDS)

FDS is chosen to demonstrate the computational complexity and the benefits of offloading (typically used by police or at an airport mobile search activities). It involves a variety of complex tasks including face detection and feature extraction. Processing one video frame using face detector takes an average of 3.2 seconds and costs 9.0 Joules using a Samsung Galaxy Nexus i9250 [12], which means continually running FDS would drain a SP fully charged battery within 1.5 hour. We developed FDS using Android studio platform and Dlib library, which is an open source library for image detection and recognition. It obtained a face bounding box using (x,y) coordinators of the face in the image, and then it detects and draw 68 (x,y) coordinators in the face, and finally, it extract the face features. Asysnc class is basically used to run the heavy part of FDS algorithm on another thread so no pressure on the main thread that is also handling the Graphic user interface. We use the mface.train function to train the algorithm to perform the image detection process. Then we called the recognizeAsync function to execute the algorithm. Then when we input a new image, the algorithm detects the faces and extract the features. Full details about the specification of the scenario and experimental devices are illustrated in section 4.2.

To illustrate more sub-tasks, we also developed a more complex version of FDS, we call FDSC. This includes recognition functions. As shown in Figure 2, the main GUI of FDSC contains three main buttons which are Offloader, Offloadee and server. The offloader button is to specify whether to run the tasks locally on Offloader or remotely on Offloadees. It shows a drop-down list of Offloadees {0-3}, (we have decided to use up to 4 Offloadees in this experiment, (note that the maximum number of devices to be used is 7 because the BT protocol only allows 7 actual devices to connect to one master node [13])). The {0} means the task run locally on the Offloader, while {1-3} specify the number of Offloadees. The Offloadee button is to represent the participated Offloadees. The server button is for running the task remotely on the server, (we have decided to use 2 servers in this experiment, the first one is a cloud AWS EC2 server, and the second is a local Edge WAMP server), it requires a server IP address to start the connection. After deciding where to offload the tasks, FDSC then displays 2 buttons weather to "select an image" from the mobile device memory or "click picture" to capture real images on the go.
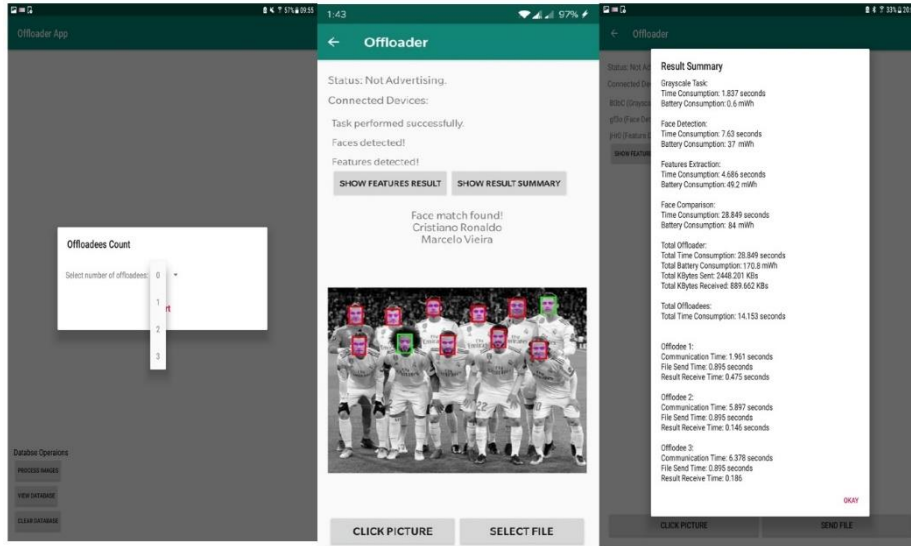
Fig. 2. Screenshots of FDSC

We have developed a simple algorithm to distribute the images among the Offloadees and the servers. Firstly, we divide the number of images (n) equally among the total devices. After that we find the remaining number of images, if the remaining images are equal to 0, then the algorithm start distributing the images. If the remaining images are > 0 then we distribute the remaining images one by one to the Offloadees. (For example, if the number of connected devices = 4, number of images = 10, then 10/4, so initially each device gets 2 images, then for the remaining 2 images, it assigns one by one to the devices, so offloader = 2, offloadee1 = 3, offloadee2 = 3, offloadee3 = 2 and so on).

We use a third-party tool (AWS rekognition service) that uses storage-based API operations to create the DB that we need later to compare with the Offloader new images. It gets the images from FDSC local repository root, then we call Detectface request, callFaceDetails, and Detectfeatures functions to build a client-side index.

4.2 Experimental Scenarios

In this section, the various scenarios for the experiments that has been done to illustrate the overhead of forming the edge resource is described. The aim of these scenarios is to examine the benefit of SOSE when offloading in terms of computation time, energy consumption and wireless connectivity costs when FDS & FDSC sub-tasks are executed by various devices together with the Offloader. These scenarios are referred as Edge Server Scenario (ESS), Edge Offloadees Scenario (EOS) and Cloud Server Scenario (CSS) in this paper, as follows:

4.2.1 The Offloader sends (FDS & FDSC) sub-tasks to a local edge server (ESS)

In this scenario, we have created a WAMPSERVER 3.1.0, which acts as a local nearby edge server. Both Offloader and server are connected through an IP address. If the decision is to run the tasks on ESS, the decision engine triggers the distribution algorithm to partition the images between the Offloader and ESS. The Offloader generates a serializable interface and decides on the class name, method signature, serialized object, and required libraries to be offloaded. Then it invokes the remote manager, to connect to the server using IP address and post API and offload the images in parallel. The edge server waits and listen to any incoming tasks, it runs the requested tasks when receive the images, record the time using timestamps, convert it to JSON format, and send the results back to the Offloader as will be stated later in section 4.3. We used BroadbandChecker tool [14] to make sure that the network is stable when offloading using Wi-Fi with 334 Mbps downlink, 260 Mbps uplink and 43 ms latency.

4.2.2 The Offloader sends (FDS & FDSC) sub-tasks to nearby edge Offloadees (EOS)

In this scenario, we perform offloading to cooperative nearby edge-devices on the go. We have used one Offloader end-device and a maximum number of 3 end-Offloadee devices, full specification of the conducted devices used are shown in Table 1. All the devices are connecting through nearby API which is a peer-to-peer networking API that allows apps to connect, share, and exchange data with each other in order to communicate over a local area network. We have used nearby connections type since it offers unlimited payload to be shared and it supports sensitive data by encrypting the data for secure payload exchange. We have defined 5 classes to establish the communication between edge Offloadees, these are Start Discovery (), Start Advertising (), Endpoint Discovery Call back (), Request Connection (), and Payload Call back (). When the device is selected as an Offloadee, the Offloader starts accepting incoming connections (the number of incoming connections is equal to the number of the Offloadees). When we select more than 0 in the drop-down list, the Offloader starts advertising itself to accept incoming connection from nearby Offloadees. The Offloadees then discovers the Offloader and send a request to connect. The Offloader then accept the connection and add the incoming Offloadee to the connected devices list. Then connection is established, and devices are ready to exchange images between them.

We have developed a simple algorithm to distribute the images among Offloadees explained in section 4.1. (for example, if we have 20 images to run, then each device executes 5 images in parallel and perform the required tasks for the images, then send the results back to the Offloader). The Offloadees waits and listens for any incoming tasks, when it receives the images, it runs the grayscale, face detection and feature extraction tasks, record the time using timestamps and send the results back to the Offloader device. A total of 100 images to perform offloading between a variety of

edge end-devices are used. The images are set to have the same resolution (700X700) and have a maximum size of 300 KB, and tests are repeated 5 times to examine stable and unstable network when offloading. The results are calculated (an average of 5 runs) in terms of computation time, energy consumption, communication saving, and offloading gain as illustrated in section 4.3.

Table 1. Experimental (Offloader & Offloadees) specifications for EOS

Devices specification	CPU	RAM	OS	Battery
Samsung S2 Sm-T710	1.3 GHz	3 GB	Android 7.0	4000 mAh
Lenovo TB-7304F tablet	1.3 GHz	1 GB	Android 7.0	3500 mAh
LG Nexus 4	1.5 GHz	2 GB	Android 5.1.1	2100 mAh
LG Nexus 4	1.5 GHz	2 GB	Android 5.1.1	2100 mAh

4.2.3 The Offloader sends (FDS & FDSC) sub-tasks to a cloud server (CSS)

In this scenario, the workload computation of FDSC is offloaded from the Offloader to a cloud server. We have created a server in the cloud using Amazon AWS services, namely t2.micro Amazon Linux 2 AMI EC2 server. We have created the credentials (secret, access, and IAM keys) to authenticate the server with (FDS & FDSC), so it can connect and push images to the cloud server. We have also used FileZilla and Putty tools to install and migrate the necessary PHP files to the server. We created a S3 bucket to save the offloaded images if needed for future execution and/or to train SOSE_INTELLIGENT engine. If the decision is to run the sub-tasks on the server, the Offloader connects to the server and starts to offload the images through an IP address and POST API. The server waits and listen to any incoming tasks, it runs the requested tasks when receive the images, record the time using timestamps, convert it to JSON format, and send the results back to the Offloader as will be stated later in section 4.3.

4.3 Results & Discussion

This section presents all the results achieved from the conducted various experiments for the scenarios we designed to illustrate the concept of this SOSE solution.
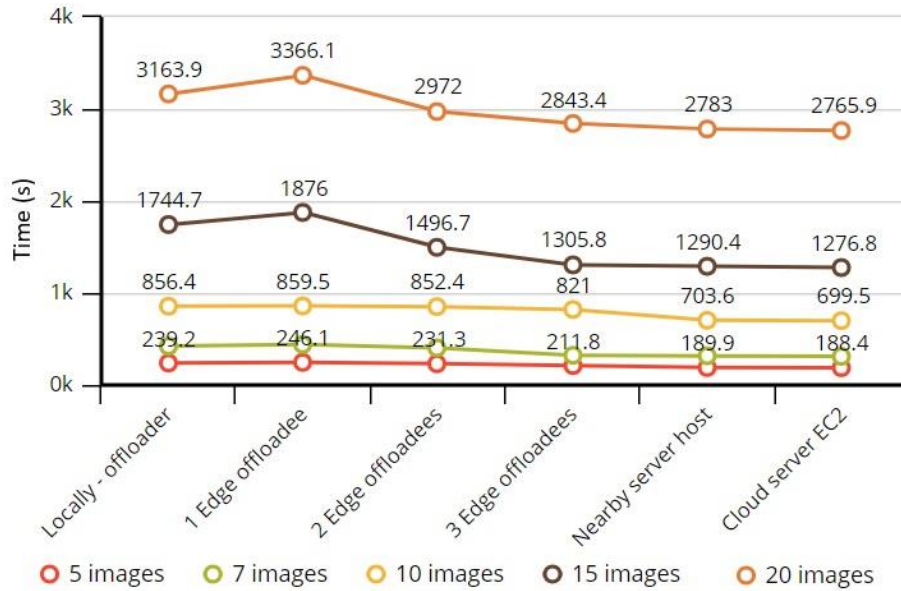Figure 3 shows the computation time of executing FDS for ESS, EOS and CSS we described in section 4.2. Offloading to ESS and CSS has reduced the burden on the Offloader by 83.4% due to their unlimited resource capability. Note that the results are testimony that having an edge server is the correct decision since it will be less overhead when communication traffic is taken into consideration. It also clear that offloading to a single Offloadee is too costly with an increase of the task by 14.3% due to the overhead not meeting the crossover point of being advantageous. However,

offloading to >1 Offloadee has significantly improved the Offloader resource capability (21.3% & 40.2% for 2 & 3 Offloadees respectively).

Fig. 3. Computation time of FDS

Figure 4 shows the computation time when running FDSC for ESS, EOS and CSS. it shows an increase of the complexity of the FDC, by adding more intensive tasks, such as matching the extracted features with a DB. This highlights the importance of SOSE, where the computation time became liner for all ESS, EOS and CSS. This means that the overall cost of SOSE is much less than having the offloading done to the cloud, without the network traffic caused by transporting the data to the cloud. For 20 images with 4 edge end-devices, we achieved 10.13% in comparison to running the tasks locally, while 12.1% for the cloud scenario, which indicates the SOSE will outperform offloading to the cloud solution when complex tasks are executed on more participated edge end-devices.

Fig. 4. Computation time of FDSC

The power consumption measured when executing FDSC for ESS, EOS, and CSS are shown in Figure 5, it clearly shows that same saving pattern is achieved with computation complexity. The behavioral trend we observed is, when only 2 nearby Offloadees are executing the FDSC, the battery consumption cost increased by 19.52%. However, when the number of Offloadees increases in EOS, we record a power saving of 28.8% for 4 Offloadees running FDSC in parallel, which is almost similar with ESS and CSS which record 31.8% power saving.

Figure 6 shows the computation time of FDS sub-tasks for ESS, EOS, and CSS, it shows that the feature extraction task is the most intensive task compared to other tasks. also, it shows, the computation time dropped down continuously, almost up to 81.2% saving when more Offloadees run FDS.
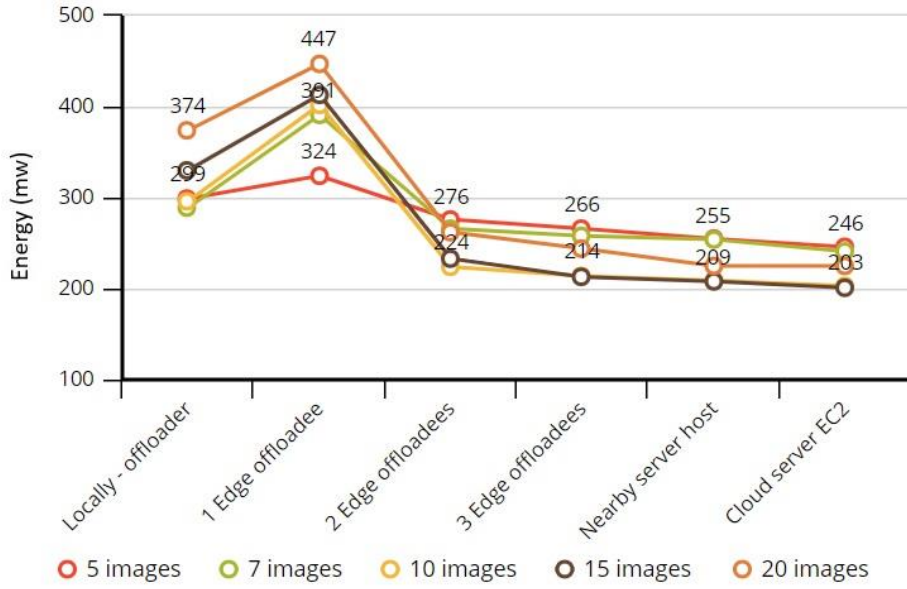
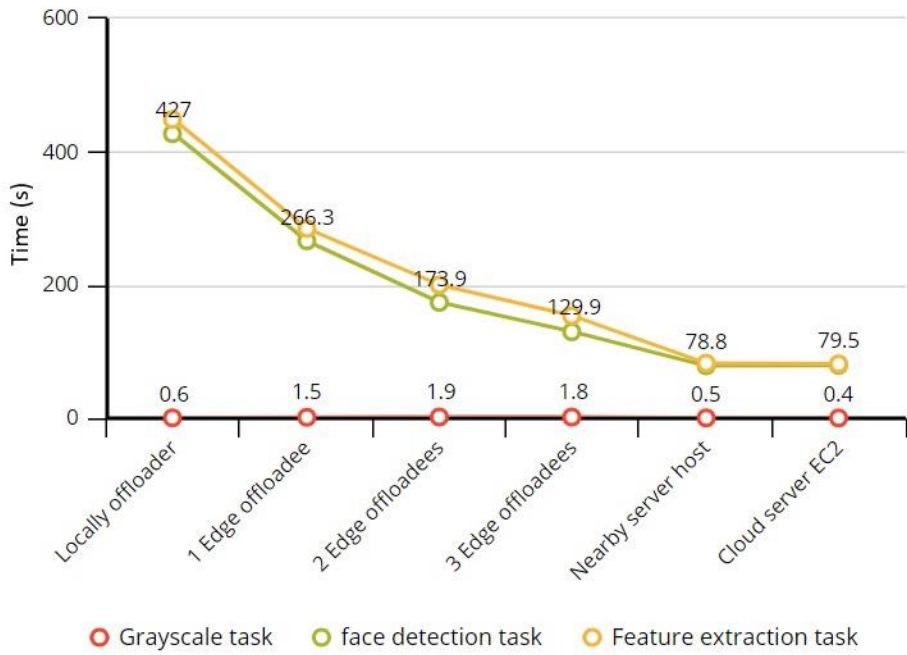**Fig. 5.** Battery consumption of FDSC

Fig. 6. Computation time of FDS sub-tasks

5   Conclusion and future work

The discussion and analysis of the experiments in the above section concludes that we can form a network of Offloadees on the go as needed, that will, even small number of devices of 4 Offloadees will perform as good as an edge computing server with unlimited resources. Our future study on this thread will focus on the granularity and partition of the sub-tasks so to maximize the benefit from the Offloadees without having to run their battery to the ground or increasing the local connectivity traffic with them. For sure having only a single Offloadee to help with the MCCS is not an option.

The impact of connectivity between our local edge resource network and the cloud is significant and depends on the location of the Offloader. For example, if the Cloud server is only accessible by cellular link, then the overheads will be 10x more than if a Wi-Fi connectivity is available to the server. This will give much more importance to our SOSE as we can form P2P connectivity with all Offloadees, including using a Wi-Fi P2P link.

For automating all the decisions on the offload or not, sub-tasks sizes, Offloadee choices and so on, an intelligent engine is very important to achieve efficient offloading. The various variations for the type of intelligence algorithms is the next study phase of this project.

6   Acknowledgment

References

1. Ali Al-ameri and Ihsan Alshahib Lami, "SCCOF: Smart Cooperative Computation Offloading Framework for Mobile Cloud Computing Services,", in the 8th Annual International Conference: Big Data, Cloud and Security, 2017.
2. S. M. Saad and S. C. Nandedkar, "Energy Efficient Mobile Cloud Computing," 2014.
3. Elmannai, Wafa, and Khaled Elleithy. "Sensor-based assistive devices for visually-impaired people: current status, challenges, and future directions." Sensors 17.3 (2017): 565.
4. Dwivedi, Akhilesh, et al. "Internet of Things'(IoT's) Impact on Decision Oriented Applications of Big Data Sentiment Analysis." 2018 3rd International Conference on Internet of Things: Smart Innovation and Usages (IoT-SIU). IEEE, 2018.
5. Wei, Xiaojuan and Wang, Shangguang and Zhou, Ao and Xu, Jinliang and Su, Sen and Kumar, Sathish and Yang, Fangchun, "MVR: An architecture for computation offloading in mobile edge computing,", in the IEEE International Conference on Edge Computing, 2017.
6. Calo, Seraphin B., et al. "Edge computing architecture for applying AI to IoT." 2017 IEEE International Conference on Big Data (Big Data). IEEE, 2017.

7. "Amazon Rekognition: Developer Guide" [Online]. Available: http://docs.aws.amazon.com/rekognition/latest/dg/rekognition. [Accessed January 2019].

8. Chen, Xu, et al. "Thriftyedge: Resource-efficient edge computing for intelligent IoT applications." IEEE network 32.1 (2018): 61-65.

9. Li, He, Kaoru Ota, and Mianxiong Dong. "Learning IoT in edge: deep learning for the internet of things with edge computing." IEEE Network 32.1 (2018): 96-101.

10. Ko, Kwangman, et al. "DisCO: A distributed and concurrent offloading framework for mobile edge cloud computing." 2017 Ninth International Conference on Ubiquitous and Future Networks (ICUFN). IEEE, 2017.

11. "Nearby Connections API" [Online]. Available: https://developers.google.com/nearby/connections/android/exchange-data. [Accessed July 2018].

12. S. Tarkoma, M. Siekkinen, E. Lagerspetz and Y. Xiao, "Smartphone energy consumption: modeling and optimization", Cambridge University Press, 2014.

13. Sirivianos, Michael, et al. "Dandelion: Cooperative Content Distribution with Robust Incentives." USENIX Annual Technical Conference. Vol. 7. 2007.

14. "BroadbandChecker," [Online]. Available: http://www.broadbandspeedchecker.co.uk. [Accessed November 2017].