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
{ali.al-ameri, ihsan.lami}@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), so to achieve minimum latency and improve performance while reducing battery power 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 sub-tasks, 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 sub-tasks to the target edge devices in parallel and retrieve results. Scenarios and experiments to evaluate SOSE show that a significant improvement, in terms of processing time (>40%) and battery power consumption (>28%), has been achieved when compared with cloud offloading solutions.

Keywords: Offloading · Edge computing · Cooperative · Mobile cloud computing

1 Introduction

The Smart Phone (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 ship the processing of such MCCS to a central server in the cloud. This 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 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 the sub-tasks for computation, from the MCCS host device/SP 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 of the best scenario to partition and offload, to achieve a low processing time and reduce the battery power 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 sub-tasks 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.

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 of SOSE_EDGE is done intelligently by an SOSE_INTELLIGENT engine based in the cloud. SOSE_INTELLIGENT engine recruits cooperative device resources, monitors (processing capability, battery status, and availability), and authenticates (access key, session key and engagement status), so to advice on available device nearby when the Offloader needs to form the SOSE_EDGE.

The rest of this paper includes: Sect. 2 that summarizes the recent literature on edge computing implementations, while Sect. 3 presents the development of SOSE. Section 4 presents the experiments, results and analysis. Finally, Sect. 5 presents the conclusion and future work.

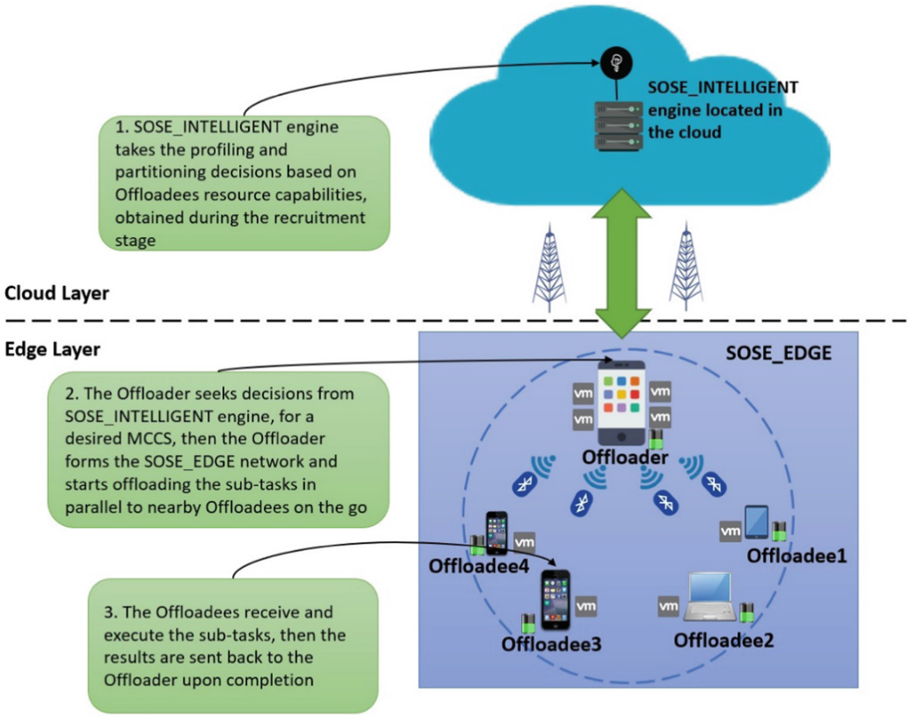

1. SOSE_INTELLIGENT engine takes the profiling and partitioning decisions based on Offloadees resource capabilities, obtained during the recruitment stage
SOSE_INTELLIGENT engine located in the cloud
Cloud Layer
Edge Layer
2. The Offloader seeks decisions from SOSE_INTELLIGENT engine, for a desired MCCS, then the Offloader forms the SOSE_EDGE network and starts offloading the sub-tasks in parallel to nearby Offloadees on the go
SOSE_EDGE
vm
Offloader
Offloadee1
Offloadee4
Offloadee3
Offloadee2
3. The Offloadees receive and execute the sub-tasks, then the results are sent back to the Offloader upon completion

Fig. 1. SOSE scheme

2 Recent Literature of Edge Computing Implementations

Review of solutions that perform offloading to a centralised server in the cloud has been published in our previous paper [1]. This review focuses on implementations/ solutions, that consider IoT intensive applications, which offload to nearby pre-setup infrastructure of edge servers. SOSE proposes to deploy the SOSE_EDGE solution on-the-go when needed. This is achieved by recruiting a group of available nearby processing resources/devices 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 focus on offloading between the edge servers and the cloud. Running these 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/devices, so to reduce long execution time and battery power consumption. A solution that facilitates offloading the service from a SP to an edge computing server, has introduced a model that provides the use of virtual resources in edge servers [5]. It achieves this, by shifting the service sub-tasks from a SP to the edge server automatically, by dividing a single task to 5 sub-tasks, using 0–1 integer liner programming. It marks the sub-tasks with a value of (0, 1), where “0” stands for sub-tasks to run locally on the SP, such sub-tasks that access SP local features or input and output tasks. Similarly, “1” stands for sub-tasks to run on an edge server, which has multiple virtual resources to handle the execution of the sub-tasks. 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 in 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 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 balances the computational workload between the 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-device will upload data, which are then labelled and tested by multiple 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, SOSE uses 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 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. (3) Finally, 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 believe other metrics like, battery power consumption, communication and efficiency must be considered in the evaluation. We deploy a similar concept to execute the sub-tasks in parallel on the nearby edge devices, so faster execution time can be achieved.

Some IoT apps required deep learning algorithms to extract accurate information for classification, especially for IoT devices deployed in complex environments. Nevertheless, such algorithms require a significant amount of processing, (i.e. each deep learning extra layer can bring extra processing among its multilayer structure). Therefore, Efficient scheduling mechanisms are needed to decide on how many layers can run on the edge servers. A solution that facilities offloading to optimize the performance of deep learning for IoT at the edge, has introduced a model that provides offline and online scheduling mechanism [9]. It achieves this by, monitoring 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 such layers in the cloud server. Then, when the dimension of the deep learning 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 deep learning 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 resources from end-devices and schedule the sub-tasks among them. I.e. SOSE_INTELLIGENT engine schedules the sub-tasks 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 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 sub-tasks among a variety of nearby edge devices, so the shared data cannot be retrieved or invoked as a package, and so stealing the sub-task will not impact the overall security of 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

This engine 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 running the MCCS. A suitable arrangement for controlling this will need to be in place as in [12], 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 to 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 decisions. 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 (named the Offloader here). Any participating device in helping to run the sub-tasks are named 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 sub-tasks, based on the instructions provided by the SOSE_INTELLIGENT engine. Note that, the choice of having the profiling and partitioning 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 when it is not busy with the other sub-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 and Analysis

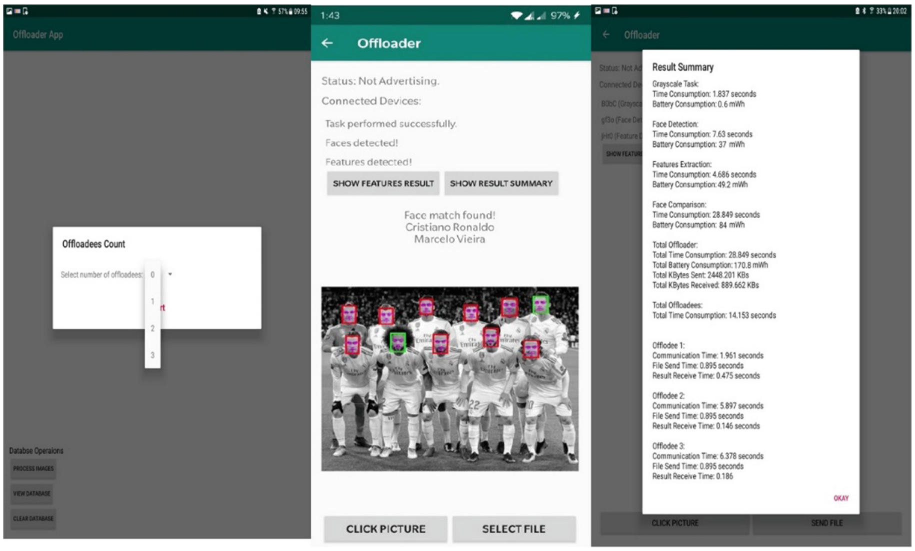
The following experiment scenarios is used to prove that SOSE_EDGE can provide an on-the-go (dynamic) edge resource from available nearby devices, and will perform as good as, or better than, a structured pre-setup edge computing server. The details of the implementation of SOSE_INTELLIGENT engine and 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. We developed FDS using Android studio platform and Dlib library, which is an open source library for image detection and recognition. It obtains a face bounding box using coordinators of the face in the image. Then it detects and draw 68 coordinators in the face, and finally, it extracts 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 GUI. FDS uses mface.train function to train the algorithm to perform the face detection process. Then it uses recognizeAsync function to execute the algorithm. Full details about the specification and experimental devices are illustrated in Sect. 4.2.

To illustrate more sub-tasks, we developed a complex version of FDS, we named it; FDSC. This includes recognition functions. As shown in Fig. 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 used up to 4 devices 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 tasks 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 tasks 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.

We 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



Status: Not Advertising.
Connected Devices:
Task performed successfully.
Faces detected!
Features detected!

Face match found!
Cristiano Ronaldo
Marcelo Vieira

Result Summary
Grayscale Task:
Time Consumption: 1.837 seconds
Battery Consumption: 0.6 mWh

Face Detection:
Time Consumption: 7.63 seconds
Battery Consumption: 37 mWh

Features Extraction:
Time Consumption: 4.686 seconds
Battery Consumption: 49.2 mWh

Face Comparison:
Time Consumption: 28.849 seconds
Battery Consumption: 84 mWh

Total Offloader:
Total Time Consumption: 28.849 seconds
Total Battery Consumption: 170.8 mWh
Total KBytes Sent: 2448.201 KBs
Total KBytes Received: 889.662 KBs

Total Offloadees:
Total Time Consumption: 14.153 seconds

Offlodee 1:
Communication Time: 1.961 seconds
File Send Time: 0.895 seconds
Result Receive Time: 0.475 seconds

Offlodee 2:
Communication Time: 5.897 seconds
File Send Time: 0.895 seconds
Result Receive Time: 0.146 seconds

Offlodee 3:
Communication Time: 6.378 seconds
File Send Time: 0.895 seconds
Result Receive Time: 0.186

Fig. 2. Screenshots of FDSC

are equal to 0, then the algorithm starts distributing the images. If the remaining images are > 0, then it distributes 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 used a third-party tool (AWS rekognition service) that uses storage-based API operations to create the DB, to compare with the Offloader new images. It gets the images from FDSC local repository root, then it calls 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 have been done to illustrate the overhead of forming the edge resource are described. The aim of these scenarios is to examine the benefit of SOSE when offloading in terms of processing time, and battery power consumption, when FDS & FDSC sub-tasks are executed by various devices together with the Offloader. In the experiments, we have focused on comparing the processing time of sub-tasks, as well as battery power consumption for this period, as being the quality of service parameters along with the accuracy and efficiency of SOSE. Full details about the processing time cost and battery power consumption cost, with the equations can be found in our SCCOF paper [1]. There are many equations developed that calculate offloading time as a whole, based on the above and including amount of computation and communication [14, 15]. However, we did not find a significant impact of all these parameters on the trend of the described experiments as explained in Sect. 4.3. These scenarios are referred, as Edge Server Scenario (ESS), Edge Offloadees Scenario (EOS) and Cloud Server Scenario (CSS), in this paper, as shown in Fig. 3.

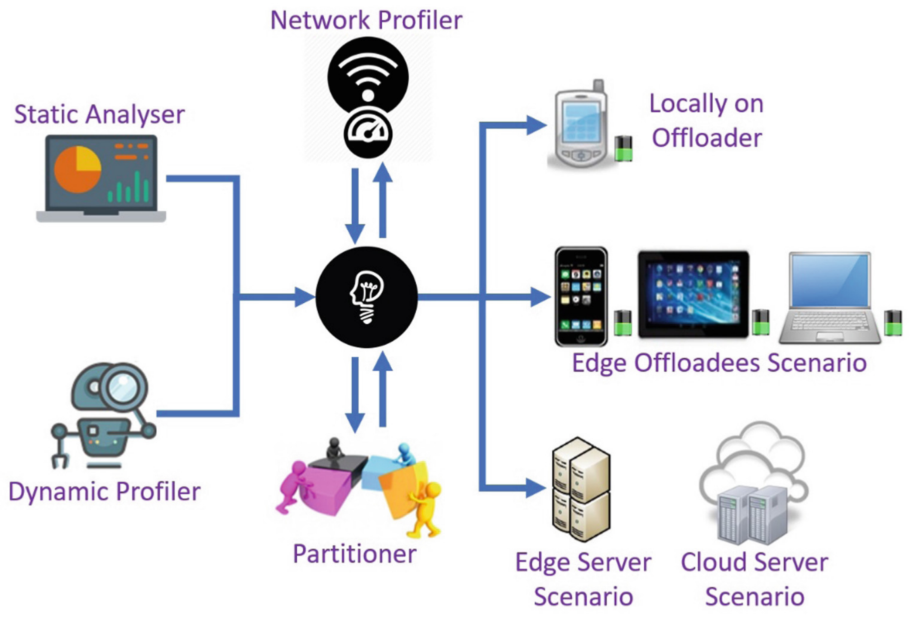


Fig. 3. SOSE architecture

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 images to be offloaded. Then it invokes the remote manager, to connect to the server using IP address and post API and offloads the images in parallel. The edge server waits and listens to any incoming tasks, it runs the requested sub-tasks when receives the images, records the time, converts it to JSON format, and sends the results back to the Offloader as will be stated later in Sect. 4.3. We used BroadbandChecker tool [16] to profile the network and make sure it is stable when offloading.

4.2.2 The Offloader Sends (FDS & FDSC) Sub-tasks to Nearby Edge Offloadees (EOS)

In this scenario, we performed offloading to cooperative nearby edge-devices on-the-go. We used one Offloader and a maximum number of 3 Offloadees, full specifications of the conducted devices are shown in Table 1. All the devices are connecting through nearby API, which is a peer2peer networking API that allows apps to connect, share, and exchange data 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 Callback (), Request Connection (), and Payload Callback (). 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 sends a request to connect. The Offloader accepts the connection and adds the incoming Offloadee to the connected devices list. Then the connection is established, and devices are ready to exchange images between them.

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

We developed a simple algorithm to distribute the images among Offloadees, as explained in Sect. 4.1. (For example, if the Offloader selects 20 images to execute, each device executes 5 images in parallel and performs the required sub-tasks, then each device sends the results back to the Offloader). The Offloadees wait and listen for any incoming tasks, run the sub-tasks, record the time and send the results back to the Offloader. A total of 100 images to perform offloading between a variety of edge devices are used. The images are set to have the same resolution (700 $\times$ 700), 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 processing time, battery power consumption, and offloading gain as illustrated in Sect. 4.3.

4.2.3 The Offloader Sends (FDS & FDSC) Sub-tasks to a Cloud Server (CSS)

In this scenario, we have created a server in the cloud using Amazon AWS services, namely t2.micro Amazon Linux 2 AMI EC2 server. We 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 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 listens to any incoming tasks, it runs the requested sub-tasks when receives the images, records the time, converts it to JSON format, and sends the results back to the Offloader as will be stated later in Sect. 4.3.

4.3 Results and 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 4 shows the processing time of executing FDS for ESS, EOS and CSS we described in Sect. 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 is also clear that offloading to a single Offloadee is costly with an increase of 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).

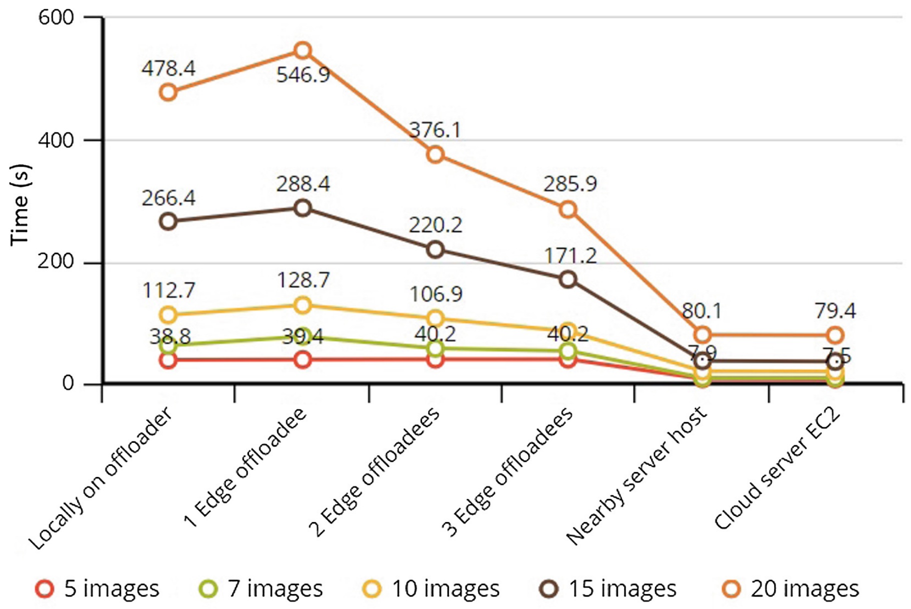


Fig. 4. Processing time of FDS

Figure 5 shows the processing time when running FDSC for ESS, EOS and CSS. It shows an increase of the complexity of the FDC, by adding more intensive sub-tasks, such as matching the extracted features with a DB. This highlights the importance of SOSE, where the processing 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 sub-tasks locally, while 12.1% for the cloud scenario, which indicates that, SOSE will outperform offloading to the cloud solution when complex sub-tasks are executed on more participated edge end-devices.

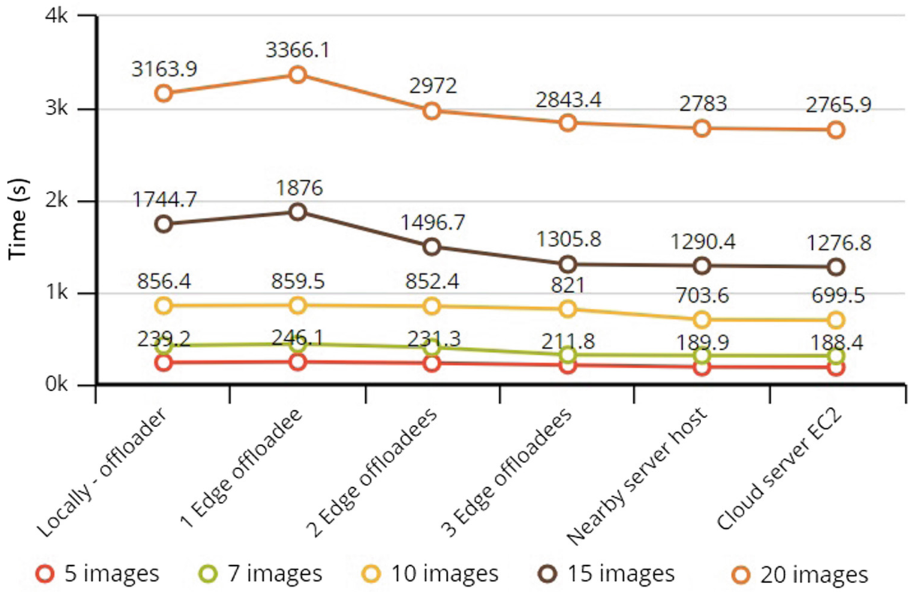


Fig. 5. Processing time of FDSC

The battery power consumption measured when executing FDSC for ESS, EOS, and CSS is shown in Fig. 6, it clearly shows that same saving pattern is achieved with processing time. The behavioral trend we observed is, when only 2 devices are executing the FDSC, the battery power 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. To the best of our knowledge, none of the reviewed solutions performed offloading of FDSC sub-tasks to nearby edge offloadees. However, to compare ESS and CSS with Thinkair [17], that performs offloading of FDS sub-tasks to a cloud server, we have achieved improvements of 12.48% and 38.4%, in terms of processing time and battery power consumption respectively.

Figure 7 shows the processing 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 processing time dropped down continuously, almost up to 81.2% saving when more Offloadees run FDS. To measure the accuracy of SOSE, we used Rekognition confidence score of similarity. The confidence score is between (0–100), that expresses the probability of the detection, if the face is predicted correctly. We achieved up to 99% accuracy as almost all the selected images are recognized successfully. We compared the accuracy rate achieved by SOSE, with the work in [18], that used facial recognition service. We achieved an increase of 23.75%.

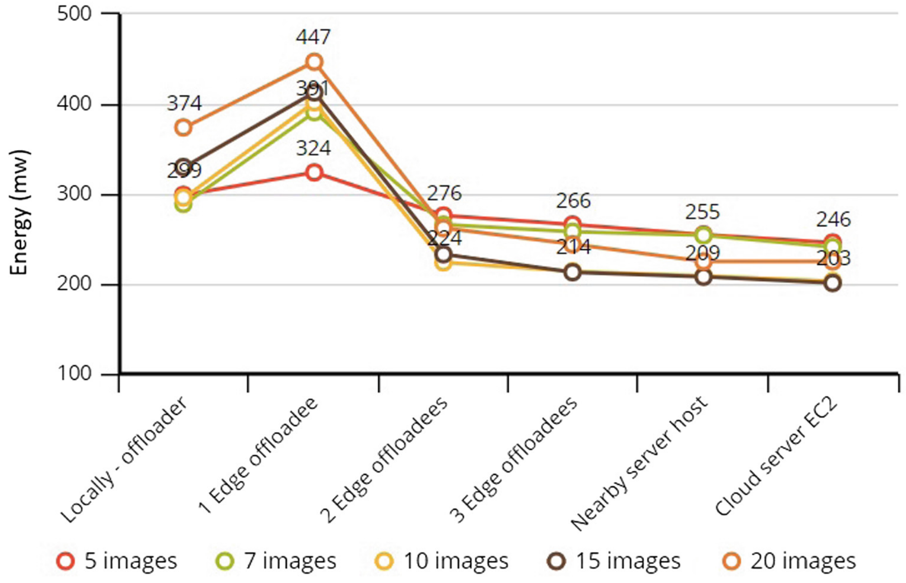


Fig. 6. Battery power consumption of FDSC

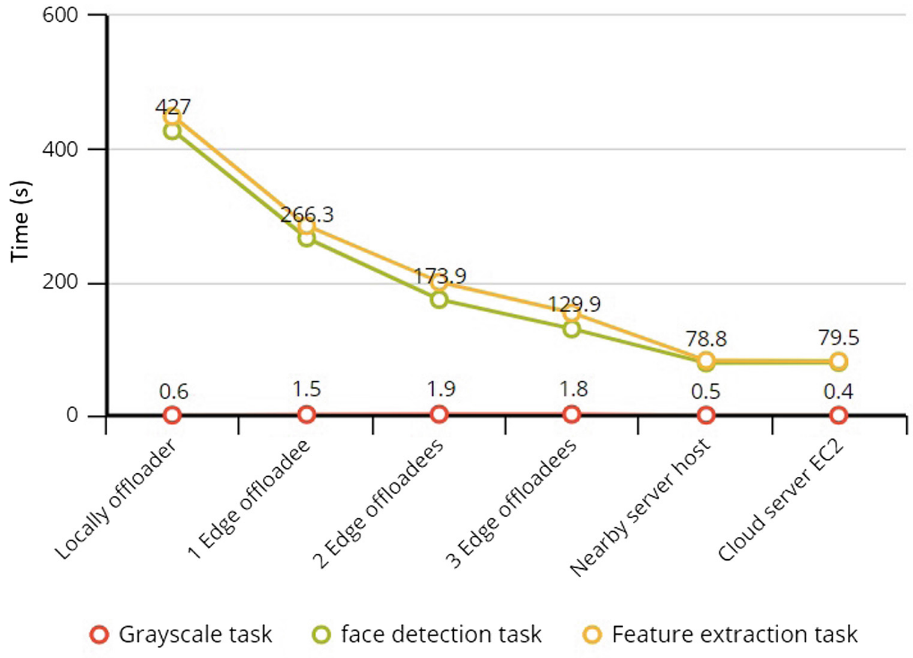


Fig. 7. Processing time of FDS sub-tasks

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 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 automatic partitioning and profiling are required instead of manual profiling. As a next step, we will implement the controller engine, all the end-devices connect to the engine, and exchange a report of features including; location, battery level, processing capability, etc. In CloudSim simulator, the broker role is to decide where to offload the service workload, based on simple broker policies such as nearest VM, fastest VM or dynamic VM. We shell deploy our engine as so, to take the broker policy in deciding where and what to offload, based on SOSE assessment criteria as described in Sect. 3. An intelligent engine is very important to achieve efficient offloading. The various variations for the type of intelligence algorithms, (such as genetic algorithm and/or Markov model) is the next study phase of this project.

Acknowledgment. Gratitude to the University of Basra, and MOHESR (Ministry of Higher Education and Scientific Research) for sponsoring this work.

References

1. Al-ameri, A., Lami, I.A.: SCCOF: smart cooperative computation offloading framework for mobile cloud computing services. In: the 8th Annual International Conference: Big Data, Cloud and Security (2017)
2. Saad, S.M., Nandedkar, S.C.: Energy efficient mobile cloud computing (2014)
3. Elmannai, W., Elleithy, K.: Sensor-based assistive devices for visually-impaired people: current status, challenges, and future directions. Sensors **17**(3), 565 (2017)
4. Dwivedi, A., et al.: Internet of Things' (IoT's) impact on decision oriented applications of big data sentiment analysis. In: 2018 3rd International Conference on Internet of Things: Smart Innovation and Usages (IoT-SIU). IEEE (2018)
5. Wei, X., et al.: MVR: an architecture for computation offloading in mobile edge computing. In: the IEEE International Conference on Edge Computing (2017)
6. Calo, S.B., et al.: Edge computing architecture for applying AI to IoT. In: 2017 IEEE International Conference on Big Data (Big Data). IEEE (2017)

7. Amazon Rekognition: Developer Guide. http://docs.aws.amazon.com/rekognition/latest/dg/rekognition. Accessed January 2019
8. Chen, X., et al.: Thriftyedge: resource-efficient edge computing for intelligent IoT applications. IEEE Netw. **32**(1), 61–65 (2018)
9. Li, H., Ota, K., Dong, M.: Learning IoT in edge: deep learning for the internet of things with edge computing. IEEE Netw. **32**(1), 96–101 (2018)
10. Ko, K., et al.: DisCO: a distributed and concurrent offloading framework for mobile edge cloud computing. In: 2017 Ninth International Conference on Ubiquitous and Future Networks (ICUFN). IEEE (2017)
11. Nearby Connections API. https://developers.google.com/nearby/connections/android/exchange-data. Accessed July 2018
12. Wang, X., Chen, X., Wu, W., An, N., Wang, L.: Cooperative application execution in mobile cloud computing: a stackelberg game approach. IEEE Commun. Lett. **20**, 946–949 (2016)
13. Sirivianos, M., et al.: Dandelion: cooperative content distribution with robust incentives. In: USENIX Annual Technical Conference, vol. 7 (2007)
14. Thu, M.S.Z., Htoon, E.C.: Cost solving model in computation offloading decision algorithm. In: 2018 IEEE 9th Annual Information Technology, Electronics and Mobile Communication Conference (IEMCON). IEEE (2018)
15. Kumar, K., Lu, Y.-H.: Cloud computing for mobile users: can offloading computation save energy? Computer **43**, 51–56 (2010)
16. BroadbandChecker. http://www.broadbandspeedchecker.co.uk. Accessed November 2017
17. Kosta, S., Aucinas, A., Hui, P., Mortier, R., Zhang, X.: ThinkAir: dynamic resource allocation and parallel execution in the cloud for mobile code offloading. In: 2012 Proceedings of the IEEE INFOCOM (2012)
18. Luzuriaga, J., et al.: Evaluating computation offloading trade-offs in mobile cloud computing: a sample application. In: Proceedings of the 4th International Conference on Cloud Computing, GRIDs, Virtualization (2013)