A Smart Edge Computing Resource, formed by On-the-go Networking of Cooperative Nearby Devices using an AI-Offloading Engine, to Solve Computationally Intensive Sub-tasks for Mobile Cloud Services

By

Ali Al-ameri

School of Computing

The University of Buckingham

United Kingdom

A Thesis Submitted for the Degree of Doctor of Philosophy in Computing

September 2020

Abstract

The latest Mobile Smart Devices (MSDs) and IoT deployments have encouraged the running of "Computation Intensive Applications/Services" onboard MSDs to help us perform on-the-go sub-tasks required by these Apps/Services such as Analysis, Banking, Navigation, Social Media, Gaming, etc. Doing this requires that the MSD have powerful processing resources to reduce execution time, high connectivity throughput to minimise latency and high-capacity battery for power consumption so to not impact the MSD availability/usability in between charges. Offloading such Apps from the host-MSD to a Cloud server does help but introduces network traffic and connectivity overhead issues, even with 5G. Offloading to an Edge server does help, but Edge servers are part of a pre-planned overall computing resource infrastructure, that is tough to predict when demands/rollout is generated by a push from the MSDs/Apps makers and pull by users. To address this issue, this research work has developed a "Smart Edge Computing Resource", formed on-the-go by the networking of cooperative MSDs/Servers in the vicinity of the host-MSD that is running the computing-intensive App. This solution is achieved by:

Developing an intelligent engine, hosted in the Cloud, for profiling "computing-intensive Apps/Services" for appropriately partitioning the overall task into suitable sub-task-chunks so to be executed on the host-MSD together/in association with other available nearby computing resources. Nearby resources can include other MSDs, PCs, iPads and local servers. This is achieved by implementing an "Edge-side Computing Resource engine" that intelligently divides the processing of Apps/Services among several MSDs in parallel. Also, a second "Cloud-side AI-engine" to recruit any available cooperative MSDs and provide the host-MSD with decisions of the best scenario to partition and offload the overall App/Services. It uses a performance scoring algorithm to schedule the sub-tasks to execute on the assisting resource device that has a powerful processor and high-capacity battery power. We built a dataset of 600 scenarios to boost up the offloading decision for further executions, using a Deep Neural Network model.

Dynamically forming the on-the-go resource network between the chosen assisting resource devices and the App/Service host-MSD based on the best wireless connectivity possible between them. This is achieved by developing an Importance Priority Weighting cost estimator to calculate the overhead cost and efficiency gain of processing the sub-tasks on the available assisting devices. A local peer-to-peer connectivity protocol is used to communicate, using "Nearby API and/or Post API". Sub-tasks are offloaded and processed among the participating devices in parallel while results are retrieved upon completion.

The results show that our solution has achieved, on average, 40.2% more efficient processing time, 28.8% less battery power consumption and 33% less latency than other methods of executing the same Apps/Services.

Acknowledgement

In the name of “Almighty Allah”, the most powerful and merciful who gave me the power and ability to complete my research successfully.

This research would have never been accomplished without advice, support and love from many kind people, who I would like to acknowledge here.

My grateful thanks and deepest appreciation to my supervisor, Professor Ihsan Lami, who contributed greatly to this thesis in a methodical approach, using the correct hardware and software tools to achieve a structured outcome in a timely manner. His direct engagement, brainstorming discussion, valuable comments, and criticism have enormously helped me to accomplish my research. Without his guidance, patience and wise advice, this thesis would have never come to light. He has been a valuable friend and mentor, I could not have asked for better supervision, Thank you.

I dedicate this work to my father, Professor Abdulhussein Alameri who has devoted his entire life motivating my siblings and I. His accomplishments and success story were my inspiration all through my life. I would like to express my gratitude to my mother, sisters and brother for their endless support and prayers, that was my pillars to achieve all through my life and to accomplish this thesis.

My genuine appreciation goes to my lovely wife for being here for me at all times! Thank you for your support, patience and encouragement, and for bearing with me for many lost weekends/nights supporting me throughout. To my beloved children, Yazan & Lara, who always put a smile on my face that made the passing of time more colourful and fun.

I delightfully acknowledge and appreciate MOHESR for sponsoring this thesis and financially supporting my research.

My sincere gratitude to all staff and colleagues in the School of Computing at The University of Buckingham who have provided a wonderful and fun environment during my research. A special thanks to all my friends who have backed me at all times, thank you for your endless support and continuous stimulus.

Motivation

After graduation in 2012, I started to work as an IT Supervisor at "Samsung Engineering Limited Company" for oil and drilling services, which was based in West Qurna-2 in Basra province south of Iraq. This company was sub-contracted by "Lukoil", (the second largest gas & energy company in Russia, who was contracted by the Iraqi government), to build an oilfield with 1-million barrels per day. One night, while I was watching the news, I came across a report with a title of "DUBAI EXPO 2020". I remember it was a great announcement, I was very keen to know the details. I quote here, the reporter said: "This event will be mainly focusing on Cloud Computing (CC), it is a great opportunity for Dubai to use and offer Cloud services to people". I was very interested to understand and read more about the topic. One of my roommates said to switch the TV off and let go to the gym, I said I will do a PhD in CC and I will be there, (I meant in Dubai Expo 2020). We both laughed and then we went to the gym. I started to read more about CC whenever I have some spare time. I was very motivated dreaming if one day I will have the chance to do my PhD.

Then in 2013, the "Iraqi Ministry of Education" started to grant the outstanding master students an opportunity for a PhD scholarship. Immediately and without hesitation, I started to prepare the required documents. Part of the necessary arrangements was to visit the "College of Science at the University of Basra", to find out about the state-of-the-art topics they need to fulfil the PhD scholarship. I was interviewed by Dr Ali Fadhil, who asked me what subject/topic are you going to research. Without hesitance, I said CC, I remember he smiled and said do you know we do not have many CC specialists in the college at that time! he happily signed off my application/proposal. This has motivated me even more to accomplish my goal.

Later on, in 2015, when I came to the "University of Buckingham", I met my supervisor "professor Lami", who invited me to his CC lecturing classes. I started to read and understand more about CC characteristics, services, models, providers, and other stockholders. This experience enriches my knowledge and helps me to come across my research area of interest; Offloading Techniques for Mobile Cloud Computing Services (MCCS).

After reading a dozen of Offloading frameworks, I concluded that the battery power consumption is the biggest concern for SmartPhone (SP) subscribers. On discussion with my supervisor and reading the literature, I started to ask focused questions of "how to enhance the battery capacity", "sleep mode", "switching GPS and Wi-Fi off when unneeded", all these issues did not fulfil SP subscribers. This has led to some further questions; "what do I want to achieve", "can I come up with a unique on-the-go offloading solution", "can I introduce something intelligent to save the battery power". After good thinking, I got very excited to feel that I might come up with a solution that helps SP subscribers to fully capitalise on using their devices without running out of battery juice very quickly.

In 2016, we have concluded that my research will be focusing on:

1. Sort out a dynamic solution to network among a group of nearby devices, (e.g. PCs, iPads, SPs etc.), that are willing to offer their unused processing resources, in a parallel/sharing environment.
2. Everything has to be on-the-go, without the need for building infrastructure to plan a migration strategy during the development stage, that might not be suitable during the deployment stage.
3. The concept of cooperative contribution by other devices became more mature once Google started asking for this in their T&C's of their Apps.
4. Focusing on the SP as the main device to use among Mobile Smart Devices, since it is more portable, exceeding desktop PCs and laptops by far.
5. Focusing on MCCS, since these services are offered as Apps/Services to be downloaded onboard SP and billions of users are accessing these every day.
6. The concept of using the processing of nearby devices rather than a Cloud server became more mature since Cisco introduced "Fog Computing".
7. With the roll-out of AI platforms and the need for my tasks, I have decided to include a Cloud-based AI-engine that can form intelligent decisions and predictions.

Achieved Publications

From the outset, it was our aim to ensure that all the novelties of this research are published in highly-ranked events that will enable me to network with like-minded scientists, so to share ideas and produce proposals that make a difference. It is our target to conclude the whole thesis work by publishing a journal paper in the "Advanced Technologies for Multi-access Edge Computing", which will take place in Autumn 2020.

Thus far, my main novelties have been published as follows:

- Al-ameri A and Lami I, "SCCOF: Smart Cooperative Computation Offloading Framework for Mobile Cloud Computing Services", the 8th Annual International Conference on Big Data, Cloud and Security (ICT-BDCS 2017), 10.5176/2251-2136_ICT-BDCS17.04.
- Al-ameri A, Lami I, "SOSE: Smart Offloading Scheme Using Computing Resources of Nearby Wireless Devices for Edge Computing Services", the Emerging Technologies in Computing, iCETiC 2019, LNICST 285, pp. 1-15, Springer 2019. (Has been selected as the Best Paper Award).
- Lami I, and Al-ameri A, "DEO: A Smart Dynamic Edge Offloading Scheme using Processing Resources of Nearby Wireless Devices to Form an Edge Computing Engine", the 3rd IEEE International Conference on Cloud and Fog Computing Technologies and Applications (IEEE Cloud Summit 2019), 978-1-7281-3101-6/19, 10.1109/CloudSummit47114.2019.00016, IEEE 2020.
- Al-ameri A, Lami I, "A Scheduling Mechanism for Edge Computing Resource using AI Offloading Engine", Journal of Cloud Computing, Advances, Systems and Applications, Springer 2020. (to be submitted).

Research Training Record & Personal Achievements

I was so proud of the training and personal development obtained during my research that I have decided to include my log of these activities. I was part of a great team in the School of Computing at The University of Buckingham, who contributed greatly to various activities, including teaching, seminars and sports. I have attended and participated in various academic and industrial conferences, workshops and seminars, to engage with like-minded researchers/developers and to expand my skills. I have organised these activities in various categories, as detailed below.

1. Conferences Participation

- BigML Virtual Conference, “Building DEEPNET Models for Prediction Solving”, Seville, Spain, (Mar 2020). (Awarded a Certificate).
- AWS Training Conference, “Build, Train and Deploy ML Models with any Framework”, London, United Kingdom, (Nov 2019). (Awarded a Certificate).
- 3rd IEEE International Conference on Cloud and Fog Computing Technologies and Applications (IEEE Cloud Summit), The Catholic University of America, Washington DC, United States, (Aug 2019). (Awarded a Certificate).
- Emerging Technologies in Computing, iCETiC Springer, London Metropolitan University, London, United Kingdom, (Aug 2019). (Awarded a Certificate and Selected as “Best Paper Award”).
- Member of the IEEE Community, (since Jul 2019). (Awarded a Membership Certificate).
- AWS Online Conference, “AI, IoT, VM Workload and Cloud Security”, (Jul 2019). (Awarded a Certificate).
- Member of the International Association of Educators and Researchers (IAER), (since Apr 2019). (Awarded a Membership Certificate).
- 8th International Conference on Bigdata, Cloud and Security (ICT-BDCS), Singapore, (Aug 2017). (Awarded a Certificate).

- Training School on Electromagnetics for the IoE Internet of Everything, University of Bologna, Bologna, Italy, (Apr 2016).

2. Teaching Experience

- Teaching Cloud Computing Practical Sessions, School of Computing, The University of Buckingham, Buckingham, United Kingdom, (Sep-Dec 2019).
- Teaching Cloud Computing Practical Sessions, School of Computing, The University of Buckingham, Buckingham, United Kingdom, (Sep-Dec 2018).
- Teaching Cloud Computing Practical Sessions, School of Computing, The University of Buckingham, Buckingham, United Kingdom, (Sep-Dec 2017).

3. Seminars Participation

- Training Seminar, “Viva Preparation”, School of Computing, The University of Buckingham, Buckingham, United Kingdom, (Feb 2020).
- 2nd IEEE Postgraduate STEM Research Symposium, “On-going Emerging Technologies”, The University of Leeds, Leeds, United Kingdom, (Oct 2019).
- IEEE Seminar, “How to Select Proper Papers and Highly-Ranked Conferences”, The University of Buckingham, United Kingdom, (Mar 2019). (Awarded a Certificate).
- Cloud Online Seminar, “Cyber Security, Bigdata, IoT and AI Tech”, London, United Kingdom, (Mar 2019).
- Birth of Artificial Intelligence Seminar, “AI Algorithms, Social Machines and Data Deluge”, University of Oxford, Oxford, United Kingdom, (Jul 2017).
- GoogleCloud Seminar, “Emerging Cloud Services, and IoT Apps”, School of Computing, The University of Buckingham, Buckingham, United Kingdom, (Apr 2017).
- Revolution of Artificial Intelligence and Future Seminar, Bedford Centre, Bedford, United Kingdom, (Sep 2016).

4. Workshops Participation

- Knowledge Transfer Network Workshop, “Implementing AI: Running AI at the Edge”, (Jun 2020).
- Online Training Course, “IT Management, Software and Databases”, (Apr 2020). (Awarded a Certificate).
- Online Training Course, “Introduction to Mobile and Cloud Computing”, (Mar 2020). (Awarded a Certificate).
- IEEE Training Course, “Designing Security Solutions for Edge, Cloud and IoT, (Oct 2019). (Awarded a Certificate).
- AI and ML Training Workshop, Institution of Coding, Kents Hill Conference Centre, Open University, Milton Keynes, United Kingdom, (Jul 2019).
- MATLAB & Simulink Expo Workshop, Silverstone, Northampton, United Kingdom, (Oct 2017).
- Introduction to AWS Workshop, London, United Kingdom, (Sep 2017). (Awarded a Certificate).
- MATLAB & Simulink Expo Workshop, Silverstone, Northampton, United Kingdom, (Oct 2016).
- MATLAB & Simulink Expo Workshop, Silverstone, Northampton, United Kingdom, (Oct 2015).

5. Webinars Participation

- Cloud Europe Virtual Webinar, “Simplify your Multi Cloud and Hybrid Cloud”, London, United Kingdom, (Jun 2020).
- GoogleCloud Tech Talk, “Productionising AI in Manufacturing”, London, United Kingdom, (Apr 2020).
- IEEE Tech Talk, “Daily Challenges of 4G, 5G and IoT Technologies”, (Apr 2020).
- IEEE Webinar, “Techniques and End-to-End System of 5G Network”, (Mar 2020).
- AWS Cloud Expert Webinar, “Building and Securing Cloud Applications”, (Mar 2020). (Awarded a Certificate).
- AWS Online Webinar, “Data Management using PaaS”, (Dec 2019). (Awarded a Certificate).

- AWS Summit Online Webinar, “The Concept of EC2, S3 and Amazon Beanstalk”, (Dec 2019). (Awarded a Certificate).
- AWS Security Online Webinar, “The Key to Effective Cloud Encryption”, (Sep 2019). (Awarded a Certificate).
- AWS Summit Online Webinar, “The Concept of EC2, S3 and Amazon Beanstalk”, (Dec 2018). (Awarded a Certificate).

Abbreviations

AI	Artificial Intelligence
AMI	Amazon Machine Image
AP	Access Point
API	Application Programming Interface
APK	Android Package Kit
Apps	Applications
AWS	Amazon Web Services
BT	BlueTooth
CC	Cloud Computing
CIASs	Computation Intensive Applications and Services
CPU	Central Processing Unit
CS	Cloud Server
CSS	Cloud Server Scenario
DAE	Deep Auto Encoder
DB	DataBase
DBN	Deep Belief Network
DC	Data Centre
DEO	Dynamic Edge Offloading
DFS	Depth First Search
DL	Deep Learning
DNN	Deep Nerul Network

DRL	Deep Reinforcement Learning
EC	Edge Computing
EC2	Elastic Computer Cloud
ECR	Edge Computing Resource
EOS	Edge Offloadees Scenario
ES	Edge Server
ESS	Edge Server Scenario
FD	Face Detection
FD-R	Face Detection and Recognition
GA	Genetic Algorithm
GB	GigaByte
GH	GigaHertz
GUI	Graphical User Interface
IAM	Identity and Access Management
IDE	Integrated Development Environment
IoE	Internet of Everything
IoT	Internet of Things
IP	Internet Protocol
IPW	Importance Priority Weighting
JAR	Java ARchive
JSON	JavaScript Object Notation
KB	KiloByte
MB	MegaByte

MCCS	Mobile Cloud Computing Services
MCS	Mobile Cloud Services
MDP	Markov Decision Process
MH	MegaHertz
MIPS	Million Instructions Per Second
ML	Machine Learning
MSDs	Mobile Smart Devices
NDK	Native Development Kit
NN	Neural Network
OPI	Offloading Programming Interface
OS	Operating System
PC	Personal Computer
RAM	Random Access Memory
RBM	Restricted Boltzmann Machine
RL	Reinforcement Learning
RTT	Round Trip Time
S3	Simple Storage Service
SCCOF	Smart Cooperative Computation Offloading Framework
SDK	Software Development Kit
SL	Supervised Learning
SOSE	Smart Offloading Scheme for Edge
SP	SmartPhone
SVM	Support Vector Machine

THPC	Tactical High Performance Computers
UL	Unsupervised Learning
VM	Virtual Machine

Table of Contents

List of Figures

List of Tables

1. Introduction

International Data Corporation forecast indicates that the number of Mobile Smart Devices (MSDs), (e.g. SPs, iPads and Wearable devices), as well as IoT devices, are increasing continuously, and there will be about 34.2 billion connected devices by 2025 [1], as shown in Figure 1-1. This kind of provision will furthermore drive Computation Intensive Apps/Services (CIASs) to be ported into MSDs. It is clear that millions of such CIASs, as well as the analysis engines/agents behind IoT deployments, (e.g. Tracking systems, Drone-based monitoring, E-health body sensors and Smart home gadgets, etc.) [2], will be emerging shortly. All of which is likely to implement an Artificial Intelligence (AI) engine that requires exhaustive processing and high wireless connectivity. Such a trend has already started, were in 2019, 3 billion MSDs are likely to use an estimation of 417 million of CIASs. As a result, the amount of generated data by these devices are also growing rapidly by 3.9 Zettabytes annually [3]. However, Cisco claims that 92% of the processing of these CIASs have been done by a server in the Cloud, which increases wireless traffic to 16.1 Zettabytes in 2020 alone [4].

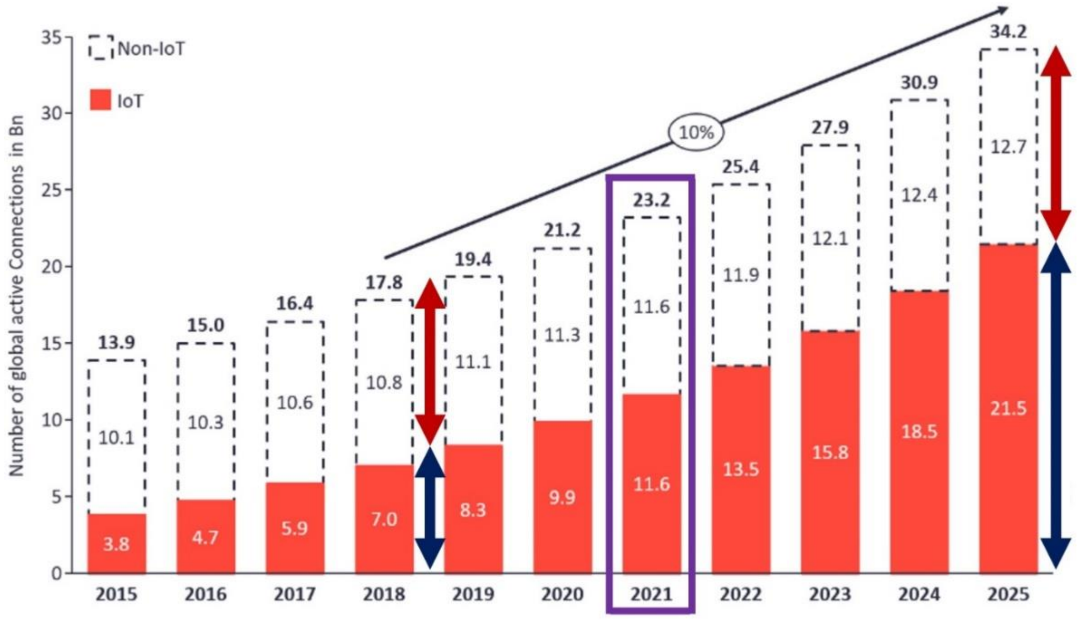


Figure 1-1. Escalation of MSDs and IoT Devices Worldwide

Existing solutions, (detailed in Section 2.1), have proposed to offload the processing of CIASs and IoT deployment from the host-MSD to a Cloud Server (CS) to execute and retrieve results upon completion. The rollout of such CIASs have caused three main issues:

1. Despite the continuous improvements to MSDs resources including connectivity, processing, display, battery capacity and sensors, etc, as shown in Figure 1-3. Yet the battery technology has been much slower to advance compared to the other resources. Based on a survey carried out in 15 countries has concluded that longer battery life is the main desired feature for subscribers to capitalise on using these devices to full [5].

2. Sending/processing sub-tasks and retrieving results from a CS have introduced high latency, processing delay and traffic cost between the host-MSD and the CS. Evidence from recent studies claims an average latency to access a GoogleCloud server is about 87 ms [6]. According to our implementation in Section 4.3 which concludes; an average latency of accessing 3 nearby assisting-MSDs in parallel is equal or even less than accessing a Cloud EC2 server [7]. Assisting-MSDs refer to nearby devices that help the host-MSD processing CIASs, (e.g. Local servers, PCs, SPs, iPads, etc.).

3. Using a CS for IoT deployment will continually suffer from connectivity cost issues, due to oversubscription to the wireless air interface spectrum. In 2019 alone, the amount of data generated from IoT devices to a CS is equal to the data gathered in all prior years [8]. 5G network deployments from 2020 will elevate some of the issues for a while [9].

Edge Computing (EC) [10]; a computing resource infrastructure has emerged to resolve the shortfalls introduce from offloading the processing of CIASs to a CS, by providing the following:

1. Minimising the long-distance communication between the host-MSD and CS by shifting the execution from the CS to servers near the Edge of the host-MSD, to improve the response time, as shown in Figure 1-2.

2. Offering local resources, (e.g. processing and storage), at one hop closer to the host-MSD to save the RTT latency and much-reduced network traffic congestion.

3. Reducing the overall cost as less data is transferred through the network, as well as providing a more robust and secure solution as data/sub-tasks are not uploaded through a public internet connection, as when using a CS [11].

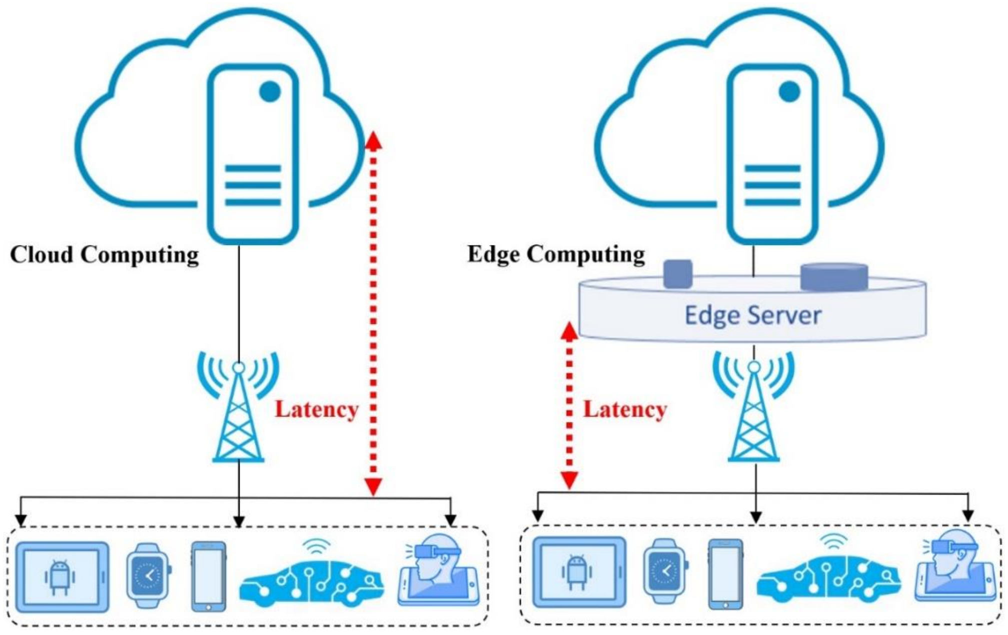


Figure 1-2. The Latency of IoT and MSDs for Cloud and Edge Servers

A review of relevant proposed EC solutions is detailed as part of our literature review stated in Section 2.1. Having the EC infrastructure is of course ideal for the ever-increasing deployment of IoT/CIASs. However, EC solutions are normally pre-planned as part of the overall computing infrastructure in the vicinity of the host-MSD, which is challenging to predict for CIASs/IoT deployments, especially when adopted at various clusters/locations.

Fog computing is a decentralised computing infrastructure, (introduced by Cisco in 2014), where the processing resources, (data, CIASs and storage), are allocated between the Cloud and the end-users. Fog computing is normally referring to bridge/network layer between the Cloud and the Edge, so to perform the computation and routed over to the internet backbone.

The rollout of future networks such as the 5G cellular deployment that are expected to be “Self-Learn”, “Self-Plan” and “Self-Predict” will also impact the progression of CIASs and will no doubt elevate some of the connectivity burden currently witnessed in current technologies. In the same vein, the adaptation of AI-based models, (e.g.

Machine Learning (ML) [12], Deep Learning (DL) [13], Reinforcement Learning (RL) and Deep Reinforcement Learning (DRL)) [14], are needed in such environments to perform intelligent decisions and predictions that will ease the implementation of more and more CIASs, but will further increase the processing/computation of the host-MSD.

Therefore, it was our aim that this development has to meet the following objectives/features to be unique to this thesis, as an individual attribute and as a working system:

1. Smartness: to build a smart solution that can enhance the launching experience of CIASs onboard MSDs, i.e. improving performance, usability and reliability, while reducing resource consumption, traffic congestion and overall cost.

2. Resource Type: to form an Edge Computing Resource (ECR), which is a one-hop communication closer to the host-MSD, reducing both latency and processing delay, as well as enabling the execution of CIASs to run on any Edge assisting-MSDs.

3. Cooperative: to recruit suitable available devices that are willing to cooperate and offer their unused processing resources in a parallel/sharing environment, to aid others voluntarily and/or based on a credit scheme.

4. On-the-go: to build an on-the-go solution, that can form an active resource network of assisting-MSDs together with the host-MSD, without prior planning of infrastructures, compared to static Cloud and Edge offloading solutions.

5. Connectivity Resource: to unleash the computational/processing power of nearby assisting computing devices, in the vicinity of the host-MSD, to reduce the Round-Trip Time (RTT) occurred from offloading to a CS or an Edge Server (ES).

6. Intelligence: to use a Cloud-based AI-engine, that is capable of forming dynamic and intelligent decisions of the available assisting-MSDs, to select the best device with a powerful processor and high-capacity battery power among others. This is necessary to automatically boost-up the decision of what and where to offload, which is normally performed manually in typical offloading frameworks.

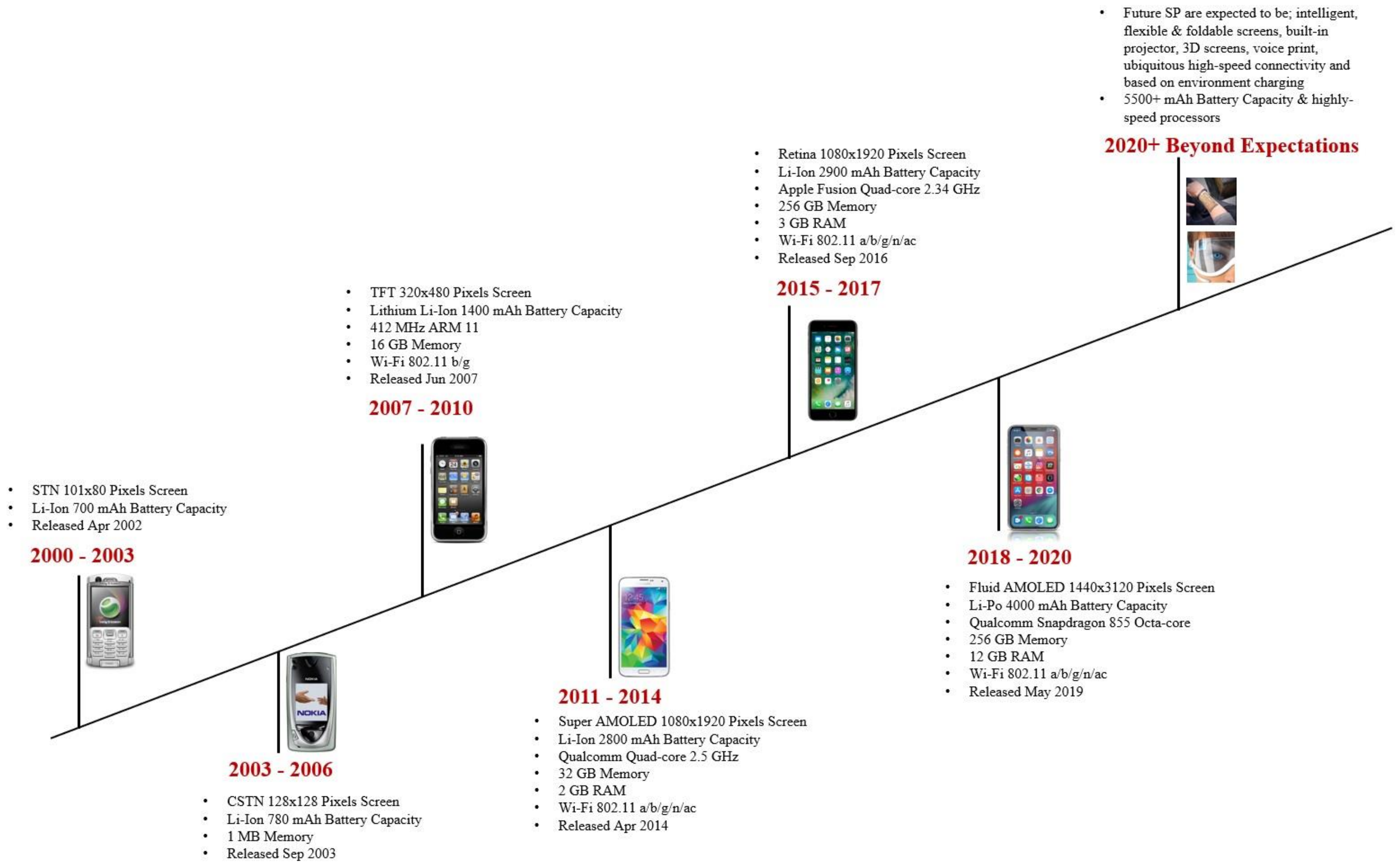

STN 101x80 Pixels Screen
Li-Ion 700 mAh Battery Capacity
Released Apr 2002
2000 - 2003
2003 - 2006
CSTN 128x128 Pixels Screen
Li-Ion 780 mAh Battery Capacity
1 MB Memory
Released Sep 2003
TFT 320x480 Pixels Screen
Lithium Li-Ion 1400 mAh Battery Capacity
412 MHz ARM 11
16 GB Memory
Wi-Fi 802.11 b/g
Released Jun 2007
2007 - 2010
2011 - 2014
Super AMOLED 1080x1920 Pixels Screen
Li-Ion 2800 mAh Battery Capacity
Qualcomm Quad-core 2.5 GHz
32 GB Memory
2 GB RAM
Wi-Fi 802.11 a/b/g/n/ac
Released Apr 2014
Retina 1080x1920 Pixels Screen
Li-Ion 2900 mAh Battery Capacity
Apple Fusion Quad-core 2.34 GHz
256 GB Memory
3 GB RAM
Wi-Fi 802.11 a/b/g/n/ac
Released Sep 2016
2015 - 2017
2018 - 2020
Fluid AMOLED 1440x3120 Pixels Screen
Li-Po 4000 mAh Battery Capacity
Qualcomm Snapdragon 855 Octa-core
256 GB Memory
12 GB RAM
Wi-Fi 802.11 a/b/g/n/ac
Released May 2019
Future SP are expected to be; intelligent, flexible & foldable screens, built-in projector, 3D screens, voice print, ubiquitous high-speed connectivity and based on environment charging
5500+ mAh Battery Capacity & highly-speed processors
2020+ Beyond Expectations

Figure 1-3. SP Roadmap of Processing Resources and Battery Capacity

1.1. Research Methodology and Design Methods

Figure 1-4 shows a typical research methodology that I have followed during my study. In the first 3 months, my focus was on reviewing papers and books about CC to gain knowledge about various services, characteristics and challenges from technical deployment as well as the business model. To have further exposure to the topic, I have decided to attend the CC module that was running at that term, (taught by my supervisor "Prof. Lami"), which has intensively helped me to understand technically about various models, services, characteristics, providers, and other stockholders. Most importantly, I have learned about actual networking issues caused by the CC traffic, including service protocols, traffic congestions and provisions, load balancing/autoscaling, as well as the cost of accessing a Data Centre (DC) to request services. Also, this has helped me to come across my research area of interest; Offloading Techniques for Mobile Cloud Computing Services (MCCS).

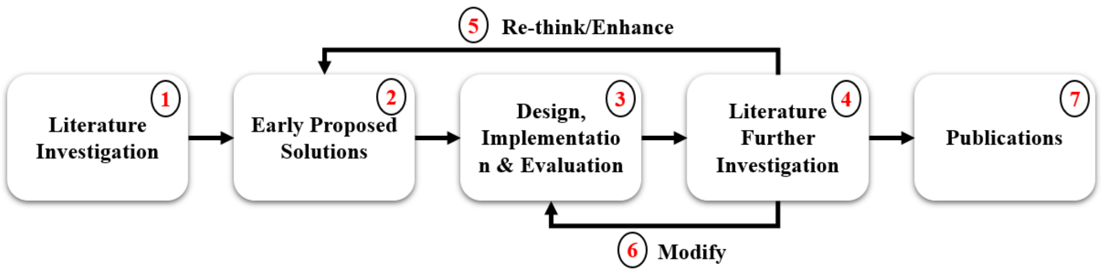

5 Re-think/Enhance
Literature Investigation 1
Early Proposed Solutions 2
Design, Implementation & Evaluation 3
Literature Further Investigation 4
Publications 7
6 Modify

Figure 1-4. Research Methodology

Reading the latest publications is an ongoing task throughout my research period, for the next research stage (approximately 4 months), I have focused my efforts on the literature review. I have reviewed about 70 publications specifically on the topic of offloading frameworks/implementations, most of these are documented in details in Chapter 2.

For the remainder of the first year, (about 3/4 months), I have dedicated my time for hands-on development and activities to truly engage with my topic and practically learn from other published materials by using various simulators. I have practically validated and re-evaluated four offloading frameworks, that I have eventually used them to compare with my implementation as detailed in Section 3.3. These experiments were conducted to offload the processing of CIASs sub-tasks from the host-MSD to a CS and to understand the offloading process including the main models; profiling, partitioning, and decision-making.

Furthermore, as a conclusion of all the studies and hands-on practice, I have done during my first year, crucially I have answered the 5 critical questions in my research summarised below and presented in Figure 1-5.

1. Why to offload: to improve the host-MSD performance/functionality when launching CIASs, so to reduce the processing time, connectivity cost and battery power consumption. As well as to eliminate the overall cost overhead as much as possible, introduced from existing static Cloud and Edge solutions, our solution does not require prior planning as part of the overall computing infrastructure, nor fully depends on the CS unlimited resources.

2. What to offload: to split any CIAS to local and remote chunks/sub-tasks, we defined a relationship to identify offloadability, granularity and complexity, so to build executable sub-tasks. To solve the issue of dependency, we built executable sub-tasks from clustering a group of offloadable sub-tasks, using a Depth First Search (DFS) algorithm [15].

3. When to offload: ideally Apps/Services that require intensive processing with few transfers between devices, through a fast-wireless network, are always advantageous to offload. It is important to consider a low-cost connectivity protocol to use during offloading, as well as a stable network link at all times. We aim to only offload when the assisting-MSDs can execute the required sub-tasks in less time and much-reduced battery power than the host-MSD.

4. Where to offload: we provide an elastic solution that can offloads the processing of CIASs sub-tasks to a pool of nearby assisting-MSDs, and a CS if needed. The selection flexibility of devices is to cope with any type of CIASs that might be cost-effective to run on a specific device and in some cases where such devices are unavailable.

5. How to offload: our solution functions as follow: (a) profiling and partitioning the CIAS to local and remote sub-tasks using a DFS algorithm and automation sampling profiling by generating executable sub-tasks. (b) Forming the ECR-engine that forms an on-the-go resource network of nearby assisting-MSDs, based on instructions received from a Cloud-based AI-engine. (c) A DL decision-making model to schedule sub-tasks to execute on the device with the highest resources, (e.g. fast processor, high-capacity battery), and lowest load, by using a performance scoring algorithm and a Deep Neural Network (DNN) [16]. (d) Establishing a local peer-to-peer connectivity protocol to communicate remotely with the available assisting-MSDs, using Nearby/Post API [17]. (e) Offloading the sub-tasks to the available assisting-MSDs in parallel and retrieve results upon completion.

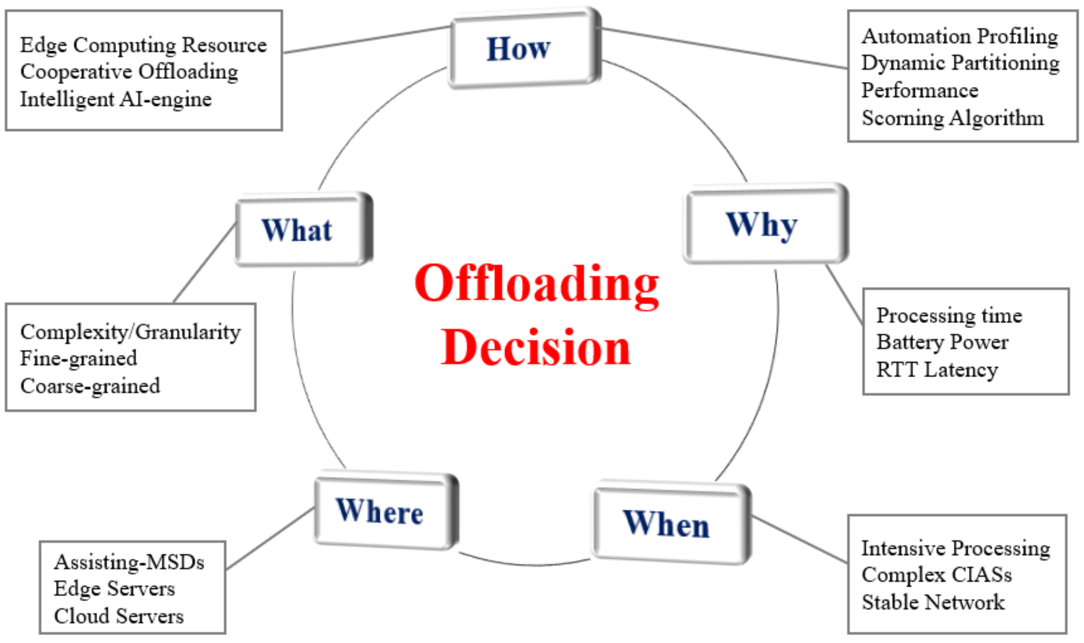


Figure 1-5. Offloading Decision

The above has led to our first proposal "SCCOF" [18], which I have built and published at the beginning of my second year. SCCOF is a Smart Cooperative Computation Offloading Framework that offloads the processing of CIASs from the host-MSD to a CS and retrieves results upon completion, as detailed in Section 3.3.3.

From thus far, it has become important that I also gain technical knowledge of relevant cooperative techniques, and therefore in the following 4 months, I have decided to investigate and read more publications. This has helped me to evaluate the overall cost overhead of processing time and battery power of CIASs deployments. This investigation has led to our second proposal development. This has led to further enhancement of forming an Edge computing engine for this solution. This work has consumed the remainder of the second year.

It took a further 6 months for implementing and testing of various scenarios and experiments, to validate my second solution. The conclusion has led to publish my second “SOSE” [19] solution. SOSE is a Smart Offloading Scheme for Edge services that offloads CIASs sub-tasks from the host-MSD to a group of nearby assisting-MSDs in parallel, by forming an ECR-engine, as detailed in Section 4.1.

Around the mid of my third year and in order to make our offloading novelty truly viable solution for an on-the-go ECR, we decided to deploy a Cloud-based AI-engine to handle the various task allocations. I started to investigate again, to gain knowledge of relevant intelligent solutions, to the best of my knowledge, I did not find an AI-based engine deployed in a cooperative offloading sharing resource. I had to implement and deploy some intelligent models to solve various typical problems, (e.g. image classifications and voice/text recognitions), so to modify such models to address the tasks which my solution intended to solve. Besides, I went to several workshops to learn how to build such AI models, until I arrived at the most suitable implementation, after testing several AI platforms, (e.g. Colab [20], TensorFlow [21], BigML [22] and Amazon SageMaker [23]).

Later on, (approximately 3 months), we have introduced and built a Cloud-based AI-engine to boost-up the offloading decision. The conclusion has led to publish our third “DEO” [24] solution. DEO is a Smart Dynamic Edge Offloading scheme that deploys a Cloud-based AI-engine to advise on the best scenario of profiling and partitioning among the assisting-MSDs in the vicinity of the host-MSD.

In Sept 2019, most of my efforts are focused on publishing my thesis, and so I was spending 4 days a week on writing up my first draft. This task has concluded in July 2020.

1.2. Novelties and Main Tasks

During my research, the following novelties were proposed and developed:

1. Introducing an Offloading Scheme to offload and share the execution of CIASs sub-tasks in parallel among nearby devices in a cooperative manner and a sharing environment. This is achieved by (a) dynamic profilers to monitor the local processing resources of the host-MSD, as well as to monitor the network condition and the complexity/granularity of CIAS sub-tasks. Then based on the output of the profilers,

(b) a DFS algorithm to cluster a group of offloadable sub-tasks together. (c) Migrate the CIASs sub-tasks to the nearby devices in parallel and retrieve results upon completion.

2. Proposing the formation of the innovative ECR-engine that forms a resource network of assisting-MSDs on-the-go, so to help the host-MSD executing the workload of CIAS. It can offload the workload to a pool of nearby devices including assisting-MSDs, Cloud and Edge servers. This is achieved by (a) a peer-to-peer connectivity to network to the devices using Nearby API networking interface. (b) Developing an Offloading Programming Interface (OPI) that can be simply ported/invoked on the participated MSDs, so to perform offloading trials, without prior planning.

3. Developing an intelligent engine, hosted in the Cloud to recruit cooperative MSDs and monitor their availability when needed, so to provide the host-MSD with decisions of the best scenario to partition and offload the overall CIAS. It uses a performance scoring algorithm and a DNN model to schedule the sub-tasks to execute on the best assisting-MSD that has a powerful processor and high-capacity battery power.

During developing the above novities, I have accomplished further achievements that I have decided to include the main contributions/tasks as follow:

- Surveying literature of about 50 Cloud and Edge offloading frameworks so to understand the offloading process, and to evaluate existing publications in terms of (i) partitioning granularity, (ii) offloading gain and overhead cost and (iii) the trade-off impact between the processing time and connectivity cost. (See Chapter 2 for more details).
- Practically validated and re-evaluated four offloading frameworks, which have helped me to build my first client-to-server implementation. Besides, conducting a pilot study to (a) offload the processing of CIASs sub-tasks from the host-MSD to a CS and (b) measure the transmission time of uploading sub-tasks using various wireless networks. (See Chapter 3 for more details).
- Developing an OPI that can seamlessly be invoked/ported on any Edge MSDs, used to perform real-time offloading trials. As we as developing a simple

offloading interface to mimic the functionality of the AI-engine in the host-MSD repository. (See Chapters 4 and 5 for more details).

- Implementing a cost estimator by defining Importance Priority Weighting (IPW) values to estimate the overall overhead cost/efficiency, such as processing time, battery power, RTT latency and throughput. The cost function combines multiple units and scales to be addressed, especially when more MSDs are available to cooperate and share their unused resources. (See Chapter 4 for more details).

- Developing a simple algorithm to calculate the performance score of each MSD used in the implementation so to schedule the most intensive sub-task to be allocated to the best assisting-MSD with the highest performance score and so on. Any device scores a performance threshold of 40% or less will be discarded from the network, as not being qualified to help the host-MSD processing CIASs sub-tasks. (See Chapter 5 for more details).

- Building a dataset of previous executions and from the process of profiling, the dataset reflects about 600 scenarios to boost-up the offloading decision for further executions and to fine-tune the model if needed. The dataset is used to train and test the DNN model, (i.e. the dataset is split to 80% for training and 20% for testing). (See Chapter 5 for more details).

- Finally, on the personal development side, I have participated in teaching CC practical sessions for 3 autumn terms during my research (2017, 2018, & 2019). The teaching sessions focus on learning how to (1) launch a service using a CloudAnalyst tool [25], to evaluate and calculate the average cost of DC response/processing time, Cloud traffic and Virtual Machine (VM) load, when users request a service from a CS. (2) Deploy a service in the Cloud using Amazon Web Services (AWS) Beanstalk [26], to handle (service deployment, on-demand auto-scaling, virtual instances, load balancing and the processing cost of such services. This experience has enriched my knowledge and helped me enormously and technically to accomplish my research. Besides, it gave me a chance to get solid feedback and contributions from the students.

  I have also participated in many academic and industry webinars/seminars, workshops and conferences, so to network with likeminded researchers and to develop my skills, this have enormously helped me to accomplish my research and also result in publishing three academic papers.

1.3. Thesis Structure

This thesis concludes six chapters, organised as follow:

Chapter Two concludes a comprehensive review of publications that focus on (a) offloading frameworks/implementations that offload the processing of CIASs workload from the host-MSD to a CS and/or ES for executions. (b) Partitioning algorithms/techniques that divide the overall CIASs to suitable sub-tasks so to be executed on the chosen devices. (c) Relevant AI-based models and algorithms.

Chapter Three demonstrates experiments and tests conducted in the early stages of this research to validate/evaluate some offloading frameworks/implementations, so to understand the offloading process. It also presents (a) a pilot study to justify the importance of various wireless networks used to offload the processing of CIASs sub-tasks from the host-MSD to a CS. (b) Typical overall cost equation used to measure the local and remote processing time and consumed battery power.

Chapter Four introduces the formation of the On-the-go ECR-engine, which offloads the processing of CIASs sub-tasks to cooperative assisting-MSDs in the vicinity of the host-MSD, and using a CS if necessary. Besides, it demonstrates the creation of the OPI that can be simply ported/invoked on the participated MSDs without prior planning. Also, it introduces a cost estimator model, based on defining IPW values so to calculate the offloading gain and cost overhead. Followed by details of the implementation process and a discussion of the achieved results.

Chapter Five introduces a Cloud-based AI-engine used to deploy a DL decision-making model that can “Self-learn”, “Self-plan” and “Self-predict” to make dynamic and intelligent decisions, so to advise the host-MSD of what and where to offload. This is achieved by implementing a DNN model and a performance scoring algorithm to select the best-assisting device that has the highest resources, (e.g. powerful processor and high-capacity battery power), and lowest load among others, using a pre-build dataset of 600 scenarios. Followed by details of the implementation process and a discussion of the achieved results.

Chapter Six concludes the research work done in this thesis. It starts with a thesis summary of the main achievements/contributions, and also a research discussion including the main remarks and achieved results. Finally, it highlights potential future research directions and recommendations.

2. Literature Review

This chapter reviews technical publications that I have studied during my research, which have enormously helped me to understand the offloading process and so to implement and test my novelties.

At the beginning of my research, I was very keen to understand the depth and breadth about CC, I have read dozens of publications so to investigate the technical aspect of CC models, services, characteristics, providers, and other stockholders. After a good discussion with my supervisor, we have agreed to focus on Offloading Techniques for MCCS as my research direction, and so reading the literature was an on-going task for me at all times. To explore the offloading concept and to get hands-on activity, I have studied the existing frameworks/implementations that focus on offloading the processing of CIASs workload from a host-MSD to a CS/ES for execution and to retrieve the results back upon completion. These frameworks/implementations are detailed in Section 2.1.

While I was presenting my research work at a conference meeting, I have discussed with some likeminded researchers about the latest offloading frameworks publications, that contribute towards the partitioning aspect. We have emphasised that any successful offloading framework/implementation would demand a dynamic and adaptive partitioning algorithm, so to perform accurate decisions and maintain a low overhead cost between the host-MSD and the assisting computing resources. This has motivated me to study various publications that focus on proposing algorithms to partition the processing of CIASs workload to chunks/sub-tasks, so to execute on the host-MSD and the assisting computing resources. These algorithms are detailed in Section 2.2.

My supervisor always believes in the spirit of automation and AI-based models, so to build dynamic and intelligent solutions. While we were attending a Matlab event that was focused on AI-based models, we have discussed whether we shall come up with a unique intelligent solution using ML/DL algorithms together with an offloading system. After a good discussion, I got inspired to investigate relevant AI-based models/algorithms to understand the technical aspect, I have also attended to various seminars and workshops to get hands-on activity, so to build my AI-engine. These AI-based models/algorithms are detailed in Section 2.3.

2.1. A Review of Publications that Focus on Proposing Offloading Frameworks

Frameworks that profile, partition and offload certain chunks/sub-tasks of the overall CIAS workload from the host-MSD to a CS or an ES, has started around 2010. This section of the literature is dedicated to technical offloading frameworks and implementations, I have chosen 17 out of 30 publications that focus on proposing various methods which have helped us in the analysis and the enhancements we adopted in our solution proposals. I have re-evaluated four of these to gain hands-on experience of how to deploy a typical client-to-server interfacing and offloading, as detailed in Chapter 3.

The authors in [27] have introduced COCA; a framework that starts at the CIAS development stage, where the developing team make all the offloading decisions in advance. As shown in Figure 2-1, the source-code developers have achieved this by performing 3 stages, (namely, Profiling, Building and Registration), as follows: (1) analysing the CIAS source-code to determine the complex loops, (e.g. such loops that iterate for ten million times), functions, and execution flow of the App/Service. (2) Using Aspect-Oriented Programming and AspectJ library [28] for various chunks of the source code, (by inserting breakpoints at various natural code boundaries), so to calculate the frame rate per second. (3) Evaluating the execution time and memory requirements for each chunk, when executing locally on the host-MSD or remotely on the CS, and log these values in a report file for future executions. The overall CIAS source-code is then partitioned and compiled into executable sub-tasks to either execute locally or remotely based on the above analysis.

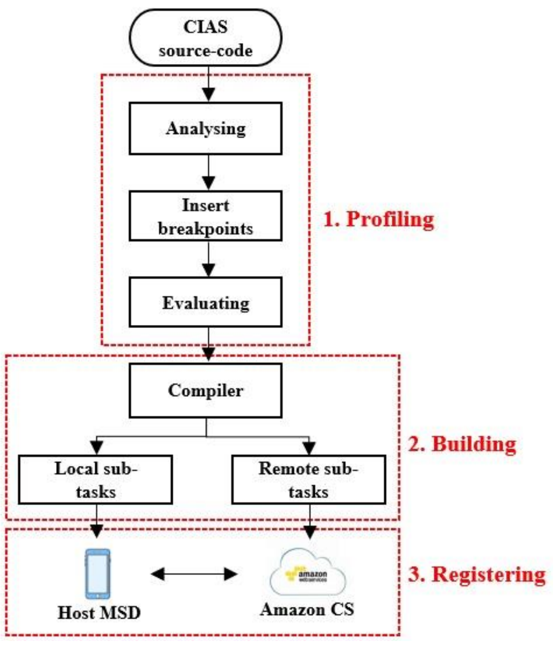


Figure 2-1. COCA Framework Architecture

To evaluate the performance of COCA framework, the authors have developed a chess game CIAS to be tested in the experiments, so to execute locally on the host-MSD, (a Google Nexus One SP that has Qualcomm MSM7225 and 528 MHz CPU), and remotely on a CS, (AWS EC2). i.e. After completing the above 3 stages, the AI module of the Chess game is selected as the most intensive sub-task, therefore it is partitioned and offloaded to the CS, where the remaining of the sub-tasks are processed locally on the host-MSD. The achieved results show improvement of 18x faster execution time and 56% battery power saving when using COCA framework, compared to processing the chess game locally by the host-MSD alone.

We believe that COCA implementation has opened up the gate for many subsequent framework proposals. Albeit, having decisions made by the actual source-code developers, (handmade tailoring), will make the offloading process static, slow, requiring the developer’s engagement in advance.

To address COCA static process and to reduce the burden on the CIASs developers, the authors in [29] have proposed MAUI; an offloading framework that performs dynamic offloading decisions during the execution time of the CIAS. As shown in Figure 2-2, they have achieved this by developing 2 modules positioned on the host-

MSD and the CS, these modules are continually working together, (i.e. using a Client-to-server proxy middleware), so to perform the offloading decision, as follows:

1. Using profilers to collect context information about the host-MSD, CS, CIAS and network, (e.g. processor capability and battery capacity, sub-tasks offloadability/complexity and network throughput). The profilers use linear programming and serialisation reflection techniques to determine local and remote sub-tasks. MAUI controller on the CS side monitors the availability of the CS using Microsoft.NET Common Language Runtime [29].

2. Deploying a solver engine to determine the processing cost for the sub-tasks if executing locally on the host-MSD or remotely on the CS, based on the collected information from the profilers. The solver pre-calculates the battery power saving for each sub-task if offloaded by Wi-Fi and 3G, using a mathematical formula.

3. Offloading the remote sub-tasks to the CS using MAUI middleware proxy, then retrieve the results upon completion. If the current connection with the CS is lost, then the proxy re-invokes the sub-task to execute locally on the host-MSD.

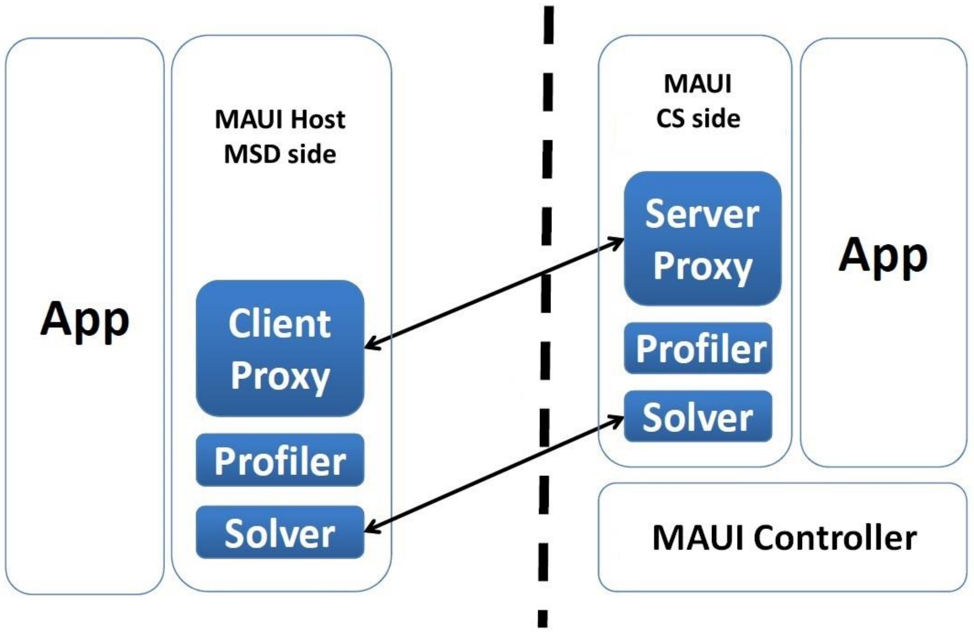


Figure 2-2. MAUI Framework Architecture

To evaluate the performance of MAUI in terms of battery power saving, the authors have used face recognition and chess game as their CIASs to execute locally on the host-MSD and also remotely on the CS, using Wi-Fi and 3G connectivity. Based on the achieved results as shown in Figure 2-3, MAUI reduces the host-MSD battery power consumption by 80% for face recognition and 19% for the chess game, compared to the local execution by the host-MSD alone.

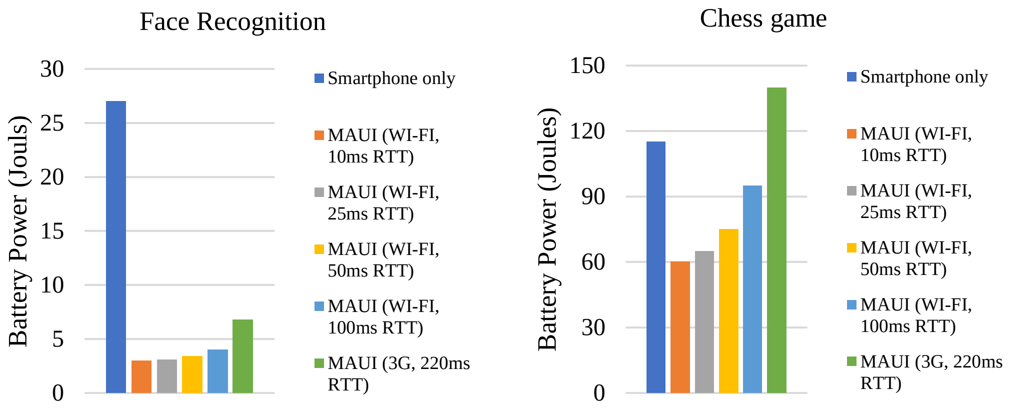


Figure 2-3. MAUI Battery Power Consumption

We believe this framework does provide a dynamic offloading decision at the run-time, and it does improve the host-MSD battery power consumption, by reducing the RTT which is necessary for the overall cost of offloading. Albeit, running profilers continually will increase overall offloading cost, we have noticed from the achieved results that using 3G has increased the overall cost by 5% for the chess game test.

To overcome this issue, the authors in [30] have proposed CloneCloud; a subsequent offloading framework that provides a fine-grained method to automatically determines CIASs local and remote sub-tasks. They have achieved this by developing (1) a static analyser positioned in the CS to determine and exclude the CIAS local sub-tasks, (e.g. sub-tasks that require to access the host-MSD local features and input/output sub-tasks), these sub-tasks must execute locally on the host-MSD at all times. (2) Using dynamic profilers positioned on the host-MSD and the CS, that work together to monitor the host-MSD and CS resources, (i.e. processing capabilities, memory requirements and battery level). The output of the profilers is used to calculate the local and remote processing time and required battery power for the remaining offloadable sub-tasks. The output from the static analyser and dynamic profiler is used by (3) an optimisation solver positioned in the CS, that calculates the overall offloading costs, including the connectivity cost, then starts to offload the sub-tasks to the CS and retrieve results upon completion.

To evaluate the performance of CloneCloud framework, the authors have used Java Virtual tool and Microsoft .NET platform [31] to implement a Virus Scanner (VS) App, (i.e. to scan a 10 MB file stored previously in the host-MSD memory), so to execute locally on the host-MSD and also remotely on the CS. Based on the achieved results as shown in Figure 2-4, using CloneCloud framework improves the host-MSD execution time by 20x and reduces the battery power consumption by 20%, compared to processing VS locally by the host-MSD alone.

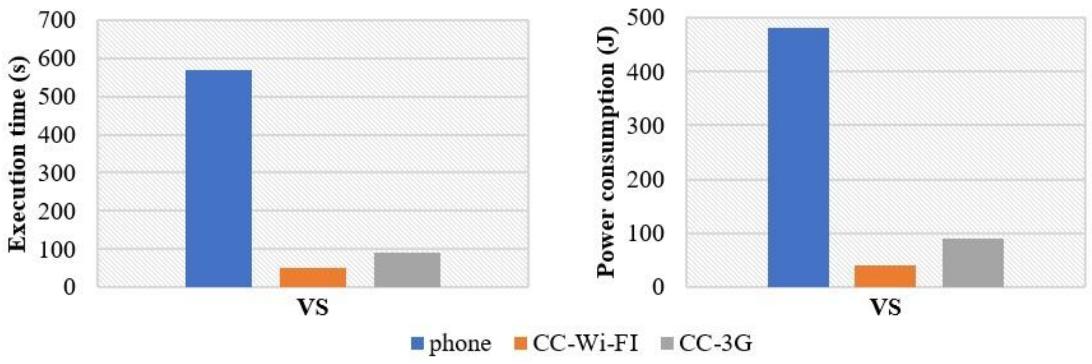


Figure 2-4. CloneCloud Execution Time and Battery Power Consumption

We believe that CloneCloud dynamic profiler method reduces the overall offloading overhead compared to MAUI, as it is much simpler than manual profiling and is executed less often. This framework was a seed for us to think of using an automatic profiling technique so to alleviate the burden on the host-MSD.

The above framework has been enhanced by the work in [32], that introduces ThinkAir; a scalable offloading framework that proposes to use multiple VMs in a CS, so to handle the CIASs workload. These VMs are requested on-demand and meant to work in parallel based on the required workload, (e.g. if the current workload exceeds the capacity of a single VM, then another VM is generated, and part of the workload is migrated to the second VM, and so on). As shown in Figure 2-5, the authors have developed 2 modules positioned on the host-MSD and the CS, these modules are continually working together using ThinkAir middleware, so to perform the offloading decision as follows: (1) using dynamic profilers to gather information from the host-MSD, network and the CIAS, (e.g. Processor capability, battery capacity, data transmission rate and sub-tasks complexity/offloadability). (2) Using an execution controller to measure sub-tasks invocation time, this, as well as the output from the profilers are used then to estimate the local and remote processing time for each sub-task. If the local processing time exceeds the remote processing time, then the sub-task is assigned as a remote sub-task, this is followed by using an API library to compile and annotate the sub-tasks as local and remote. (3) Using a VM manager to start, resume and control the parallelisation of the VMs, it monitors CPU utilisation and MIPS of the VM and invokes further VMs if the workload of any VM exceeds 70% of its processing capabilities. After the analysis of steps 1, 2 and 3 are completed then the framework starts to offload the sub-tasks to the VMs and retrieves the results upon completion, using a client handler which manages the communication.

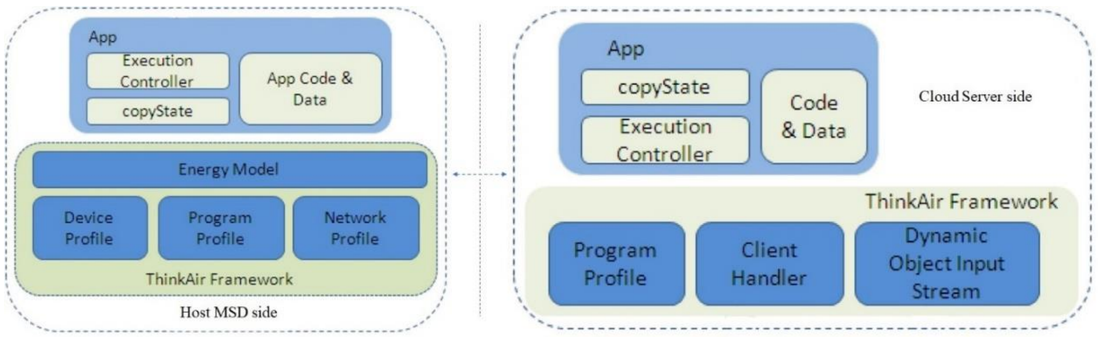


Figure 2-5. ThinkAir Framework Architecture

To evaluate the performance of ThinkAir framework, the authors have implemented N-Queens puzzle game [33] as their CIAS to use in the experiments, so to execute locally on the host-MSD and also remotely on a CS with 6 VMs, using Wi-Fi and 3G networks. Based on the achieved results as shown in Figure 2-6, using ThinkAir framework reduces the execution time and battery power consumption by 60%, compared to the local processing by the host-MSD alone.

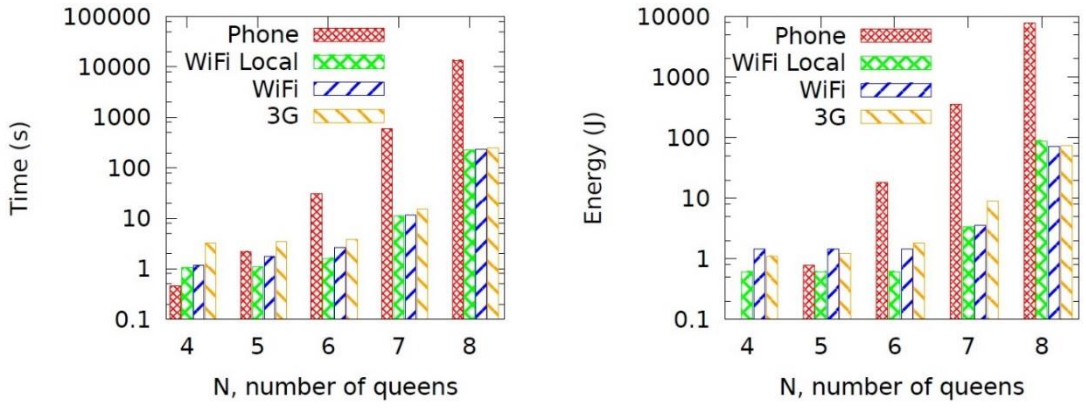


Figure 2-6. ThinkAir Execution Time and Battery Power Consumption

We believe the concept of parallel processing is novel and it is very important to scale up/down resources on-demand, so to cope with the processing demand of CIASs. We have performed a test to generate multiple VMs and scale up/down on demand, so to evaluate the effect on the overall processing and response time, as detailed in Section 3.2.

The authors in [34] have claimed that ThinkAir framework will suffer from connectivity issues, especially if a poor network signal is used at the time of offloading, (e.g. user movement or a sudden drop in the network). They have modified ThinkAir framework and introduced an Enhanced ThinkAir; a delayed offloading algorithm to bypass network drops by avoiding offloading at certain time intervals. This is achieved by (1) checking the network condition using network emulator during the offloading process. If the network condition is poor, (i.e. the remote bitrate is 250 Kbps or less), then the algorithm delays the offloading of the sub-tasks until the network condition is improved for a period of time $T_{deadline}$. (2) Using a power cost model to predict the consumed battery power when offloading the sub-tasks, based on previous executions fetched from a pre-build DB. The algorithm compares the predicted power cost with the actual cost for the sub-tasks if processed on the host-MSD or the CS. (3) Re-evaluating the network condition, if it does not improve, the algorithm then resumes the offloading process as described in the ThinkAir framework.

To evaluate the performance of Enhanced ThinkAir, the authors have implemented an image processing CIAS, so to execute locally on the host-MSD and also remotely on the CS. The following network conditions were emulated and evaluated: (a) stable connection network, (b) sudden drop in the network, (bit rate dropped from 950 Kbps to 600 Kbps), and (c) persistent drop in the network, (bit rate dropped from 900 Kbps to 250 Kbps). Figure 2-7 shows the accuracy of the predicted power cost model, compared to the actual power cost measurements, the model was able to predict the power cost of sub-tasks especially when the number of iterations/executions increases.

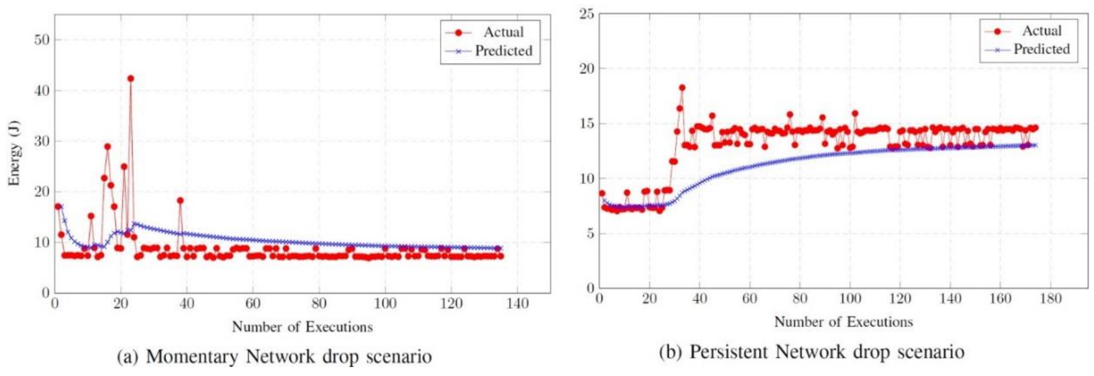


Figure 2-7. Actual and Predicted Power Cost

Also, Figure 2-8 shows that using Enhanced ThinkAir delayed offloading algorithm reduces the host-MSD consumed battery power by 57%, compared to the local processing by the host-MSD alone.

We believe this algorithm proves to adapt itself to various network conditions and changes and will perform well with different connectivity networks. Nevertheless, we also think it will encounter processing delays in making offloading decisions.

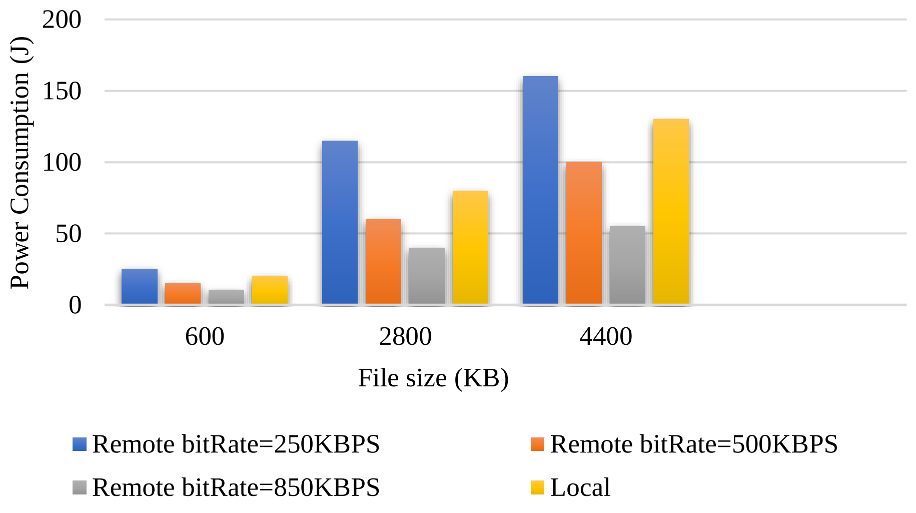


Figure 2-8. Enhanced ThinkAir Battery Power Consumption

To address the issue of the processing delay in the above proposal, the authors in [35] have introduced MACS; an elastic offloading framework that performs automatic decisions for any CIAS, with minimum modification. As shown in Figure 2-9, the authors have developed 2 modules positioned on the host-MSD and the CS, these modules are continually working together using MACS service proxy, so to enable the deployment of MACS to perform the offloading decision as follows:

1. A performance context monitor to profile the complexity and granularity of CIAS workload, (e.g. searching for loops and functions that iterate for 100 times). Then it calculates the memory requirements, CPU load and battery power for processing each loop/function locally on the host-MSD and also remotely on the CS, using an offloading manager.

2. A pre-build MACS library to generate local and remote executable sub-tasks from these loops/functions, so such sub-tasks can execute automatically on the host-MSD or the CS.

3. An adaptive scheduling mechanism to determine where to execute such sub-tasks based on a history DB that registers all previous executions. It stores the sub-tasks in a queue and starts to schedule them in sequential order.

4. A service manager to monitor the connectivity middleware between the host-MSD and the CS, it uses inter-process communication and Android proxy to enable remote procedure calls, so certain sub-tasks can be processed remotely on the CS. After the process of steps 1, 2, 3 and 4 are completed, then MACS starts to process the local sub-tasks on the host-MSD and offloads the remote sub-tasks to the CS, then retrieves the results upon completion.

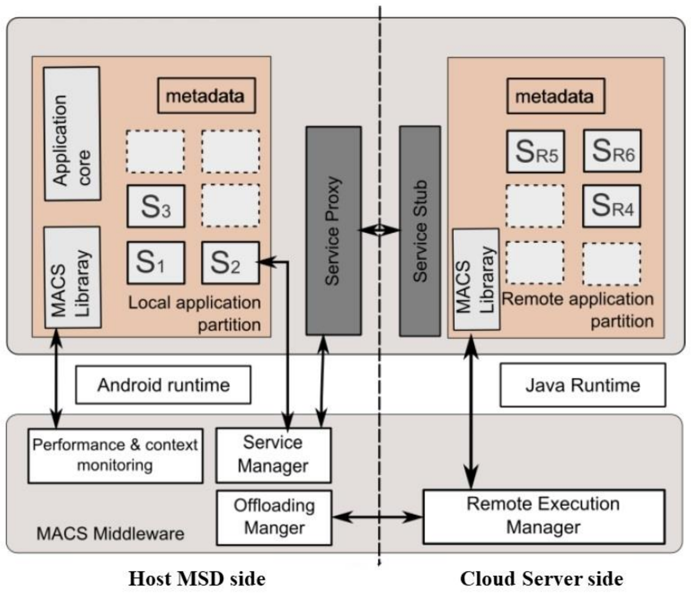


Figure 2-9. MACS Framework Architecture

To evaluate the performance of MACS framework, the authors have implemented a video face detection as their CIAS to test in the experiment, (i.e. to detect faces from a video stream with a duration between 10-60 s). The face detection module is assigned as a remote sub-task as being the most intensive sub-task and therefore must execute on the CS, where all the remaining sub-tasks are assigned as local sub-tasks and must execute on the host-MSD. They have used a host-MSD, (an SP that has ARM A8 600 MHz and 256 MB memory), and a CS, (a PC that has Quadcore 2.83 GHz and 8 GB memory). As detailed in Table 2-1, the achieved results show that using MACS can achieve 20x speed up and 95% battery power saving, compared to the local run by the host-MSD alone.

Table 2-1. Execution Time and Battery Power Consumption

Video Duration (s)	Execution Time (min) – Host/Server	Battery Power (kw) – Host/Server
10	14.8/0.7	0.9/0.1
20	27.6/1.0	1.7/0.1
30	42.6/1.8	2.6/0.1
40	62.5/2.2	3.7/0.2
50	77.7/2.8	4.7/0.2
60	96.7/3.3	5.8/0.3

We believe that the concept of this framework is important, as a scheduling mechanism is necessary to determine where and when to execute certain sub-tasks, the motivation of this work has led us to include a scheduling mechanism to our solution, as will be detailed in Section 5.2.5.2.

Similarly, other authors have introduced some associated offloading frameworks which I have decided to briefly mention here for more information. The authors in [36] have proposed a multi-parameter decision framework to offload the processing of CIASs workload from the host-MSD to fog computing networking/infrastructure. They have used Femtocloud, (i.e. small low-power APs positioned in a home or small office), so to reduce the offloading traffic. The authors in [37] have proposed a Cloud migration strategy to offload the intensive sub-tasks from the overloaded servers to the unloaded servers in a Cloud DC, so to maintain the workload within the DC.

Reviewing the above offloading frameworks have helped me to conclude the following: (1) a successful offloading framework/implementation would have to improve the host-MSD execution time and consumed battery power consumption when processing CIASs workload. Offloading is most useful for sub-tasks requiring intensive processing that require few transfers between the host-MSD and assisting CS, using a fast connectivity, (p2p Wi-Fi), otherwise offloading is not beneficial. Albeit, it is important to consider sub-tasks complexity/granularity, connectivity cost between the host-MSD host and assisting CS and the total processing delay to receive the results back. (2) A suitable partitioning algorithm is essential to decide on sub-tasks complexity, offloadability and granularity to be offloaded, as well as the overhead cost between the host-MSD and the assisting CS.

An advantage of EC is to provide computing resources, (e.g. processing and storage), near the end-users, so to reduce processing delay and overall traffic cost, introduced by offloading sub-tasks to a typical CS for execution. This has been an incentive for us to review publications that address the saving and improvement that can be offered when offloading sub-tasks to an ES. The below publications focus on proposing EC implementations, which have helped me to build my second proposal solution, SOSE; a Smart Offloading Scheme for Edge services, as will be detailed in Chapter 4.

The first of these publications is an auto-scaling model that offloads the processing of CIASs sub-tasks from the host-MSD to an ES. It enables multiple virtual resources in the ES to be utilised for running sub-tasks offloaded from the host-MSD [38], as shown in Figure 2-10. The authors have achieved this by implementing four models, (namely, static and dynamic analysers, proxy monitor, distributor and executor), deployed on the host-MSD. As well as six models, (namely, register manager, predictor, decision solver, virtual resource controller, partition analyser and executor), deployed on each virtual resource in the ES. These models are working together to perform the offloading decisions as follows (1) determine local and remote sub-tasks and exclude the sub-tasks that require to interact with the host-MSD local features, (e.g. input/output, nested loops and sub-tasks that access to camera/memory). The remaining of the sub-tasks are considered as offloadable sub-tasks. The static analyser uses 0-1 integer linear programming to annotate the sub-tasks with 0/1, where 0 annotates a local sub-task, and 1 else wise.

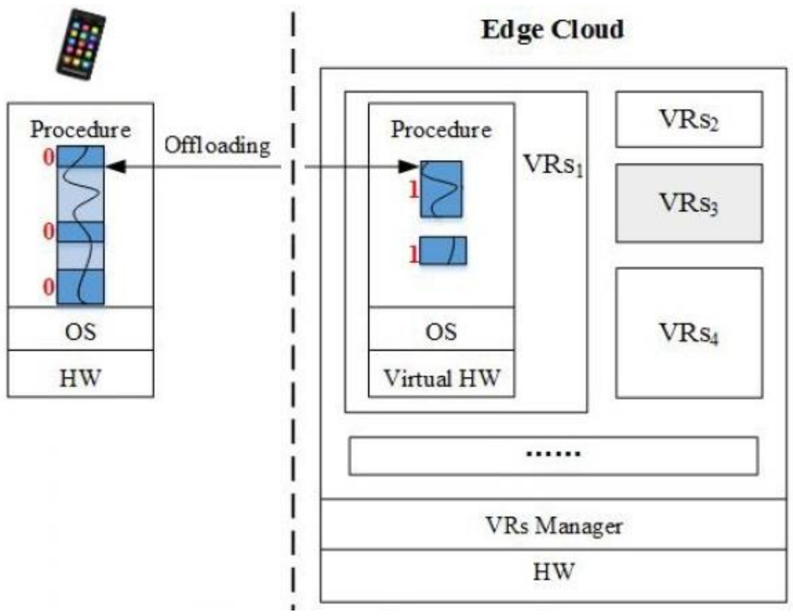


Figure 2-10. ES Auto-scaling Model Architecture

(2) Estimate the cost of running the sub-tasks locally and/or remotely, by analysing the current network condition based on the bandwidth of the connection link used at the time of offloading. (3) Monitor the virtual resources and sub-tasks queue on the ES. The decision solver decides which virtual resource to select for the incoming sub-tasks from the host-MSD based on the current queue and completion time. (4) Interact with the ES and start to offload the sub-tasks accordingly and retrieve results upon completion. The virtual resource manager is responsible to migrate the sub-tasks from the loaded virtual resource to the unloaded one if the capacity of a certain virtual resource exceeds its limit.

The concept of shifting the processing of CIASs sub-tasks from the CS to an ES is unique in that it will save the RTT latency and battery power of the host-MSD, and this eventually will no doubt minimise the overall traffic cost caused by existing CS implementations. However, this model needs to be properly thought off and processed intelligently to achieve maximum gains. This was a seed for thinking about forming our Edge computing resource.

Tracking systems, Drone-based monitoring, E-health body sensors and Smart home gadgets are examples of nowadays Apps [39]. These Apps require AI-based models to analyse streams of audio/video data coming from many sensors. Deploying such AI-based models require significant processing resources, which may not be ideal for running an ES, but more suited for a CS. An offloading model that balances the processing load between a CS and ES has been proposed in [40], so to address the computation intensity introduced from deploying such AI-based Apps. The authors have achieved this by shifting the AI training and testing sub-tasks to the CS, as being the most intensive sub-tasks, whereas the AI inference engine sub-task is deployed in the ES, that is necessary to handle processing/analysis nearer to the host-MSD, as shown in Figure 2-11. i.e. The host-MSD offloads data to the CS, the data is then labelled, trained and tested by multiple AI models, based on the chosen decision, the model is retrieved, sterilised and stored in a shared repository. The AI inference engine, which is a microservice is invoked/positioned at the ES that can be accessed through the shared repository.

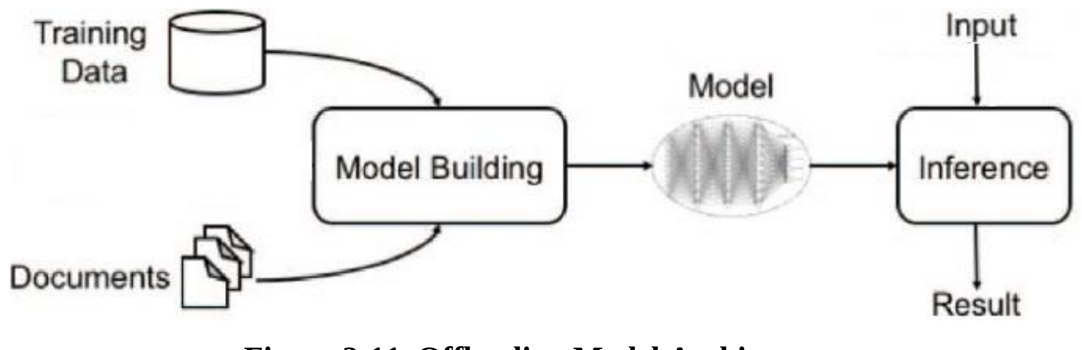


Figure 2-11. Offloading Model Architecture

This model concept is impressive in that it balances the load among the CS and ES, resulting in fewer data being shipped to the CS, which theoretically reduces the traffic cost. We believe that the concept of including a CS for the overall decision-making in splitting the processing load between the CS and ES is commendable. We have introduced a similar concept, we used the Cloud-layer to deploy an AI-engine to boost-up the typical offloading decision, as well as the Edge-layer to deploy the ECR-engine to form an on-the-go resource network of nearby assisting-MSDs, as will be detailed in Chapter 4.

The concept of the above-proposed model has been enhanced by the work in [41], that introduces a mechanism to implement a DL model for IoT Apps and schedule its layers to execute/invoke seamlessly across the CS and ES, as shown in Figure 2-12. The authors have achieved this by (1) deploying a mathematical formula to calculate the capacity of each ES based on the processor resources and current load. (2) Using a historical DB that provides a maximum and minimum required data transmission rate for executing each sub-task on the ES. It monitors the ES and starts to execute DL layers in a sequential order while observing the current queue and load, then it starts to increase the number of layers while monitoring the overall performance of the ES. i.e. The first input layers are consisting of many processing computes layers, therefore, it is more beneficial to run such layers in a CS. Then, when the dimension of the DL network is reduced, and the size of the intermediate layers becomes smaller than the input layer, these layers are then deployed in the ES.

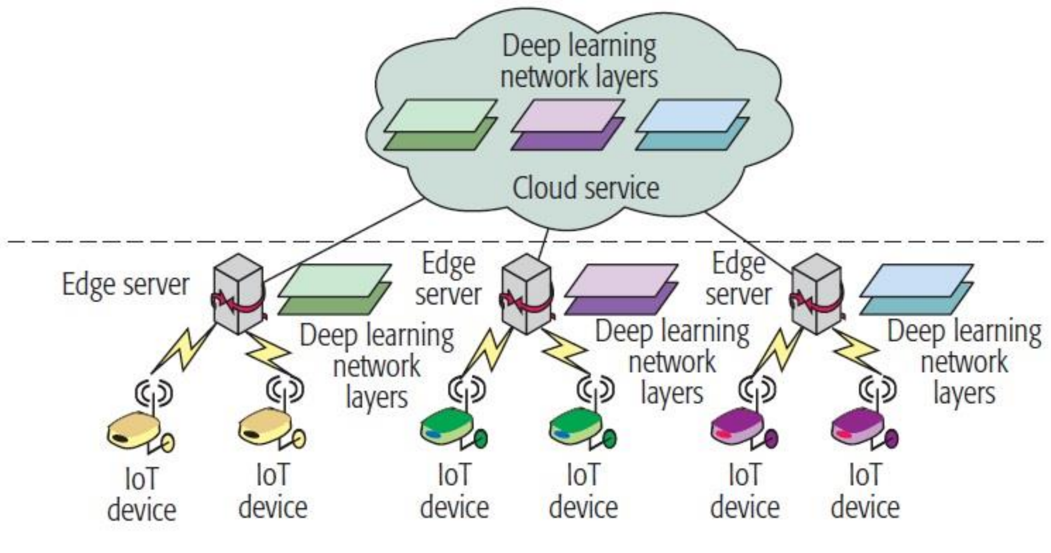

Deep learning network layers
Cloud service
Edge server
Deep learning network layers
Edge server
Deep learning network layers
Edge server
Deep learning network layers
IoT device
IoT device
IoT device
IoT device
IoT device
IoT device

Figure 2-12. Scheduling Mechanism Architecture for IoT Deployments

To evaluate the performance of the scheduling mechanism, the authors have used AlexNet CNN [42] that consists of 8 layers. After running the above analysis, the network is structured to run across the CS and ES and the decision is formed to deploy the first 5 layers in the CS, where the last three layers are positioned in the ES. The scheduling mechanism is compared with two popular scheduling mechanisms, (namely, First In First Out and Low Bandwidth First), the testing shows that the proposed mechanism outperforms the two other mechanisms, as it generates less data transfer by shifting more sub-tasks from the CS to the ES.

This has inspired us to include a scheduling capability in our AI-engine, so to schedule CIASs sub-tasks to execute among the assisting-MSDs, and also to select the MSD that has the highest resources, (e.g. fast processor and high-capacity battery power), as will be detailed in Section 5.2.5.

A model that aims to enhance the above-proposed scheduling mechanism has introduced a delay-aware task-graph algorithm and an optimal VM selection method, so to dynamically allocate sub-tasks to run across a CS and ES [43], as shown in Figure 2-13. The authors have achieved this by (1) implementing a sort algorithm that analyses a typical task-graph algorithm and chooses the best topological cut-off of sub-tasks that is likely to perform well if executed on a CS or an ES. It uses a Kahn algorithm [43] that has a low polynomial processing complexity, so to efficiently find the best cut-off among sub-tasks while reducing the overhead cost of the deployed algorithm. (2) Ranking the available VMs based on the processing resources and current load, as well as considering the processing time that each VM takes to execute a sub-task at a certain time interval. (3) Selecting the most appropriate VM that consumes less time which therefore helps to minimise the overall processing delay.

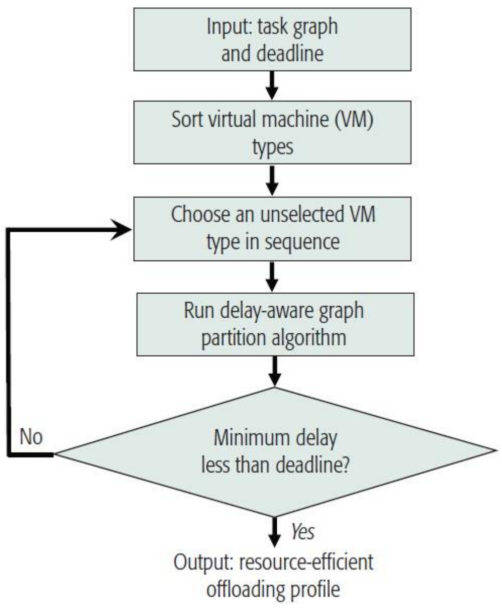

Input: task graph and deadline
Sort virtual machine (VM) types
Choose an unselected VM type in sequence
Run delay-aware graph partition algorithm
Minimum delay less than deadline?
No
Yes
Output: resource-efficient offloading profile

Figure 2-13. Proposed Solution Architecture

The experiments and achieved results of the algorithm have reduced sub-tasks processing delay by 19%, compared to a typical task-graph partitioning algorithm. Albeit, it only considers the processing time as a metric to evaluate the proposed model.

We believe other metrics like battery power and connectivity cost are very essential to evaluate the performance of the algorithm and to measure the RTT latency using various connectivity technologies. Therefore, we believe an adaptive algorithm that can intelligently learn and adapt itself accordingly is needed, especially if deployed in some heterogeneous environments, (e.g. accessibility to the assisting computing resources might be restricted to various wireless technologies).

Various wireless network technologies, (e.g. Wi-Fi, cellular and BT), have a precise data transmission rate and RTT latency that can influence the connectivity cost negatively. As a result, using a particular wireless network will certainly affect the offloading processing time and overall cost. A model that addresses the above has proposed an emulation testbed to emulate the network conditions of various wireless technologies, so to select the best network to use during the offloading process [44]. The authors have achieved this by (1) manipulating four wireless technologies, (namely BT, Wi-Fi direct, Wi-Fi and 3G), while offloading the processing of CIAS sub-tasks from the host-MSD to several assisting ESs, using a traffic shaper. (2) Emulating the network conditions of such technologies to generate specific RTT, packet loss and data transmission rate, so to reflect a real network condition. The connectivity cost was examined while pushing multiple sub-tasks through the network. The testing shows that using BT to offload improves the host-MSD processing time and battery power consumption, it outperforms Wi-Fi and 3G by 25% and 44%. Albeit, using Wi-Fi network is still promising if small RTT is introduced, (e.g. nearby server). Nevertheless, 3G is yet to be the ultimate network use while offloading the processing of CIASs sub-tasks from the host-MSD to an ES.

To justify our implementation, this model, as well as our connectivity cost analysis test conducted in Section 3.3, have affirmed us that an efficient wireless network is vital to use, so to achieve a better offloading outcome, (i.e. to eliminate the connectivity cost, which will eventually reduce the overall processing cost). We have addressed this by implementing a local peer2peer connectivity, using Nearby API interface, so to network with nearby assisting-MSDs, while eliminating the RTT latency and the overall network traffic congestion.

To measure the cost overhead introduced from forming an EC infrastructure, the authors in [45] have proposed a mapping-optimisation technique to select the most suitable server to use, so to achieve a low-cost overhead. The authors have achieved this by implementing (1) a mathematical cost equation to calculate the cost of deploying such a prediction technique, so to minimise the overall cost overhead. It forms the mapping-optimisation problem as an MDP model to calculate the State, Action and Reward functions. The State function defines how many servers are available and what is the processing capacity for each. The Action function decides which server to use and the Reward function identifies a minimum cost to achieve. (2) A knowledge-passing mechanism to exchange information between several ESs in multiple regions, (such as location, current load and processing capability), so to predict the next available server to select. It selects the next available server by calculating the distance, compare the shortest distance and choose the best server.

The authors have claimed that their technique reduces the overall cost overhead by 80%, compared to other relevant static techniques. However, their claim is based on the assumption of the existence of a resource scheduling algorithm that can schedule the load across servers, and migrate the load between servers if necessary. Nevertheless, the motivation of this model has helped us to think of forming a cost model to estimate the cost overhead of our solution, in terms of sub-tasks processing time, consumed battery power, RTT latency and efficiency gain.

From a different perspective, offloading the processing of CIASs sub-tasks and sharing the end-user data to a CS or ES may lead to an insecure deployment inviting malicious activities. The authors in [46] have proposed a scheme that intends to secure sub-tasks/data before being shared from the host-MSD to an ES. They have achieved this by (1) implementing a security manager interface to encrypt, exchange session keys and verify the data before starting the offloading process. (2) Synchronising to the ES through a secure middleware that handles the communication and monitors the offloading process, it observes the ES and generates alerts if a breach has occurred.

Even though, to the best of our knowledge, this scheme is the first to address security issues, yet, it lacks details of the proposed architecture, nor experiments to prove the novelty.

Similarly, another publication has proposed to provide a secure model to select trustable assisting computing resources to use during the offloading process [47]. The authors have achieved this by (1) implementing a social-trust engine that infers social-trust factors from social relation between two devices. i.e. A trustable assisting resource is only selected if it meets the social-trust factor launched by the host-MSD. (2) Deploying a secure service broker that uses a tracker to authenticate the process and build a secure firewall between the chosen assisting devices. In the same vein, the authors in [48], have listed some techniques that can be implemented to secure the data before sharing it with a CS or an ES and to authenticate the assisting devices. These are steganography, trusted session keys, hardware-based secure deployment and homomorphic encryption.

There is no doubt that the above three security publications are immature contributions that theoretically intend to secure the offloading process, we believe more work needs to be done in such direction, (e.g. the cost of deploying such models/engines influence on the overall offloading cost). I must admit that our solution does not claim to provide a security engine as the above, as not being the focus of my research. Albeit, our solution does implement a secure deployment by using (1) AWS Rekognition [49]; a highly secure service that uses access and secret keys to authenticate the nearby assisting-MSDs. (2) Nearby API protocol; a secure middleware that provides fully encrypted p2p data transfer [17].

The motivation of the above publications has helped me to build a novel ECR-engine that forms a resource network on-the-go of nearby assisting-MSDs via short-range wireless connectivity, without pre-setup planning of the overall infrastructure. Our solution offloads and shares the processing of CIASs sub-tasks among the participated assisting-MSDs in parallel together with the host-MSD.

2.2. A Review of Publications that Focus on Proposing Partitioning Techniques

Partitioning is a major task in the offloading process as explained in the introduction of Chapter 2. The partitioning process of interest to this research is to determine that the CIAS is divided into multiple chunks/sub-tasks, that can eventually be offloaded to run amongst the newly formed resource network of participating MSDs/Servers. I have reviewed around 20 publications that helped me to understand the depth and breadth of such partitioning algorithms. In this section, I have chosen to include 10 algorithms that proposed to comply with our criteria to meet the processing demand of CIASs and to cope with different networking infrastructures, (e.g. Cloud and Edge deployments), so to make accurate offloading decisions.

The authors in [50] have proposed a static partitioning algorithm to divide the workload of CIAS to multiple chunks/sub-tasks, so to execute locally on a host-MSD for most sub-tasks while offloading the intensive sub-tasks to execute remotely on the CS. The algorithm works as follows: (1) making all sub-tasks that interact with the host-MSD local features, (e.g. input/output sub-tasks, sub-tasks that access the camera/memory and dependent sub-tasks that require output from the previous sub-task), to execute locally on the host-MSD. (2) All the remaining sub-tasks are considered as offloadable sub-tasks and therefore can be executed remotely on the CS, based on a mathematical partitioning formula, which calculates the overall offloading cost, including the local and remote processing time and the connectivity time between the host-MSD and CS. The formula is based on a static analysis using HP iPAQ PDA and based on the assumption that the CS is always available to help and have 10x powerful processor, compared to the host-MSD.

This algorithm was a seed for us to understand the effort of partitioning, so to make a good decision of where is best to execute such sub-tasks, as well as to calculate the local and remote processing time. We believe this algorithm will perform well in static scenarios, (e.g. the CS is always available/accessible and the network type is pre-defined from the outset). But we also think the algorithm is likely to suffer in heterogeneous computing environments, (i.e. network changes from Wi-Fi to 3G and/or the CS is busy executing some other sub-tasks), in such cases, a dynamic algorithm is important to adapt itself to these changes.

To address the above issue, the authors in [51] have proposed a dynamic partitioning algorithm using Genetic Algorithm (GA), so to find the optimal offloading strategy that is likely to fit the workload of CIASs, if deployed in stochastic/changing environments. i.e. A stochastic environment means when there are multiple assisting computing resources, including a CS and/or ES together with the host-MSD. The authors have achieved this by applying GA that decides where to execute the sub-tasks, by calculating the overall processing time, latency and cost for each sub-task, if executed locally on the host-MSD or remotely on the CS/ES. GA uses 5 mathematical search operations, (namely, initialisation, selection, crossover point, mutation, and fitness function), so to topologically sort a chain of sub-tasks sequentially. As well as to solve the issue of sub-tasks dependency, by defining a weight factor that represents the size of data transmission for each sub-task. Based on further calculations, it selects the optimal portion of sub-tasks, without modifications to the chain execution order, and therefore starts to offload the sub-tasks to the assisting computing resources.

To evaluate the performance of the algorithm, the authors have tested the algorithm to run under different scenarios, (i.e. (i) all sub-tasks execute locally on the host-MSD, (ii) all sub-tasks execute remotely on the CS, (iii) or using GA to partition the sub-tasks to execute between the host-MSD and the ES). The algorithm is developed using Java language and implemented using Eclipse platform [52], the achieved results show that GA reduces the processing time by 14%, compared to executing all sub-tasks on the CS. As well as GA reduces the consumed battery power by 40%, compared to the local execution by the host-MSD alone. The cost overhead of deploying GA has increased by 33%, compared to the CS scenario, this overhead has been triggered by performing GA five mathematical operations repeatedly until it reaches the best offloading decision.

We believe the concept of this algorithm has opened the light to more work in this direction, there is no doubt that this dynamic algorithm performs better than the previous static algorithm, as it can adapt itself to consider multiple assisting computing resources.

The authors in [53] have indicated that using GA will trigger an overhead cost, and so they have proposed an adaptive partitioning algorithm that can be adapted to various dynamic scenarios, (i.e. any number of assisting computing devices without a need to pre-plan the launching infrastructure). They have achieved this by (1) developing an algorithm to establish if the sub-task has been offloaded before from previous executions, and therefore if it does, it fetches the local and remote processing time and the required battery power, from a pre-build local DB. (2) Using a dynamic profiler positioned on the host-MSD to monitor the host-MSD local resources and network status of the available connectivity type used at the time, (e.g. CPU cores, Memory capability and latency). If the sub-task has not been offloaded before, and therefore there is no records of required processing time and battery power in the DB, the algorithm then decides where to execute such sub-task locally on the host-MSD or remotely on the assisting computing device, based on the output from the dynamic profiler. After completing the above analysis, the assisting computing devices execute the requested sub-tasks and send the results back upon completion. Finally, the algorithm updates the DB for new sub-tasks and/or if there are any changes gained from the current execution.

To evaluate the performance of the algorithm, the authors have implemented quicksort and N-Queens as their CIASs, so to test various scenarios, (i.e. (a) all sub-tasks execute locally on the host-MSD, (b) all sub-tasks execute remotely on the assisting devices or (c) using the algorithm to partition the sub-tasks between the host-MSD and the assisting computing devices). The achieved results show that using the algorithm to partition the sub-tasks to execute among the computing devices has reduced the processing time by 3x, compared to the local execution by the host-MSD alone. Nevertheless, the algorithm is based on the assumption that the sub-tasks are already offloadable and compatible to execute on any computing device, without worrying about sub-tasks complexity and granularity. This may affect the sequence order of the App/Service structure, (e.g. dependent sub-tasks where a sub-task is required output from a previous sub-task, therefore both sub-tasks have to be assigned and executed by the same assisting device).

We believe the concept of using a local DB to learn from previous executions is novel, as this can save the overall delay from calculating the processing time, battery power and cost from scratch. This has inspired us to think of an adaptive algorithm that makes benefit from previous executions, as well as to learn from current executions so to update itself for future executions.

To address the above issue, the authors in [54] have proposed a partitioning algorithm that automatically determines the suitable chunks/sub-tasks to be offloaded and shared, without affecting the sequence order of the App/Service flow. They have achieved this by implementing a call-graph algorithm that forms a topological chain of sub-tasks, so to mimic an App structure in sequential order. The call-graph algorithm is normally formed using mathematical equations, so to estimate the weight cost of Vertex that represents a sub-task and Edges that represents a relationship between two sub-tasks. Each Vertex calculates the processing weight cost for executing a sub-task locally on the host-MSD and remotely on the assisting CS, while each Edge calculates the connectivity weight cost between the host-MSD and the CS. The algorithm continually calculates the above for each sub-task individually, until it reaches the best possible decision so to accommodate the sub-tasks to be offloaded, that are likely to minimise the overall cost overhead.

The authors have indicated that the algorithm can accurately calculates the weight cost for each sub-task, and therefore it forms the decision automatically during the deployment time. We believe this algorithm can effectively partition the sub-tasks to execute among the assisting computing devices, without affecting the execution sequence order, but it calculates the weight cost for a single sub-task at a certain time interval, which will eventually burden the host-MSD that is also consuming processing time and battery power executing the local sub-tasks.

The authors in [15] have proposed to eliminate the individual weight calculation of sub-tasks processing time and connectivity cost at a certain time interval, but rather aimed to cluster and offload a group of sub-tasks at once, so to minimise the overall cost overhead. They have achieved this by developing a DFS algorithm that searches for the best integration point to divide a chain of sub-tasks, and therefore group multiple sub-tasks that will be offloaded as a whole package. The DFS algorithm uses a linear time-search scheme, so to search for different integration points and determine the best among them, without modifying to the sub-tasks sequence of order. It excludes the sub-tasks that require to access to the host-MSD local features and UI sub-tasks, these sub-tasks are executed locally on the host-MSD at all times. For the rest of the sub-tasks, it clusters them together in groups and offloads them as a package to the assisting CS. The CS then receives the package, executes the sub-tasks in sequential order and sends the results back to the host-MSD upon completion. To evaluate the performance of the algorithm, the authors have implemented the algorithm and compared it with the above algorithm that performs individual weight calculations. The achieved results show that DFS algorithm reduces the overhead cost by 60%, compared to the individual weight calculation algorithm.

The concept of this algorithm is unique as it reduces the overall cost overhead, as well as it can dynamically partition the CIAS to a chain of offloadable sub-tasks without affecting the execution order. Albeit, the algorithm is implemented on the host-MSD and this may increase the deployment overhead cost. The motivation of this has inspired us to use a similar concept in our solution to create executable sub-tasks. We simply modified it to fit our solution, we only stored 2-inputs in the host-MSD local repository, local-vertex and remote-vertex, rather than fetching these inputs from the assisting device every time a sub-task is executed.

To minimise the overhead cost introduced from implementing the partitioning algorithm on the host-MSD. The authors in [55] have introduced a D2D model that aims to partition the execution of CIAS to local and remote sub-tasks. They have achieved this by (1) deploying the partitioning algorithm on a laptop that acts as a middleware layer between the host-MSD and the assisting computing resources, so to elevate the overhead cost from the host-MSD. The algorithm selects a device only if it satisfies the following: (a) executes a sub-task in less time compared to processing the sub-task locally on the host-MSD. (b) Maintains a minimum data-rate with the middleware layer, and (c) must have a sufficient battery power level of 40% to execute the sub-task and return results. (2) Connecting the host-MSD and assisting devices to the middleware through Wi-Fi link, upon a successful connection, the assisting devices wait for any upcoming sub-tasks, then they execute the sub-tasks and send the results back to the host-MSD. To evaluate the performance of the algorithm, the authors have developed a prime App as their CIAS, (i.e. to find the prime number between 1 to 300000), which is then divided it to 6 sub-tasks and offloaded to multiple assisting devices. The achieved results show an improvement of 3x saving to the overall processing time, compared to the MDC model [56].

We believe the concept of this algorithm is important to alleviate the burden on the host-MSD from doing the offloading analysis, which will eventually reduce the overall offloading cost, as this will bring more savings to the processing time and battery power.

In the same vein, a subsequent publication that aims to overcome the overhead cost introduced by the partitioning algorithm has proposed a Web-browsing sharing mechanism to take over from the partitioning algorithm [57]. As shown in Figure 2-14, the author has achieved this by (1) using web browsers that support HTML5 protocol, which allows a parallel sharing of sub-tasks by default, using Web-workers and Web RTC to communicate among assisting computing devices, so to share sub-tasks with each other. (2) Implementing a peer-to-peer middleware protocol that allows devices to peer together through WLAN, using node.js and peer.js [58]. It uses a laptop as the main server, where the assisting computing devices have to communicate to the server and register their peer Id using Web RTC so that the client/host-MSD will have a list of available assisting computing devices. Upon a successful connection, a peering request is sent to the available assisting devices to start sharing the sub-tasks among devices using Web-workers. Then the devices process the requested sub-tasks and send the results back to the host-MSD, and the process is terminated. The author claimed that the proposed mechanism has achieved a similar reading/trend of processing time and battery power consumption without the need of a partitioning algorithm, compared to executing all sub-tasks locally on the host-MSD or by executing all sub-tasks remotely on the CS.

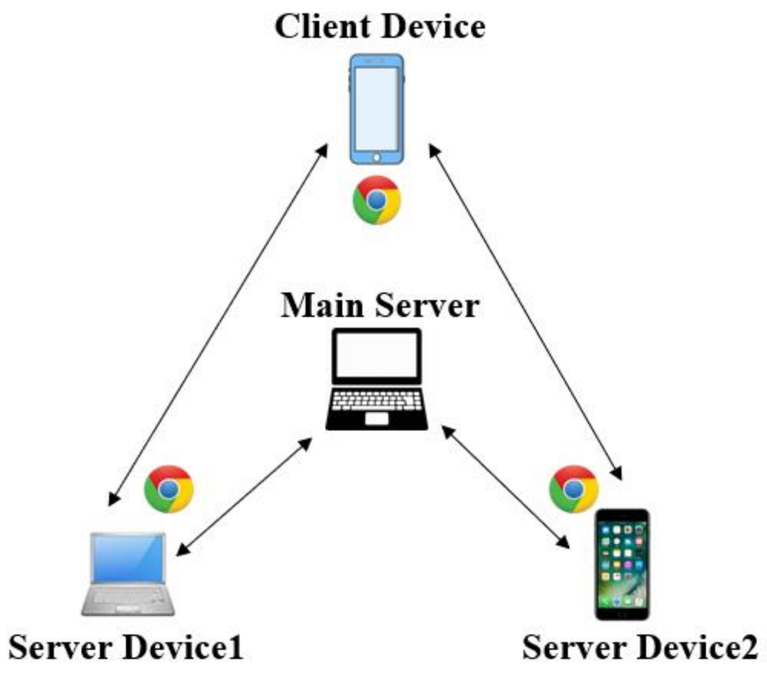


Figure 2-14. Web-browsing Sharing Mechanism

We have noticed from the experiments that the author did install a local server on the assisting device, with the help of peer.js, so to enable a client-to-server architecture. This mechanism is not practical, as it is based on a pre-setup of the overall infrastructure, therefore cannot be deployed in our solution. As well as, the concept of having to ensure the assisting computing devices to have the same browser is immature.

In the same vein, I have decided to briefly mention here some other associated partitioning algorithms for more information. The authors in [59] have proposed an adaptive power analysis to partition the execution of CIASs to sub-tasks, based on mathematical equations. The authors in [60] have proposed a cost trade-off algorithm that partitions the workload based on a cost optimisation equation, (i.e. a server speedup factor (F), assumes to be >1). The authors in [61] have implemented an auto-splitting algorithm to distribute the sub-tasks among the host-MSD and a CS, based on a history DB that includes information of sub-tasks previous processing time, consumed battery power and RTT latency between the host-MSD and the CS.

Reviewing the above publications have helped me to understand technically various partitioning algorithms, which have therefore led to develop and implement my own method. We have clustered a group of independent sub-tasks without affecting the execution order of the CIAS, and offloaded them as a package at one time rather than offloading sub-tasks individually, as will be detailed in Chapter 4.

2.3. A Review of Publications that Focus on Proposing AI-based Models

The last three years have witnessed a sharp rise in AI-based solutions performing intelligent and dynamic decisions/predictions for all sort of Apps, albeit with a great emphasis on image processing and pattern recognition. This trend has inspired us to review the literature on ML algorithms and AI-based models that might help in our implementation, so to intelligently facilitate the assisting-MSDs that can be chosen for each of our offloading sub-tasks. I have chosen to include six publications that focus on proposing relevant AI-based models and algorithms to our solution. These publications have helped me to build my own AI-engine, so to maximise the performance and benefit of our on-the-go ECR novelty, as will be detailed in Section 5.1.

The first of these publications has focused on providing an intelligent decision to measure the load balancing of VMs deployed on a different set of servers in a DC. It proposes a prediction model to select the most appropriate VM usage for any workload, by monitoring the actual CPU utilisation of several VMs over time and learn from the trend of usage [62]. The authors have achieved this by using a DL algorithm to train, test and estimate the accuracy of their AI-model. They have implemented the following two algorithms (1) a Deep Belief Network (DBN) [63], that extracts VM features, (CPU utilisation, workload, and memory utilisation), from observing the behaviour of VMs over 10 days. (2) A Regression algorithm that uses the extracted features from the first step as inputs then fine-tunes the model in a supervised manner to predict VMs workload in the output layer. The CPU utilisation of 20 VMs was monitored for 10 days at different time intervals, using a CloudSim Simulator [64]. A DBN was formed to decide on the number of hidden layers and units in each layer, the value of the CPU utilisation was normalised between (0,1) and fed into the model to predict the workload for each VM. Figure 2-15 shows a structure diagram of the DL network including the above two algorithms and the three hidden layers.

To measure the accuracy of their prediction model, the authors have plotted the actual and predicted CPU utilisation, as shown in Figure 2-16. The results of the prediction have improved by 1.3% for a single VM workload prediction and by 2.5% for multiple VMs workload prediction, compared to simple NN model and multilayer NN model and other such widely used prediction models.

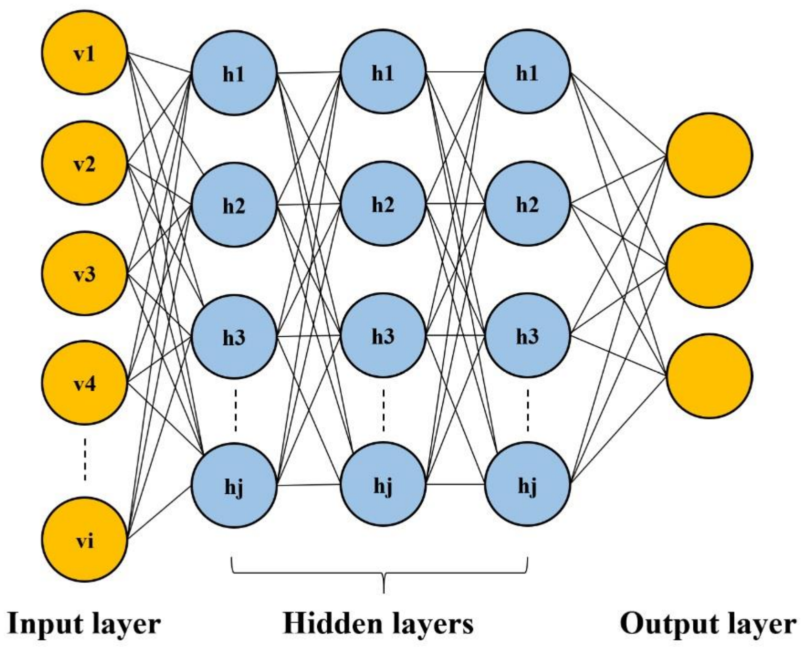


Figure 2-15. Proposed DL-Network Structure

We believe that the concept of using AI to make intelligent and accurate decisions is commendable, and therefore this publication was a seed for understanding the effort, training process, comparative results and inference accuracy when developing an AI-engine so to make intelligent decisions. This has helped us to visualise the enhancement that such an engine will bring for the offloading decisions in our solution.

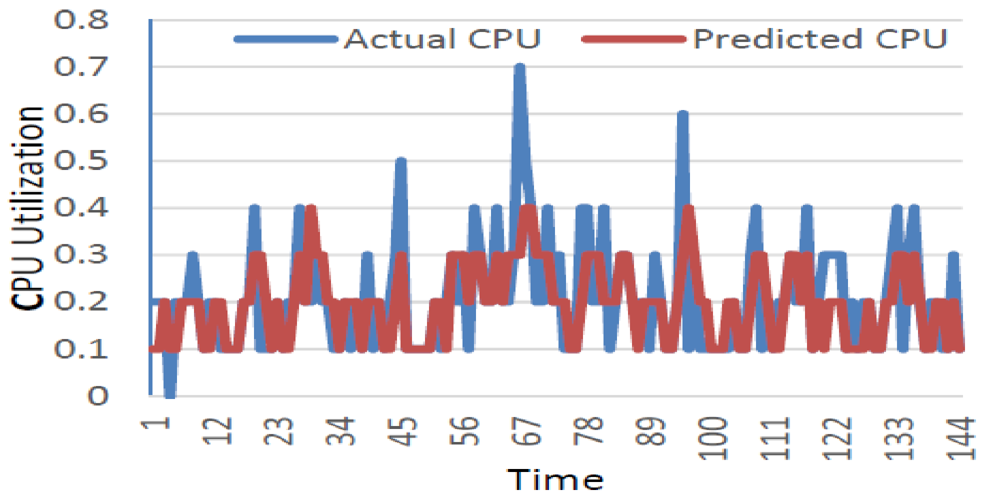


Figure 2-16. Actual and Predicted VM Workload

For user's location tracking App, a team of researchers have used a DL algorithm to select suitable nearby devices in close proximity to the host-MSD, by observing their current movement [65]. They have achieved this by implementing (1) a Deep Auto Encoder (DAE); a powerful and efficient algorithm that can extract/learn more features compared to typical prediction algorithms, (e.g. Principle Component Analysis (PCA)). The authors have implemented a 4 layers network that can learn/predict the location of devices by observing their typical movement for a certain time, as shown in Figure 2-17. An App was developed to collect GPS signal every 30 seconds for a group of volunteers using their SPs on their 8-hour daily routines. The data was collected for 38 days, as 30 days for training and 8 days for testing, all the collected data then input into the DAE network. (2) A Restricted Boltzmann Machine (RBM) algorithm with 2 NN layers to learn from the output of DAE algorithm and reduce the dimension of the network, by using backpropagation to fine-tune the network, so to make an accurate prediction. Their experiments results show that the mean square error of the model is relatively small when using the DL algorithm, compared to using the PCA algorithm that was 3x higher.

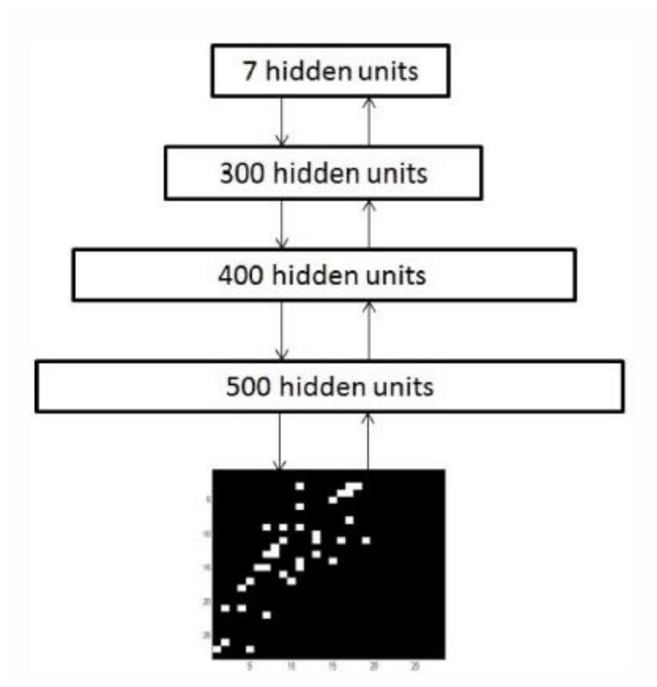


Figure 2-17. DL Prediction Model Architecture

Although this model can help with identifying possible nearby devices to use, the authors did not examine nor conduct real offloading scenarios to evaluate the performance or overhead cost of deploying such model in a real sharing environment. Their implementation is based on a DL algorithm and the results have proved to achieve better performance and lower error rate compared to some ML algorithms such as the PCA. This has encouraged us to investigate further DL algorithms as a potential implementation for our solution, as will be detailed in Section 5.2.1.

The authors in [66] have proposed to deploy the AI-based model in a CS, so to handle exabytes of data that have been generated from deploying classification models to deal with bigdata Apps. They have achieved this by implementing (1) a DNN computing model hosted in the Cloud, so to enable the training and testing of streams of audio/video data coming from various end-devices/sensors in a supervised and unsupervised manner, which otherwise cannot be hosted in an ES positioned on the end of a computing network. This DNN model has been further enhanced to offer the ability to test multiple algorithms, (i.e. DAE, DBN and RBM), in parallel and select the best one among them, that is likely to perform well for a specific type of data. (2) Using a linear algebra mathematical model to decide on the number of layers, hidden units and inputs that each algorithm can take. It can automatically increase/decrease these parameters to form a network that fits the uploaded data so to match the type of problem the model aims to solve. After the training and testing process is completed, the best network structure is then selected and therefore it is shared with the ES, where end-devices can seamlessly invoke such model to verify the new upcoming data, as well as to fine-tune the model for further executions and/or update the network structure if needed. To evaluate the performance of this implementation, MNIST dataset was used in the experiments containing 6000 labelled samples, and the model was examined to classify the samples, using 3 RBM layers. To expose the model to different learning rate, the authors have split the data to various training and testing datasets as follows: (i) 25% for training and 75% for testing, (ii) 50% for training and 50% for testing, and finally (iii) 75% for training and 25% for testing. The results show up to 93% accuracy has been achieved, compared to 54.7% for a relevant model that uses an ML algorithm.

The idea of hosting the AI-engine in a CS to train and test the dataset is important. We concluded that our implementation will be using a CS for hosting. Furthermore, we inspired to implement our AI-engine using a DNN model, as will be detailed in Section 5.2.1.

The concept of the above-proposed model has been enhanced by the work in [6], that introduces a resource scheduling algorithm to distribute CIAS sub-tasks among multiple ESs and predict the processing time of each sub-task that executes on a certain ES, using an ML algorithm. They have achieved this by (1) implementing a prediction method to model a relationship between the actual and predicted processing time of each sub-task that executes on the ES, using a linear regression algorithm that considers previous executions. (2) Developing a load balancing module based in the Cloud that monitors available ES's and redirects the processing load if a certain ES exceeds a threshold of 70%, using Grafana tool [67]. After the process of steps, 1 and 2 is completed, and the decision is formed, then (3) offloading sub-tasks from the host-MSD to the ESs and retrieve the results upon completion. To evaluate the performance of the proposed algorithm, the authors used TensorFlow platform to deploy a Google-Cluster dataset that contains several Cloud and Edge nodes. A 6 layers regression network was used with a total of 97 neurons in the training phase to predict the expected sub-tasks processing time. Their achieved results show a decrease of 38% in the processing time, compared to executing sub-tasks by a single ES, also the model shows accurate prediction, with a small error rate between the actual and predicted processing time.

The concept of this model has inspired us to introduce a scheduling model to allocate the most intensive sub-task to execute on the best assisting-MSD, (i.e. that has the highest processing resources, lowest load and best connectivity type, etc.), using a performance scoring algorithm, as will be detailed in Section 5.2.4.

Further to the DL, DNN and ML models implemented in the previous publications, this contribution proposes to use a DRL algorithm claiming it will achieve more accurate offloading decisions in a dynamic/stochastic environment [14]. It claims that DRL can minimise the overall cost overhead introduced by traditional ML-based algorithms that use complex mathematical computation equations. The authors have designed three levels offloading model, where the first level is an SP that acts as a host-MSD and has CIAS sub-tasks to execute, and so, it requires the processing support for executing its sub-tasks. In the second level, multiple ESs offering limited processing resource to the host-MSD when needed. Finally, a CS in the third level is available to support and offer its unlimited resources to the host-MSD if needed. They have implemented a Deep Q-network algorithm [68] that can intelligently decide on where to execute the sub-tasks, (level 1, 2 or 3), based on learning from previous experience, (i.e. unlike ML/DL algorithms that learn from training data). The algorithm randomly executes the sub-tasks on all the available devices, (i.e. host-MSD, CS and ESs), then it measures the cost overhead of executing these sub-tasks on each device. This process is continuous, it compares the cost overhead gained from all the devices continually and selects the best device that has consumed the lowest overhead. The sub-tasks can seamlessly be offloaded and executed on all the devices, irrespective of sub-tasks complexity, offloadability and granularity. The algorithm has been formulated as an MDP model, and therefore the authors have implemented the State, Action and Reward functions, so to minimise the weighted summation of the average sub-tasks delay and battery power consumption.

This contribution has shown that deploying a DRL algorithm can improve the offloading decision, it reduces the average processing delay of sub-tasks by 30% and the overall cost by 23%. Albeit, we think that deploying a DRL algorithm might not be suitable for our solution because the performance of the DRL algorithm is based on experience and so it can be unpredictable from the outset if the initial prediction results are not accurate.

Reviewing the above publications have helped me to introduce and implement my Cloud-based AI-engine. It achieves this by (1) recruiting available assisting-MSDs that are willing to help the host-MSD executing the processing of CIAS sub-tasks in a sharing environment. (2) Providing the host-MSD with decisions of the best scenario to offload the sub-task, so to reduce the processing time and battery power consumption. (3) Scheduling the intensive sub-tasks to be allocated to the assisting-MSD that has a powerful processor and high-capacity battery power, using a performance scoring algorithm. As will be detailed in Chapter 5, our implementation was eventually led to the publication of our third proposal solution DEO; a Smart Dynamic Edge Offloading Scheme.

3. Evaluation of Relevant Offloading Frameworks and Techniques

Despite the continuous improvements to MSDs resources, (e.g. processing, battery capacity, display, etc.), yet the battery technology is still lacking behind to cope with the fast roll-out of CIASs. Executing such CIASs on the host-MSD will reduce its execution time and consume the battery power, due to the compute intensity, network throughput and RTT latency of such Apps/Services. i.e. Executing a Face Detection App on a Samsung Galaxy A20 with 4000 mAh battery capacity will drain its battery power in less than one hour [69]. Offloading and executing all/part of the CIAS workload from the host-MSD to a CS will no doubt alleviate the above shortfalls, as detailed in our literature in Section 2.1.

This chapter demonstrates the experiments and tests which I have conducted in the early stages of my research to (i) understand the depth and breadth of the offloading process, (ii) evaluate some offloading frameworks and (iii) get hands-on experience so to help me build and implement my novelties, as follows:

1. Develop a typical client-to-server architecture between a host-MSD and a CS, (Section 3.1).

2. Simulate and deploy Cloud DCs and VMs, so to learn how to scale Cloud processing resources up/down on-demand using three load balancing algorithms, this has enormously helped me to understand the importance of how the decision of where to offload will affect the overall offloading decision, (Section 3.2).

3. Re-evaluate four offloading frameworks using various wireless networks, so to (i) calculate the execution time and battery power consumption costs when the host-MSD offloads CIAS sub-tasks to the CS. (ii) Measure the transmission time of uploading sub-tasks between two MSDs, (Section 3.3).

3.1. A Client-to-server Architecture between a host-MSD and CS

To build a typical client-to-server architecture, I had to implement a scenario to enable sending and receiving of packets between nodes/end-devices, this was important to establish a communication channel, so to offload sub-tasks between the host-MSD and CS. I have used the NS3 [70]; an open-source simulator that mainly focuses on forming networking scenarios, (e.g. Wi-Fi and LTE), Figure 3-1 shows NS3 stack when sending/receiving packets/acknowledgements between 2 nodes.

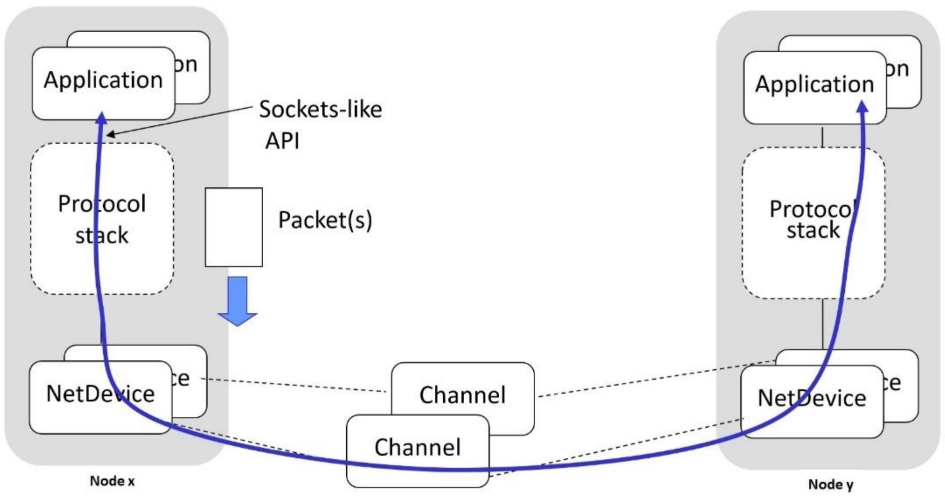


Figure 3-1. NS3 Stack: Sending and Receiving Packets between 2 Nodes

3.1.1. A Simple Test Scenario

In this test, I have created two nodes, (x and y), in which node x is customised as a client, (so to act as the host-MSD), while node y is customised as a server, (so to act as the CS). To formulate the network, I have simulated multiple APs connected through a wired ethernet link and can be accessed through Wi-Fi or cellular. Then, I set the simulator attributes as follow, (10-100 Mbps data rate, 10-20 ms delay rate, 1 KB packet size and assigned IP address), as shown in Figure 3-2. The client sends a packet to the server in a given time, the server receives the packet and sends an acknowledgement back to the client. The server starts from 1 s and waits until 10 s for the client to start sending packets. The client starts from 2 s and starts sending packets to the server, then it waits for 10 s to receive acknowledgements from the server. Upon a successful connection, the devices start to send/receive packets among them, I have also simulated more nodes so to enable multiple clients and servers communicating together in a sharing environment.

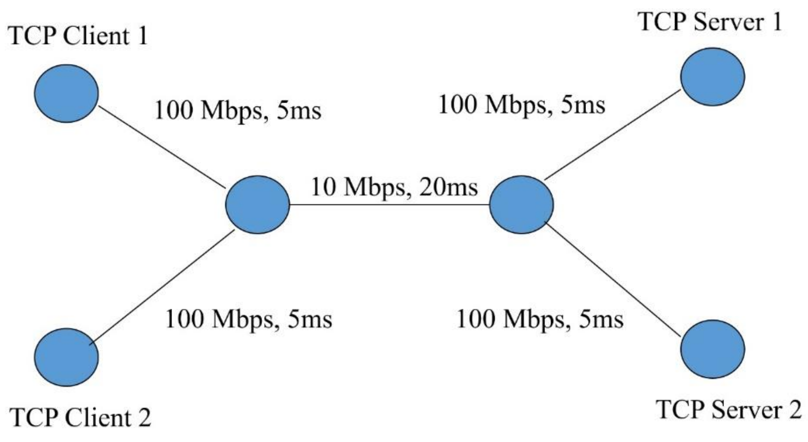


Figure 3-2. A Client-to-server Architecture

Figure 3-3 shows the test scenario using NS3 visualise tool, (NetAnim) [71], the client sends a packet of 1 KB to the server at 2 s, the server receives the packet successfully at 2.1 s. Then the server sends an acknowledgement to the client at 2.19 s, and the client receives the acknowledgement from the server at 2.31 s.

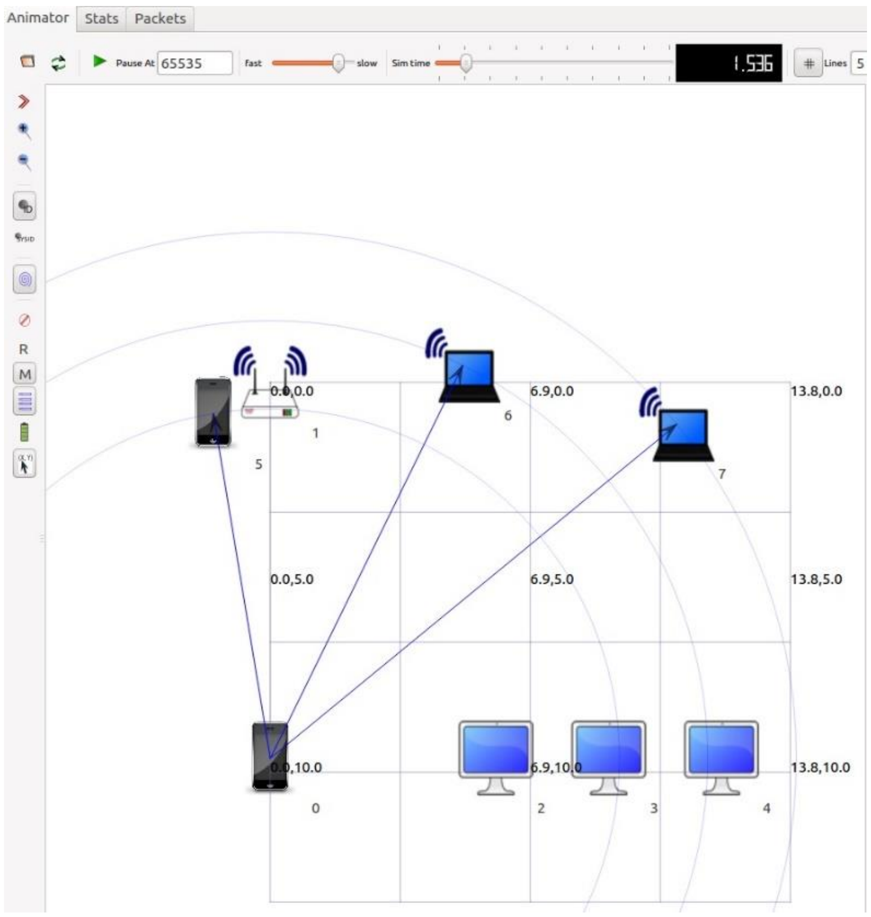


Figure 3-3. A Client-to-server Scenario

I must admit that this was my first test scenario, which was conducted in the early stages of my research, so to implement a client-to-server architecture, where end-nodes are only used to send and receive packets, (i.e. nodes do not perform any processing). This was important to me to start learning how to simulate nodes and servers for communication between them, so to help me later when building my solution. Albeit, our implementation is mainly based on a peer-to-peer architecture, so to network among a group of assisting-MSDs, as will be explained in Section 4.1.4.

3.2. Simulating Cloud DCs and VMs using Load balancing algorithms

To research the concept of virtualisation and scalability from a Cloud provider point of view, I have decided to understand the technical aspect of load balancing, resource allocation and how to scale up/down computing resources on-demand, (e.g. DCs, VMs and other processing resources). This was important to my work as my solution involves parallel processing, resource sharing and allocation, so understanding the above was necessary to implement my own work.

We have conducted various tests to simulate multiple users, (a user here represents a host-MSD), offloading sub-tasks to a DC in the Cloud that contains multiple VMs/servers, while measuring DC response time, data transfer and VMs cost. The sub-tasks are forwarded from the host-MSD to the Cloud DC using a broker. The broker role here is to (i) decides which DC/VM to use when multiple sub-tasks are received, (ii) monitors the current load and capacity of the servers and (iii) migrates sub-tasks from the overloaded servers to the unloaded servers, using three load balancing algorithms, as will be explained in Section 3.2.2.

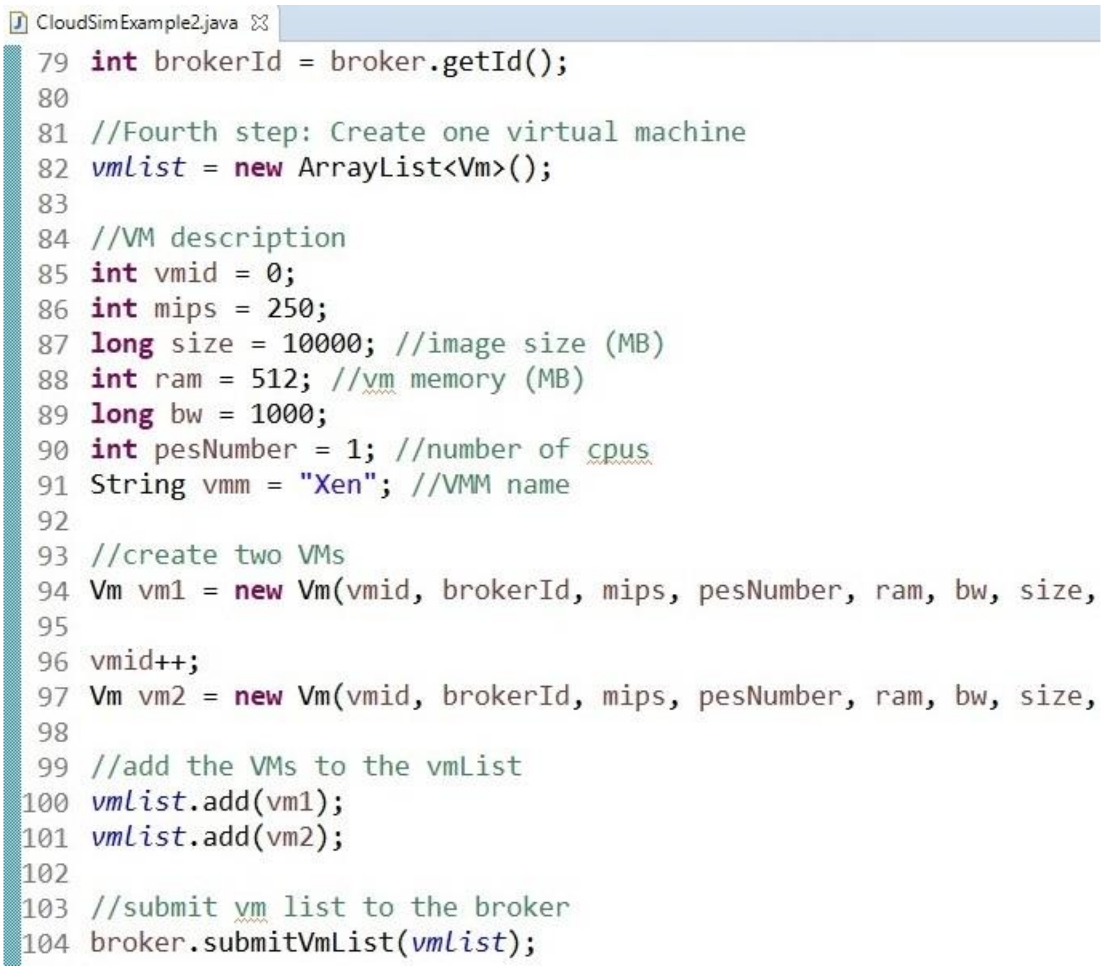
3.2.1. Simulation Environment

I have used a CloudSim Simulator [72], as being widely preferable in simulating CC architecture and scenarios. CloudSim is a java-based open source simulator that offers libraries, packages and classes, so to virtualise and manipulate CC infrastructure, (e.g. DCs, servers and other computing processing resources). I have launched the CloudSim library using Eclipse Platform and imported a CloudAnalyst GUI [73], which was developed on top of the CloudSim stack, so to graphically visualise and distribute the deployments of users and DCs across six regions. i.e. Mainly is the typical deployment of servers used by Cloud providers, (e.g. AWS EC2) [74].

3.2.2. Experiments and Evaluation

The tests aimed to measure the overall DC response time, VMs processing time, latency and VM cost when users offload sub-tasks to a server in a Cloud DC. As well as to monitor the DC load balancing using brokers to route and manage the load among VMs, using three load balancing algorithms. The broker invokes/terminates VMs when the load on a certain VM exceeds a threshold of 70%.

I did program a simple test scenario to create 2 DCs and 2 users that offload multiple sub-tasks of 100 KB to the DCs, as shown in Figure 3-4. I have varied the number of VMs for each DC from (1-25), so it can scale up/down on-demand, using a Cloudbus Library, each VM has (1000 MB image size, 250 Mips and 512 MB Memory). I have created and assigned a broker based on a simple assessment to route the load to the nearest DC, the test was repeated 5 times, while the results were collected and analysed, as will be detailed next in Section 3.2.3.

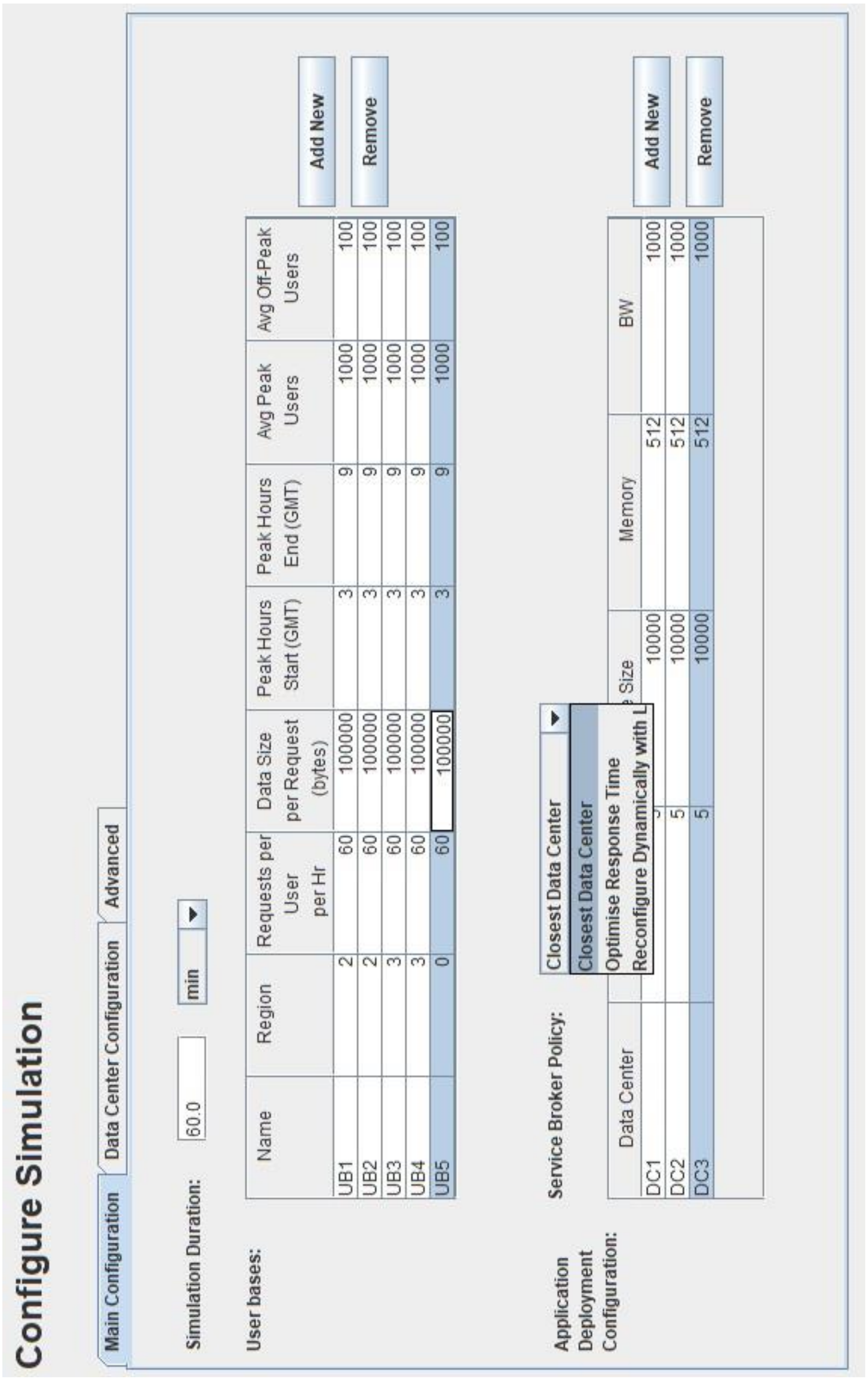

CloudSimExample2.java

```
int brokerId = broker.getId();

//Fourth step: Create one virtual machine
vmlist = new ArrayList<Vm>();

//VM description
int vmid = 0;
int mips = 250;
long size = 10000; //image size (MB)
int ram = 512; //vm memory (MB)
long bw = 1000;
int pesNumber = 1; //number of cpus
String vmm = "Xen"; //VMM name

//create two VMs
Vm vm1 = new Vm(vmid, brokerId, mips, pesNumber, ram, bw, size,

vmid++;
Vm vm2 = new Vm(vmid, brokerId, mips, pesNumber, ram, bw, size,

//add the VMs to the vmList
vmlist.add(vm1);
vmlist.add(vm2);

//submit vm list to the broker
broker.submitVmList(vmlist);
```

Figure 3-4. Simulating a Test Scenario of 2 DCs with Multiple VMs

I have used CloudAnalyst to simulate the three load balancing algorithms, (i.e. Nearest DC, Optimize DC response, and Reconfigure dynamically the load among DCs), so to monitor and handle the load among the available VMs in a certain DC. The broker controls the traffic routing between users and DCs, so to decide which DC should be allocated to process the sub-tasks, as follows. (1) The Nearest DC algorithm calculates the quickest path among the available DCs based on the network latency, (i.e. it selects the DC with least network delay among others), using proximity-based routing algorithms [75]. (2) The Optimise DC response algorithm monitors all DCs and selects the DC that is likely to complete the assigned sub-tasks in less time than the other DCs, by monitoring the number of VMs available and the load on each VM. (3) The Reconfigure dynamically the load among DCs algorithm is a combination of the Nearest DC and Optimise DC response, i.e. it uses a VM manager to map a rational relation between the two, so to calculate the current processing time of each VM against the best processing time it has ever achieved [76].

I have also simulated a dynamic scenario where multiple users, (deployed in different regions), are offloading 100 sub-tasks to multiple DC using the above three algorithms, as shown in Figure 3-5 and Figure 3-6, while measuring the overall DC response time, VMs processing time, latency and VM cost. I have also manipulated the transmission rate and delay rate among DCs, so to reflect real values used by some Cloud providers, (e.g. AWS [77] and Microsoft Azure [78]).



Simulation Duration: 60.0 min

User bases:

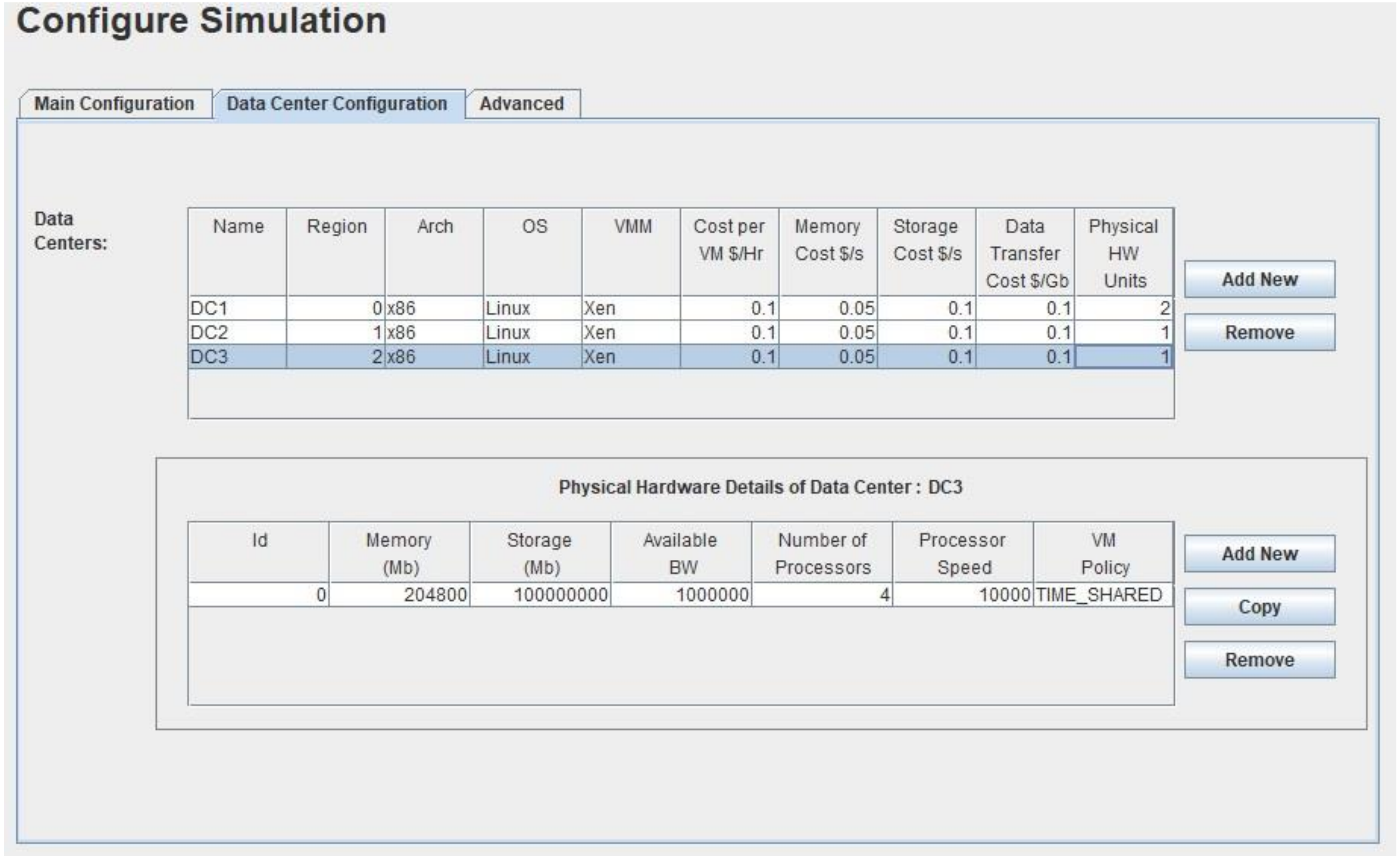
Name	Region	Requests per User per Hr	Data Size per Request (bytes)	Peak Hours Start (GMT)	Peak Hours End (GMT)	Avg Peak Users	Avg Off-Peak Users
UB1	2	60	100000	3	9	1000	100
UB2	2	60	100000	3	9	1000	100
UB3	3	60	100000	3	9	1000	100
UB4	3	60	100000	3	9	1000	100
UB5	0	60	100000	3	9	1000	100

Service Broker Policy: Closest Data Center

Application Deployment Configuration:

Data Center		Size	Memory	BW
DC1	5	10000	512	1000
DC2	5	10000	512	1000
DC3	5	10000	512	1000

Figure 3-5. The Deployment of Multiple Users


Configure Simulation

Data Centers:

Name	Region	Arch	OS	VMM	Cost per VM $/Hr	Memory Cost $/s	Storage Cost $/s	Data Transfer Cost $/Gb	Physical HW Units
DC1	0	x86	Linux	Xen	0.1	0.05	0.1	0.1	2
DC2	1	x86	Linux	Xen	0.1	0.05	0.1	0.1	1
DC3	2	x86	Linux	Xen	0.1	0.05	0.1	0.1	1

Physical Hardware Details of Data Center : DC3

Id	Memory (Mb)	Storage (Mb)	Available BW	Number of Processors	Processor Speed	VM Policy
0	204800	100000000	1000000	4	10000	TIME_SHARED

Figure 3-6. The Deployment of Multiple DCs

3.2.3. Results and Discussion

We have conducted these tests to examine various scenarios/algorithms while launching multiple users and DCs in different regions (0-5), so to offload 100 sub-tasks of 100 KB size generated by the users in a dynamic environment. We have measured the overall DC response time, VM processing time, data transfer and VMs cost. We have varied the number of users between 1-5 using 2 DCs with a total of 25 VMs per DC with each user generating 100 sub-tasks with a size of 100 KB for each sub-task, as explained in Section 3.2.2. Then the broker decides which DC to use based on monitoring the load on each DC using three load-balancing algorithms, and therefore it terminates/invokes more VMs when the load on each VM exceeds a threshold of 70%. Figure 3-7 shows the average DC response time when the users offload the sub-tasks to the DC.

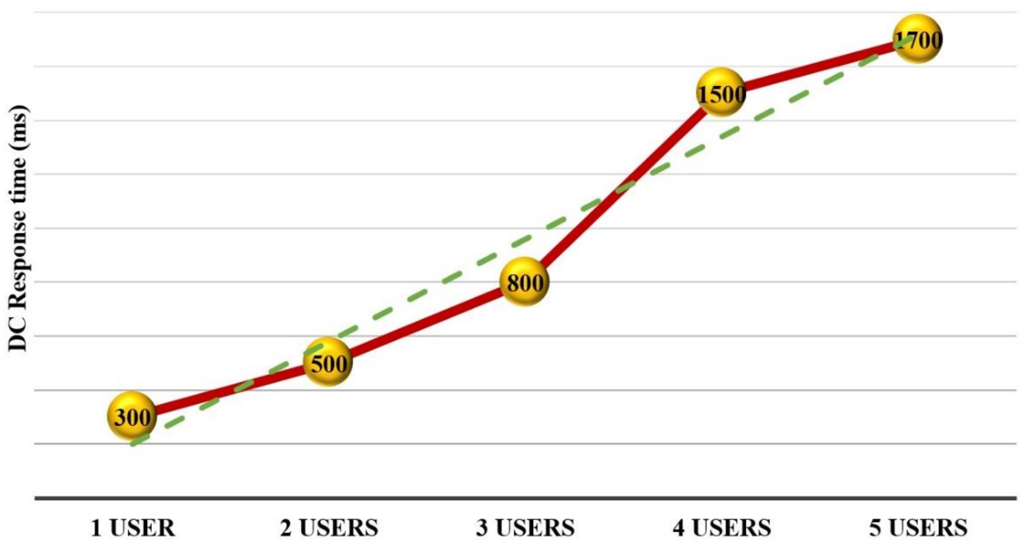


Figure 3-7. DC Average Response Time when the Number of Users Increases

It is clear that when the number of users accessing the DC increases, the response time increases rapidly. This is because when multiple users join the network and generate more sub-tasks, more processing is required by the DC. We have used AWS to generate EC2 server in the Cloud [7], we have manipulated the number of users between 1-5 accessing the server at the same time, the green dotted line in the figure represents the expected response time from user 1 to user 5 recording a linear trend when the number of users increases.

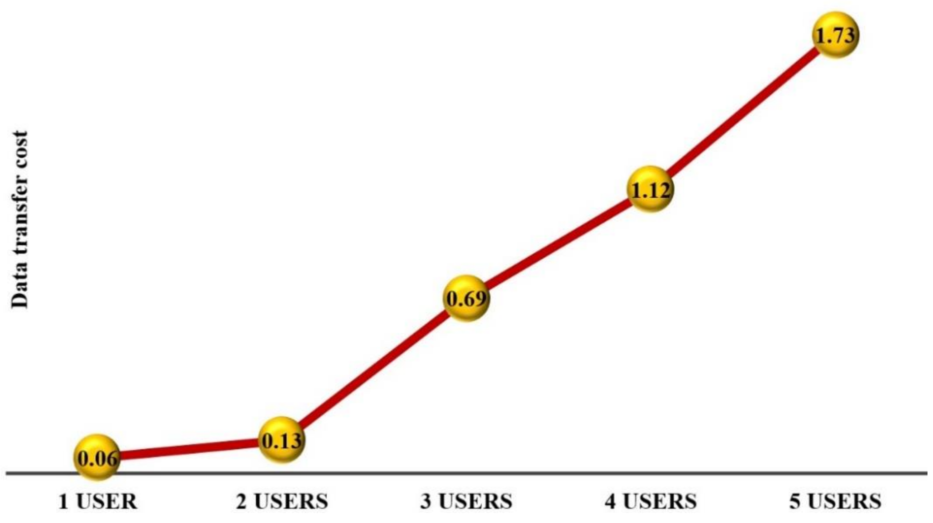


Figure 3-8. Data Transfer Cost when the Number of users Increases

Similarly, Figure 3-8 shows that the data transfer cost of the DC when more users join the network and offload the sub-tasks continually, it has increased by 1.6x following the same trend of the DC average response time. On the contrary, Figure 3-9 shows the DC response time has dropped sharply by 87% when more processing resources are assigned to DC, using the dynamic load balancing algorithm. The nearest DC that is likely to execute the sub-tasks in less time is selected, compared to the other DCs. We have used AWS to generate EC2 server in the Cloud and we assigned multiple VMs/instances between 1-25 allocated to the server, the green dotted line in the figure represents the expected response time when the number of VMs increases. Hence, it is important to decide on the number of resources allocated to the CS.

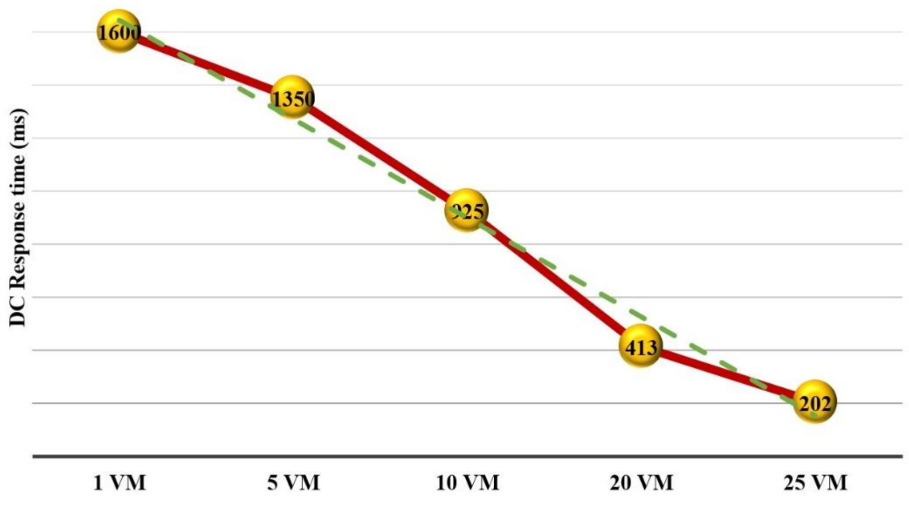


Figure 3-9. DC Response Time when more VMs are assigned to a DC

Figure 3-10 shows the VMs cost has increased by 9x when we have increased the number of VM from 1 to 25 in a single DC.

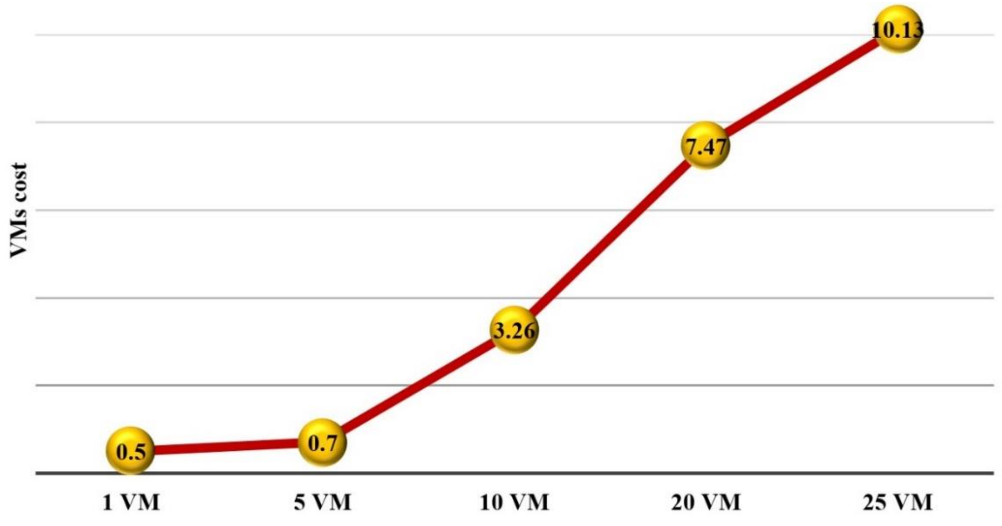


Figure 3-10. VMs Cost more VMs are assigned to a DC

There is no doubt that there should be a balance between deciding how many sub-tasks are allocated to a single DC, against the number of processing resources available for each DC. The results show that the dynamic algorithm can achieve the best user's satisfaction in terms of response time and VM cost.

From testing the above, we have concluded that a dynamic decision of where to offload is vital, considering the load on each assisting device, and how many sub-tasks to allocate to each device. Especially, when the complexity/granularity of sub-tasks increases, in such case more processing resources are required for each assisting device. Also, the location of the assisting DC and CS is important, so to eliminate the long RTT between the host-MSD and the CS.

3.3. Evaluating Relevant Offloading Frameworks

We have mainly focused here on conducting simple test scenarios to get hands-on learning experience, so to develop a framework that offloads the processing of CIASs workload from the host-MSD to a CS, using Wi-Fi and 3G, and retrieves results upon completion.

Firstly, I have conducted a pilot case study to re-evaluate four offloading frameworks, so to calculate the typical offloading cost equations for execution time and battery power. Secondly, I have implemented a test to offload a coarse-grained N-Queens Puzzle, as being the CIAS to be tested, from the host-MSD to a CS. Finally, I have measured the transmission time of offloading a sub-task of 100 KB between 2 MSDs, using various wireless networks.

3.3.1. A Pilot Case-study

In this study, I have re-evaluated four offloading frameworks, which focus on offloading a sub-task, (around 30-100 KB) from the host-MSD to a CS, so to calculate the average execution time and battery power consumption. The study focuses on the tested CIASs, bandwidth link, network availability, host-MSD and CS computing processing resources.

1. Chess game CIAS was tested locally on the host-MSD, (a Nexus One SP that has Qualcomm 8250 Snapdragon and 1 GHz CPU), and remotely on a CS, (an AWS EC2 server operates in the Cloud). Sub-tasks were generated and offloaded, (around 30-100 KB), between the host-MSD and CS while measuring the average execution time and battery power consumption, using Wi-Fi, (latency is set to 100 ms), and 3G, (latency is between 200 to 400 ms). The achieved results show that using Wi-Fi to offload a sub-task from the host-MSD to CS consumes an average of 77 ms, compared to 1622 ms for 3G. Also, executing the sub-task remotely on the CS consumes 480 mw, compared to 1150 mw for the local execution by the host-MSD alone. It is clear that offloading with Wi-Fi has a better outcome compared to 3G network, this is because 3G is slower and has limited coverage in some indoor locations [79], which will eventually have a negative impact on the offloading process. Also, the remote execution by the CS always outperform the local execution by the host-MSD, this is because the CS have unlimited processing resources. The overall findings show that using Wi-Fi to offload have achieved around 8.6x execution speed-up and 30% battery power saving, compared to 1.59x and -85.2% for the 3G network.

2. Face detection CIAS was tested locally on the host-MSD, (a Motorola SP that has an ARM A8 processor and 600 MHz CPU), and remotely on a CS, (a PC that has Quadcore 2.83 GHz). The server connects through an ethernet cable to the internet with a data transmission rate of 100 Mbps. The complexity of face detection is to detect faces from a 10 s video duration. Sub-tasks were generated and offloaded, (around 30-100 KB), between the host-MSD and CS while measuring average execution time and

battery power consumption, using a Wi-Fi network. The achieved results show that executing the sub-task locally by the host-MSD requires around 14 minutes, compared to 60 s when executing the same sub-task remotely on the CS. Also, executing the sub-task locally on the host-MSD consumes a battery power of 0.9 mw, compared to 0.1 mw for the remote execution on the CS. The overall findings show that offloading and executing face detection sub-tasks on the CS have achieved 20x speed up and have saved 90% battery power saving, compared to local execution by the host-MSD alone.

3. Face detection CIAS was tested locally on the host-MSD, (an HTC Fuze SP that has Qualcomm 528 MHz, 288 MB RAM running Windows Mobile 6.5 OS), and remotely on a CS, (a PC that has dual-core 3 GHz CPU, 4 GB RAM, running Windows 7 OS). 10 Remote sub-tasks were generated and offloaded, (about 30-100 KB), between the host-MSD and CS while measuring the average execution time and battery power consumption, using Wi-Fi, (latency is set to 100 ms), and 3G, (latency is set to 220 ms). The achieved results show that it requires 18 s to execute the sub-task locally on the host-MSD, compared to 3 s and 5 s to offload and execute the same sub-task remotely on the CS for Wi-Fi and 3G respectively. The overall findings show that offloading and executing face detection sub-tasks on the CS have achieved 6.5x speed up and have saved 89% battery power saving using Wi-Fi, and 4.75x and 76% for a 3G network, compared to the local execution by the host-MSD alone.

4. Image search CIAS was tested locally on the host-MSD, (an HTC G1 SP that has Qualcomm MSM7201A and 528 MHz CPU), and remotely on a CS, (3.0 GHz Xeon CPU, running VMware ESX 4.1). The complexity of Image search is to access to the SP local memory and invoke face detection library, so to detect faces of 100 images and return the midpoint between eyes. Sub-tasks were generated and offloaded between the host-MSD and CS while measuring the average execution time and battery power consumption, using Wi-Fi, (latency is set to 69 ms using a data transmission rate of 6.6 Mbps), and 3G, (latency is set to 680 ms using a data transmission rate of 0.4 Mbps). The achieved results show that it requires 22.1 s time to execute the sub-task locally on the host-MSD, compared to 1 s to offload and execute the same sub-task remotely on the CS. The overall findings show that offloading and executing image search sub-tasks on the CS have achieved 20x speed up and have saved 20% battery power saving using Wi-Fi, and 16x and 14% for a 3G network, compared to local execution by the host-MSD alone.

3.3.2. Offloading Execution Time and Battery Power Costs

A good offloading framework will have to improve the host-MSD execution time and reduce its battery power consumption when executing CIASs. The offloading decision is essential and must be taken based on sub-tasks complexity, granularity and offloadability, as well as some other inputs, (e.g. the used wireless network, availability and computing resources of the host and assisting devices. Generally, offloading is mainly advantageous for sub-tasks requiring computationally intensive-processing that require few transfers between the host-MSD and the assisting devices, using fast network connectivity, (e.g. Wi-Fi), otherwise offloading is not beneficial [80]. This study calculates the local and remote execution time and battery power as being the main objectives when offloading the CIAS sub-tasks from the host-MSD to the CS. Albeit, we have developed a simple cost estimator to define a priority weight factor values so to normalise the cost equations and combine different units and scales, as will be explained in Section 4.2.2.4.

- **To Improve the host-MSD Execution Time:**

The time (T_{sp}) to execute a sub-task on the host-MSD is:

$$\mathrm{Tsp} = \frac{\mathrm{w}}{\mathrm{Ssp}} \tag{1}$$

Where S_{sp} is the speed of the host-MSD, (instructions per second), w is the instruction required to process a sub-task.

The time (T_s) to offload and execute a sub-task on the CS and retrieve the results back is:

$$\mathrm{Ts} = \frac{\mathrm{D}}{\mathrm{B}} + \frac{\mathrm{w}}{\mathrm{Ss}} \tag{2}$$

Where, S_s is the speed of the CS, (instructions per second), D is the data size (bytes), and B is the data transmission rate (Kbps). Offloading improves the host-MSD execution time only when Equation 1 is > Equation 2.

- **To Save the host-MSD Battery Power:**

The power (P_{sp}) to execute a sub-task on the host-MSD is:

$$\mathrm{Psp} = \mathrm{Pp} \times \frac{\mathrm{w}}{\mathrm{Ssp}} \tag{3}$$

Where, P_p is the host-MSD power consumption (watts).

The power (P_s) to offload and execute a sub-task on the CS and retrieve results back is:

$$\mathrm{Ps} = \mathrm{Pc} \times \frac{\mathrm{D}}{\mathrm{B}} + \mathrm{Pi} \times \frac{\mathrm{w}}{\mathrm{Ss}} \tag{4}$$

Where P_c is the power consumption of sending and receiving a sub-task, and P_i is the idle power of waiting (watts) for waiting for the results back. Offloading saves the host-MSD battery power only when Equation 3 is > Equation 4.

3.3.3. Experiments and Evaluation

Figure 3-11 shows our offloading scenario, we have implemented a scenario to offload the processing of N-Queens puzzle from the host-MSD to a CS for execution, using Android Studio 3.2.1 IDE platform [81]. We have used an iPad that represents the host-MSD, (a Samsung Galaxy Tab S2 SM-T710 that has Qualcomm Snapdragon 652 Octa-core 1.3 GHz, 3 GB RAM, and 4000 mAh battery capacity, running Android 7.0 OS), and a PC that represents the CS, (Intel Core TM i5 3.30 GHz, 1 GB RAM, running Android 6.0 OS installed on Oracle VM VirtualBox). The CS is connected to the internet using an AP with ethernet connection, and the host-MSD is connecting to the same AP using a Wi-Fi network. PowerTutor tool [82] is used to profile the host-MSD local processing resources, (e.g. battery level, CPU processing capabilities, and memory requirements), when executing N-Queens locally. Also, we have used a Speedtest tool to measure the internet speed and make sure the network is stable during offloading, using Wi-Fi with 216 Mbps uplink, 50 Mbps downlink, and 7 ms latency.

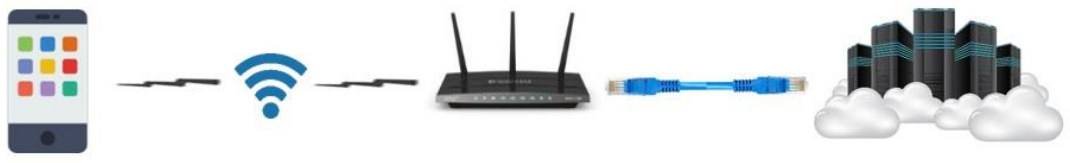


Figure 3-11. Offloading Scenario from the host-MSD to a CS

To evaluate our implementation, we have decided to use N-Queens CIAS as being one of the main benchmarks used in almost every offloading framework, it demonstrates sub-tasks complexity, offloadability, and granularity. N-Queens is a puzzle game of placing queens on NXN checkerboard, where queens do not threaten each other horizontally, vertically or dynamically. Executing N-Queens locally on the host-MSD consumes much processing time and drains battery power consumption. When N increases, more iterations need to be found, for N=12, the algorithm needs to perform 14200 iterations to find a suitable square to place a queen [35]. Details about N-queens sub-tasks complexity and granularity, as well as the intensive backtracking algorithm, will be explained in-depth in Section 4.2.1.1. In this test, we have performed a coarse-grained offloading scenario, (i.e. sub-tasks are not partitioned in a fine-grained manner). as the purpose of this test is to examine the connectivity networks used during the offloading process, not sub-tasks complexity, and granularity. Albeit, in Section 4.1.2, we did partition N-Queens to sub-tasks, so to execute in a dynamic environment among assisting-MSDs using a DFS algorithm.

We have used SDK and NDK Android Studio libraries to import and compile N-Queens puzzle on an Android Studio IDE, we used Async class function to exclude the GUI sub-task from all other sub-tasks. Then we created separate threads using Android multi-threading function, so to generate executable local and remote sub-tasks for the host-MSD and CS, using Java sockets to enable the client-to-server connection. After the connection is established, we then run N-Queens locally on the host-MSD and remotely on the CS while measuring the execution time and battery power consumption.

Furthermore, we conducted a simple test to measure the transmission time of offloading a sub-task of 100 KB between 2 MSDs using Wi-Fi, 3G and BT. In this test, we have used 2 SPs, (iPhone 6 Dual-core 1.4 GHz Typhoon ARM v8-based, 1 GB RAM, DDR3 memory, Li-PO 1810 mAh battery capacity, running iOS 10.2), so to represent a host and assisting-MSDs. CalcTool was used to calculate the transmission time to offload the sub-tasks from the host-MSD to the assisting-MSD.

3.3.4. Results and Discussion

Figure 3-12 shows the average execution time and battery power consumption required to execute the N-Queens locally on the host-MSD and remotely on the CS, using Wi-

Fi and 3G. Using Wi-Fi to offload the workload of N-Queens to the CS consumes less time compared to using a 3G network and the local execution by the host-MSD alone.

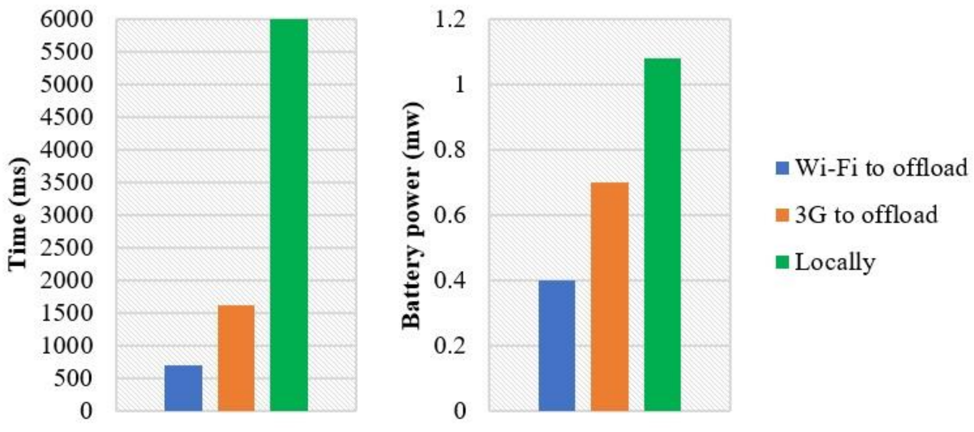


Figure 3-12. Local and Remote Execution Time and Battery Power

This is because 3G is slower than Wi-Fi and has higher latency, especially in congestion areas. Albeit, 3G results are still reasonable and promising to use for offloading. For the battery power consumption results, it consumes less power to offload the workload to the CS using Wi-Fi and 3G, compared to the local execution by the host-MSD alone. Table 3-1 shows the overall savings, using Wi-Fi to offload achieves a speedup of 16.27x and battery power saving of 57.25%, compared to 7.44x and 1.6% for 3G.

Table 3-1. The Overall Savings using Wi-Fi and 3G

Network Connectivity	**Sub-tasks**	**Speed UP**	**Battery Power**
Wi-Fi	30 – 100 KB	16.27x	57.25%
3G	30 – 100 KB	7.44x	1.6%

Figure 3-13 shows the transmission time of offloading a 100 KB sub-task from the host-MSD to the assisting-MSD, using Wi-Fi, 3G and BT. Using BT to offload the sub-task takes 363 ms, compared to 77 ms for Wi-Fi and 838 ms for 3G. We believe that using BT can also help to reduce the time taken to offload sub-tasks to nearby computing assisting-MSDs, so to form a local resource network, which eventually would bring in advance more achievements to add to the offloading process. Albeit, Wi-Fi is still preferable and mainly used in our implementations at all times.

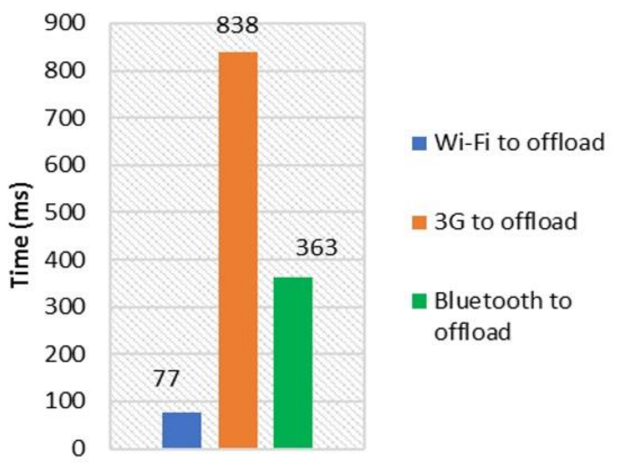


Figure 3-13. The Transmission Time to Offloading a Sub-task

We have concluded that (1) an efficient wireless technology to network to the assisting devices is very important, so to reduce the processing time and latency which will eventually eliminate the overall offloading cost. That is why we have extended our solution design to implement short-range peer-to-peer connectivity, as will be explained in Chapter 4. (2) The concept of parallel sharing is important to reduce the processing time, especially when multiple sub-tasks are offloaded at the same time. (3) The decisions of where to offload and which computing resource to select are essential, to address this, we have developed a Cloud-based AI-engine to advise on the best nearby MSD to select that has the highest processing resources and lowest load, as will be explained in Chapter 5.

4. Edge-side Implementation

International Data Corporation forecast predicts that there will be around 12 billion MSDs connected to Cloud services in 2021, (i.e. an increase of 10% year on year), 92% of the data generated by these devices are offloaded and processed in Cloud DCs causing an increase in network traffic to 16.1 Zettabytes in 2020 alone [83]. This increase in offloading CIASs sub-tasks to the CS, to be processed and then retrieving results back, has introduced high latency between the host-MSD and the CS, as shown in Figure 4-1. According to [29], offloading processing is also causing more battery power consumption as well as other networking/connectivity issues.

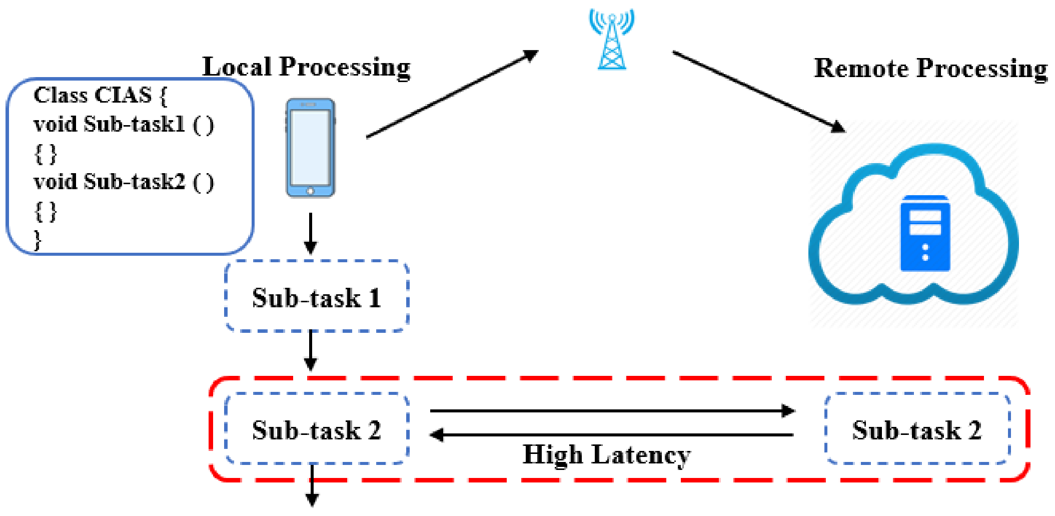


Figure 4-1. High Latency Occurred from Offloading CIASs Sub-tasks to a CS

This chapter describes our novel proposal that mainly addresses the high latency and battery power issues, as well as it details our Edge-side implementation, including:

1. Explanation of our architecture for the Edge Computing Resource (ECR), (Section 4.1).
2. The cooperative process of offloading from the host-MSD to the nearby devices including the processes of profiling, partitioning and execution, (Section 4.1).
3. The forming of the on-the-go resource network on the Edge, (Section 4.1.4).
4. The chosen four CIASs that we used in the experiments, (Section 4.2.1).
5. The six scenarios used for testing the performance of our ECR-engine, (Section 4.2.2).

Review of literature that helped us to arrive at the implementation of each of the above steps is detailed in the relevant part of Chapter 2. I also would like to direct the reader that the explanation of the AI-engine and associated functions will be detailed in Chapter 5.

4.1. Edge Computing Resource (ECR)

Figure 4-2 shows our smart solution which uses:

1. The Edge-layer to generate the ECR-engine that forms an on-the-go resource network to execute the processing of CIASs sub-tasks in parallel among the cooperative MSDs and retrieve results upon completion, using short-range p2p wireless network. The cooperative assisting-MSDs refer to nearby computing devices in the vicinity of the host-MSD, (e.g. SPs, iPads and PCs), that are willing to help and share their unused resources, (e.g. processing, battery and memory).

2. The Cloud-layer to host the AI-engine, (detailed in Chapter 5), to select the best device that has the highest resources and lowest load.

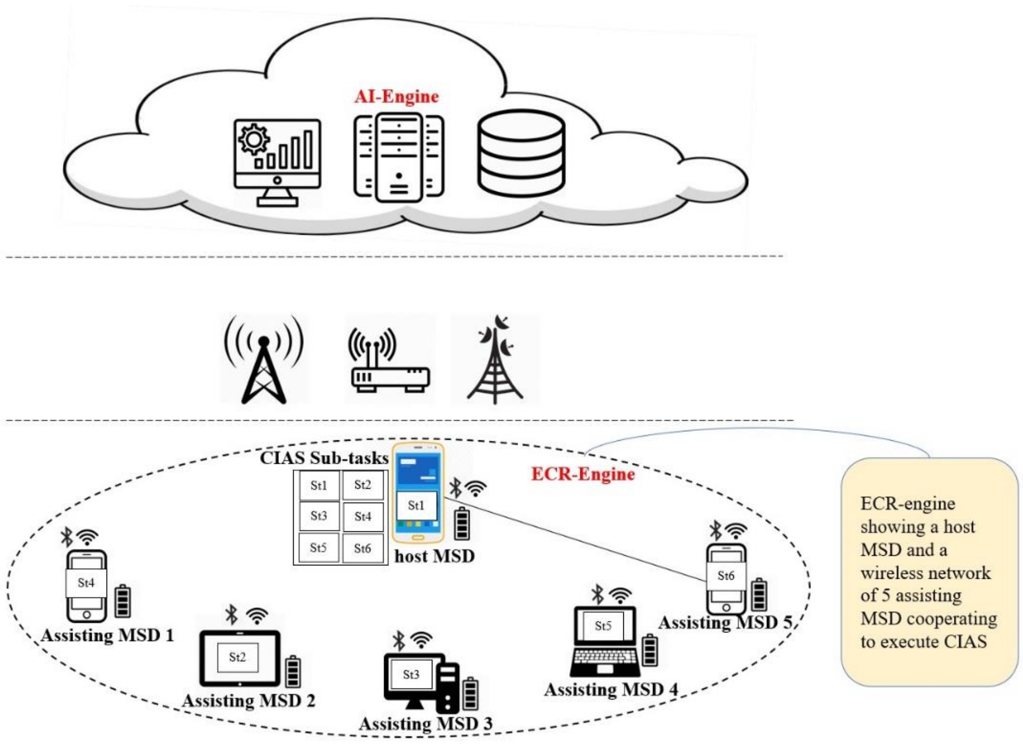


Figure 4-2. Our Smart Solution: ECR-engine and AI-engine

In conjunction with the Cloud-based AI engine, and for each CIAS, the process is (1) understanding the local resources of the host-MSD and deploying dynamic profiling of CIAS sub-tasks offloadability/complexity. (2) Implementing a DFS algorithm to partition the workload of CIAS to executable sub-tasks that can run on the assisting-MSDs. (3) Offloading and processing the sub-tasks on the cooperative assisting-MSDs in parallel and retrieve results. (4) The on-the-go formation of the resource network is based on a low-cost peer-to-peer wireless network, using a Nearby API networking interface. Figure 4-3 shows a flow chart of these steps, and the following sections give a detailed implementation of each of the 4 steps.

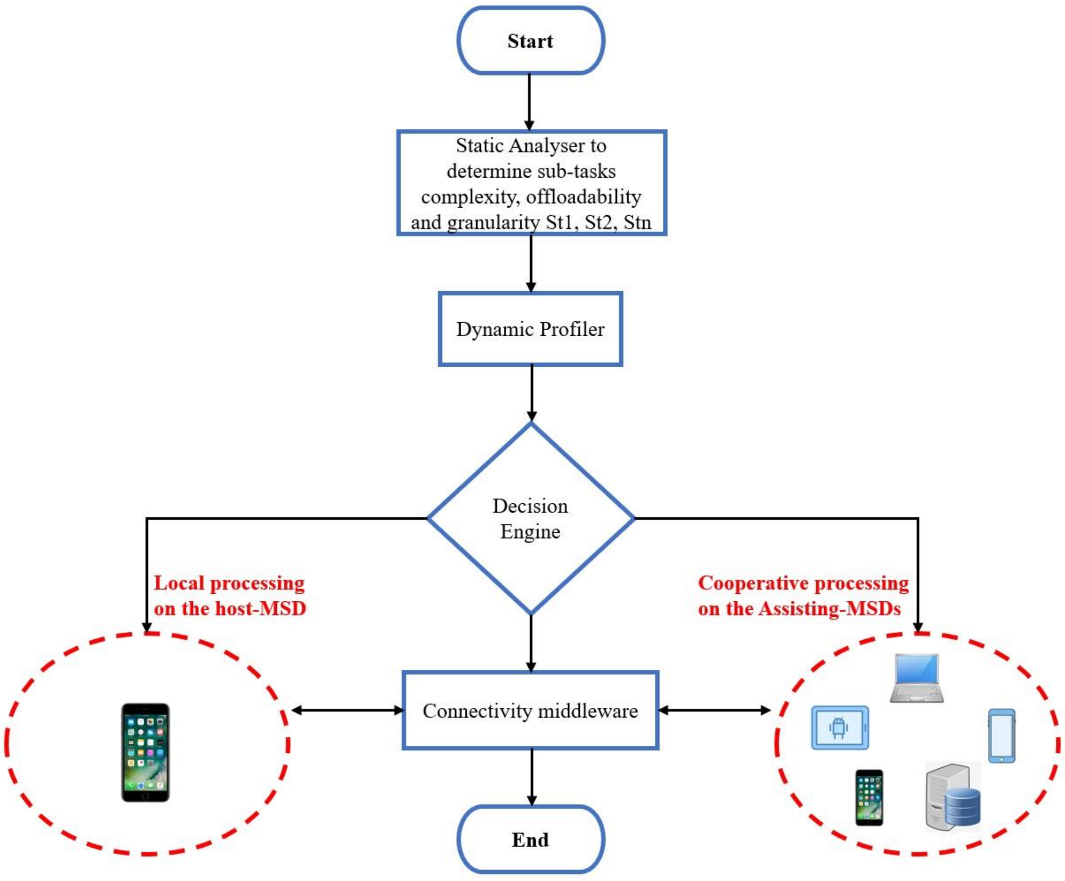

Start
Static Analyser to determine sub-tasks complexity, offloadability and granularity St1, St2, Stn
Dynamic Profiler
Decision Engine
Local processing on the host-MSD
Cooperative processing on the Assisting-MSDs
Connectivity middleware
End

Figure 4-3. Our Solution Architecture

4.1.1. Profiling the CIAS, Resource Network and host-MSD

To profile the CIAS sub-tasks offloadability/complexity, the used network and the local resources of the host-MSD, we have used a static analyser and dynamic profiler method. The static analyser determines the CIAS sub-tasks complexity, offloadability and granularity, so to identify local and remote sub-tasks to be executed on the host and assisting-MSDs. We excluded sub-tasks that require the host-MSD direct engagements, (e.g. inputs and outputs sub-tasks), these sub-tasks are executed locally on the host-MSD at all times. All other independent sub-tasks which are in sequence order are annotated as offloadable sub-tasks, using DFS algorithm as will be explained in the following Section. We have analysed the functions and loops that require intensive processing, (e.g. feature extraction, backtracking, search and sort algorithms), these must be offloaded to the assisting-MSDs.

The dynamic profiler fetched information from the host profiler and network profiler, also it waits for the report of features obtained from the assisting-MSDs, so to schedule the sub-tasks among the participated assisting-MSDs. App Tune-up kit tool [84] is used to profile the host-MSD local resources, (e.g. battery capacity, processing capabilities, and memory requirements when executing CIAS locally). We used a BroadbandChecker to profile the internet speed and make sure the network is stable during offloading. All the gathered information is input to the decision engine that decides on the sub-tasks to execute locally on the host-MSD and the sub-tasks to execute remotely on the assisting-MSDs.

4.1.2. Partitioning the Workload of CIAS to Sub-tasks

To solve the issue of sub-tasks dependency, we have defined a relationship assessment between sub-tasks offloadability, granularity and complexity, by monitoring sub-tasks call hierarchy, topology sequence order and dependency between sub-tasks. We have built remote executable sub-tasks from clustering a group of offloadable sub-tasks in sequence order, using a DFS fine-grained algorithm, so to partition the processing of CIAS to local and remote sub-tasks. It uses a linear time searching topology to search for the best integration point to divide a chain of sub-tasks without modifying to the sub-tasks sequence of order. It calculates the vertex and edges for each sub-task by estimating the execution time of the sub-task, based on the output obtained from the profiler. It excludes the sub-tasks that require to access the host-MSD local features and UI sub-tasks, these sub-tasks are executed locally by the host-MSD at all times. For the rest of the sub-tasks, it clusters them together in groups and offloads them as a package to the assisting-MSDs. We have modified the algorithm to accommodate our tested CIAS scenarios, as shown in Figure 4-4, so to build executable remote sub-tasks, (i.e. APK files for the available assisting-MSDs, or JAR/PHP files for Cloud/Edge servers). We have built a remote executable sub-task for the backtracking algorithm for the N-Queens CIAS so that it is marked as an offloadable sub-task and must be executed remotely by the assisting-MSDs. We only stored 2-inputs in the host-MSD local repository, local-vertex and remote-vertex, rather than fetching these inputs from the assisting-MSDs every time a sub-task is executed, so to eliminate the overhead cost. Then we offloaded the generated sub-tasks in parallel to the assisting-MSDs, using Async class and multi-threading functions. The assisting-MSDs executed the requested sub-tasks as the above and send the results back to the host-MSD upon completion.

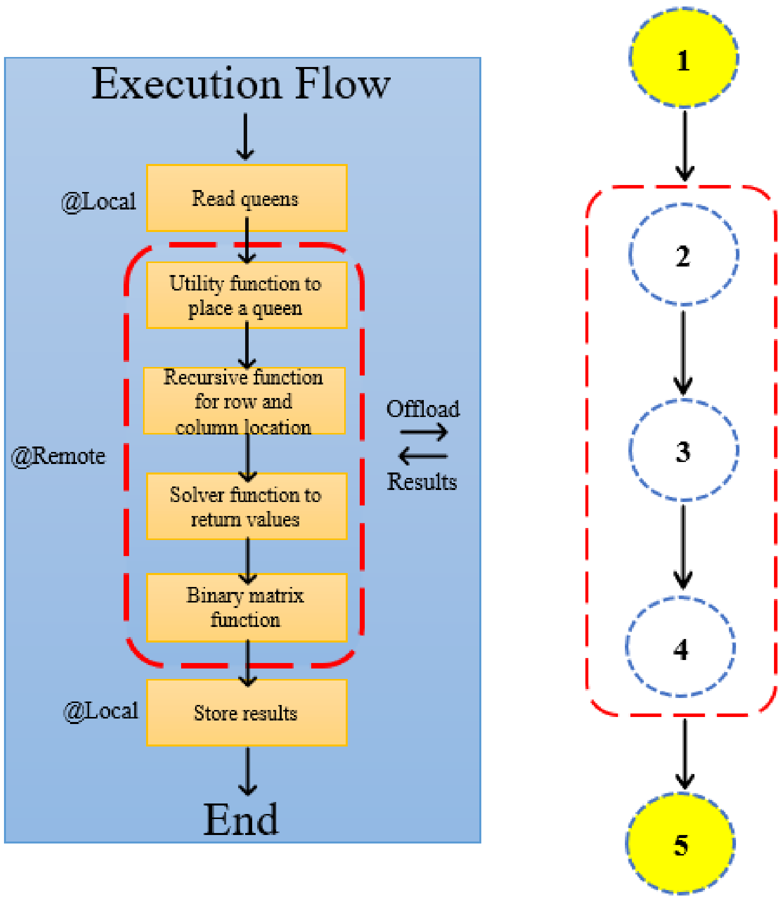

Execution Flow
@Local
Read queens
Utility function to place a queen
Recursive function for row and column location
@Remote
Solver function to return values
Binary matrix function
@Local
Store results
End
Offload
Results
1
2
3
4
5

Figure 4-4. N-Queens Execution Flow

Figure 4-5 shows the FD-R CIAS, we used DFS to build remote sub-tasks, (i.e. convert to greyscale, detect faces and extract features sub-tasks, as being the most intensive sub-tasks). The remaining of the sub-tasks are annotated as local sub-tasks and therefore must be completed locally by the host-MSD. We have used Async class and multi-threading functions to generate and execute the sub-tasks in parallel among the MSDs, as will be detailed in Section 4.2.1.2.

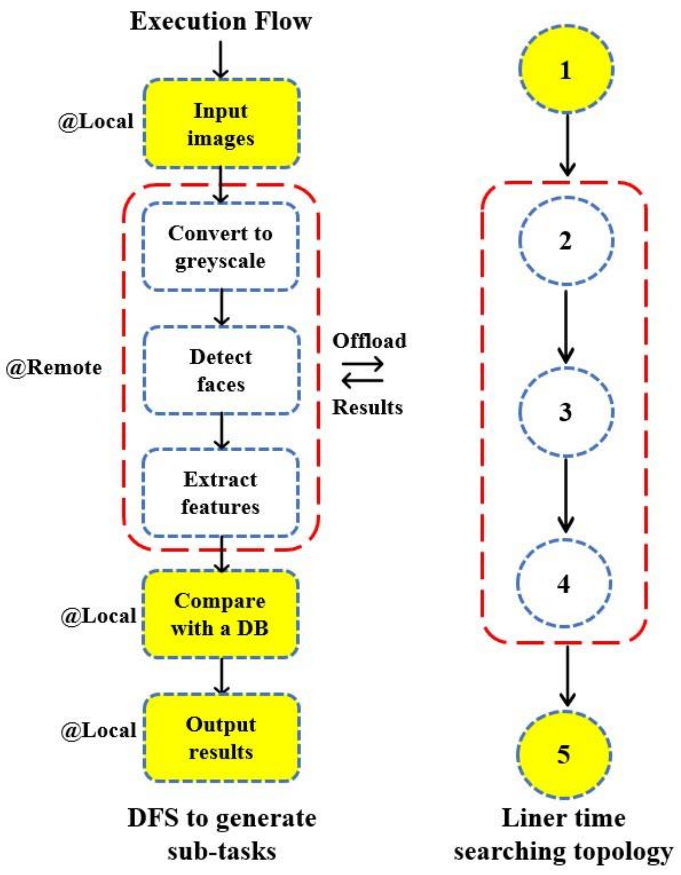


Figure 4-5. FD-R Execution Flow

4.1.3. Offloading CIAS Sub-tasks from the host-MSD to Assisting-MSDs

Figure 4-6 shows the execution stage in the offloading process, it consists of (1) the host-MSD, (we also call it here the Offloader), that have CIAS sub-tasks to execute and therefore needs help with processing these sub-tasks, (e.g. St1, St2, St3, St4, St5 and St6). (2) The Assisting-MSDs, (we also call here the Offloadees), that agree to help the host-MSD and are willing to share their unused processing resources. The host-MSD establishes connectivity with all the assisting-MSDs using a peer-to-peer network as explained in the next Section 4.1.4, and starts to offload the sub-tasks to the assigned MSDs, and communicates the results back upon completion. Based on the device having, at that time (a) the lowest load, (b) the highest processing resource, (c) a good battery capacity based on a defined threshold and (d) the best network connectivity to use. Meanwhile, the host-MSD executes its share of the sub-tasks as and when it is not busy with the other sub-tasks. When the CIAS run is completed, a summary record of this experience is fed back to the AI-engine to train and update it for future execution if needed by any other MSDs, as will be explained in Chapter 5.

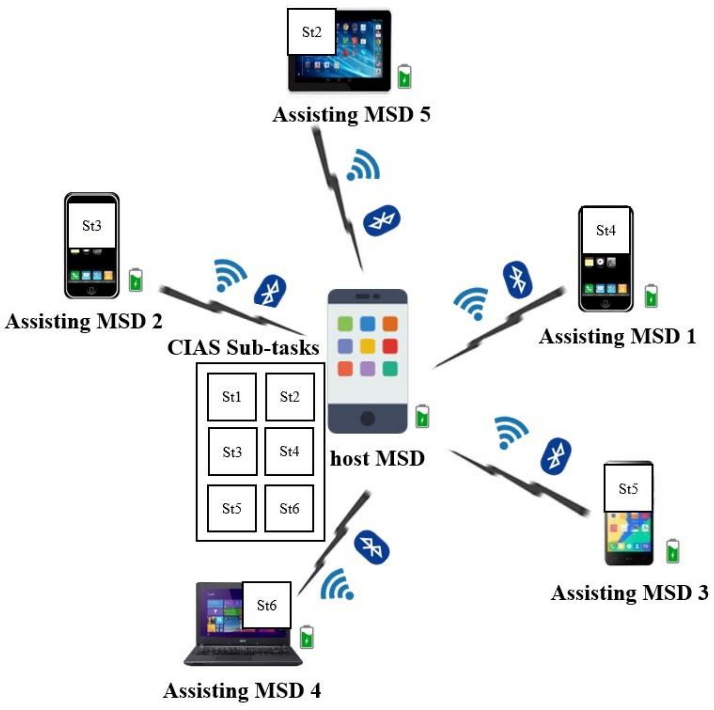


Figure 4-6. Cooperative Offloading Architecture

Upon a successful network connection, the assisting-MSDs wait and listen to any upcoming sub-tasks, then the host-MSD sends the sub-tasks in parallel to the assisting-MSDs. Finally, the assisting-MSDs execute the requested sub-tasks and send the results back to the host-MSD and the process is terminated.

4.1.4. The Forming of the On-the-go Resource Network

We have implemented peer-to-peer connectivity to form a network on-the-go of a group of cooperative assisting-MSDs in a sharing environment. The on-the-go in our solution refers to an offloading system that can adapt itself to enable the execution of any CIAS on ECR, by forming a resource network of MSDs as needed, and without prior planning. It provides a balance sharing environment to run across CS, ES, host and assisting-MSDs, as explained as part of our conducted scenarios in Section 4.2.2.4. We used Nearby API; a networking interface that allows end-devices to discover, connect and exchange data, (e.g. images, files, and videos), using p2p Wi-Fi and BT

[85]. Albeit, Wi-Fi is still preferable and mainly used in our implementations at all times, it enables low latency, high data transmission rate and fully encrypted p2p data transfer between MSDs that are in close proximity with each other. We have used it as a communication middleware between the MSDs, and since it enables a distributed architecture in which CIAS sub-tasks can be executed in parallel across peers [86]. It offers 2 types of protocols, “Nearby Connection” and “Nearby Messaging”, as well as it supports 2 topologies, “Star” and “Cluster”, we have used Nearby Connection type and Star topology. The reason for that is because (a) Nearby Connection offers unlimited data to be shared and it supports sensitive data by encrypting the data for secure payload exchange between MSDs. (b) Star topology where a master node connects to multiple MSDs, this is needed in my implementation to support the execution of CIAS sub-tasks from the host-MSD to the assisting-MSDs. We have defined 5 classes to establish the communication middleware between the host and assisting-MSDs, (i.e. Start Discovery (), Start Advertising (), Endpoint Discovery Call back (), Request Connection (), and Payload Call back ()).

We have used Android Studio to generate a simple interface to enable devices to communicate with each other as part of our end-to-end offloading App, as will be detailed as part of the experiments in Section 4.2.2.4. If we select more than 0 in the drop-down list, the host-MSD starts advertising itself to accept the incoming connection requests from the assisting-MSDs. The assisting-MSDs start discovering the host-MSD and send a request to connect. The host-MSD starts accepting incoming connections, (i.e. the number of incoming connections is equal to the number of assisting-MSDs). Then the host-MSD accepts the connection request and adds the incoming device to the connection list. Upon a successful connection, all the MSDs are up and ready to start sharing the sub-tasks among them.

4.2. Experiments and Evaluation

This section demonstrates the implementation of our experiments, (a) it explains the chosen CIASs we have developed to test in the experiments. As well as (b) it details various scenarios we have performed, so to offload the processing of CIAS sub-tasks from the host-MSD to the cooperative assisting-MSDs, and also to Cloud and Edge servers, (which is necessary for evaluating our achieved results with other offloading implementations). (c) It also explains the developed IPW cost model to calculate the actual and predicted efficiency gain and overhead cost. The results and discussion of the experiments are detailed next in Section 4.3.

4.2.1. The Four Developed CIASs we Chosen to test the ECR-engine

This section explains the CIASs we have developed to test in our offloading scenarios, so to prove the performance of our ECR-engine. We have used Android Studio IDE with NDK and SDK libraries to develop these CIASs, (namely, N-Queens Puzzle, FD, Fibonacci and FD-R). We have tested many CIASs, yet we have decided to use these as they met our solution and deployment criteria, (a) they are good examples to demonstrate sub-tasks offloadability, complexity and granularity. As well as (b) these CIASs are widely used by other offloading frameworks and so can be used for evaluation purpose, and also other reasons that involve (c) ease of configuration, code open-source availability/usability and compatibility to fit our testing environment.

4.2.1.1. N-Queens Puzzle

N-Queens puzzle is a game of placing N queens on an NXN checkerboard, where queens do not threaten each other horizontally, vertically, or diagonally. It uses a backtracking algorithm that performs forward and backward recursions to solve computational problems. The backtracking is a very intensive search algorithm, it checks for every single possible position to place a queen on the board [87]. Figure 4-7 shows an example of a backtracking algorithm to solve the puzzle of placing 4 queens on a 4x4 board.

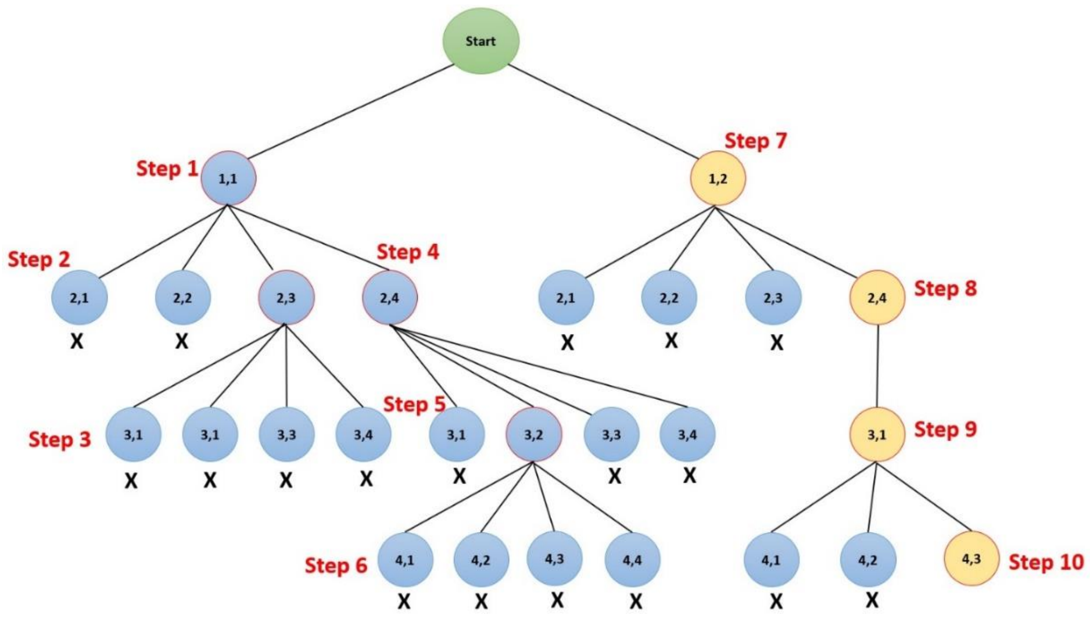


Figure 4-7. Example: A Backtracking Algorithm to Solve the N-Queens Puzzle

The algorithm performs the following steps to solve the 4 queens puzzle:

- It starts with placing the first queen in the top left square (1,1) of the first row. As a result, it cannot place any more queens in the first row, so it moves to the second row to prevent the attack.
- It tries to place the second queen in the second row at the first possible column, starting from the most left column, where it does not threaten the first queen. So, it places the queen in the square (2,3) and moves to the third row to prevent the attack.
- It tries to place the third queen in the third row, again starting from the left column until it finds the appropriate square that does not threaten other queens, which is not possible in the third row. So, the algorithm has to go back one step to change the position of the second queen in the second row.
- It shifts the second-placed queen to the square in the next column if available, here (2,4). Then it moves forward and tries to place the third queen in the third row until it finds the appropriate square which is (3,2).
- It moves forward to place the last queen in the fourth row, which is not possible, as all the squares are threatened by the previous three queens. The algorithm goes back to the third row again and moves the third queen to the next column (3,3) or (3,4), both columns are threatened by other queens. Then it continues the backword

recursion to move the second queen in the second row, which not possible too, as it reaches the last column in the second row. So, it reverts forward again to the first row to change the position of the first queen to the next column (1,2), and starts the same process all over again, until it succeeds to place all the four queens. As a result, the 4 queens are placed in (1,2), (2,4), (3,1) and (4,3) respectively, and the process is terminated.

The backtracking algorithm combines several intensive functions and loops, (e.g. utility function to check if the queen can be placed on a board, recursive function to solve row and column location and solver function to return the values). We have imported and launched N-Queens using Android Studio IDE, so to determine intensive loops and functions. After analysing the execution flow order and sub-tasks call hierarchy, we found that when N increases, more iterations are needed to find more solutions resulting in more processing. The solution is formed in a binary matrix with a value of 1 if the queen is placed in a specific square and 0 else wise. Each iteration involves several forward and backward placement steps consuming the host-MSD execution time and battery power. For example, when N is equal to 8, there are 92 iterations to place the queens on the board and when N is greater than 8, it needs hours to execute locally on the host-MSD, as shown in Table 4-1.

Table 4-1. Example: Number of Possible Iterations for N-Queens Puzzle

N	**4**	**6**	**8**	**10**	**12**	**14**
Iterations	2	4	92	724	14200	365596

The host-MSD generates a serialisable interface and decides on the sub-task, method signature and required functions to offload. We have built a remote executable sub-task for the backtracking algorithm using DFS so that it is marked as an offloadable sub-task and must be executed remotely by the assisting-MSDs. When the assisting-MSDs receive the serialisable interface from the host-MSD, it uses Java reflection API [56] to re-serialise the sub-task, invokes requested functions, executes the sub-task, and return the value of the sub-task. Details of the testing and experimental devices are illustrated as part of the conducted scenarios in Section 4.2.2.1.

4.2.1.2. Face Detection (FD) & Face Detection and Recognition (FD-R) CIASs

I have used a lot of knowledge for choosing these 2 CIASs coming from our school "image analysis and pattern recognition" research team. These two CIASs has been the main testing theme for us as not only show unique sub-tasks, but also can be deployed in parallel to serve bigger library of images for major recognition search.

FD is chosen to demonstrate the complexity of CIAS sub-tasks, it contains intensive functions and loops, (e.g. face detection and feature extraction). Recent studies indicate that using FD App to offload a 1 MB image takes 5 s, and the detection process tasks 10 s [69]. We have developed FD using Android Studio IDE and Dlib [88], a library for face detection and recognition that uses prebuild face prediction-algorithms, (e.g. SVM, ML and DL). It obtains a face-bounding box using (x, y) coordinators to detect faces in any given images, so to detect and mark 68 (x, y) coordinators for each face. Then, it extracts nodal points for each face such as the shape of cheekbones, the distance between eyes, length and width of the nose, as shown in Figure 4-8.

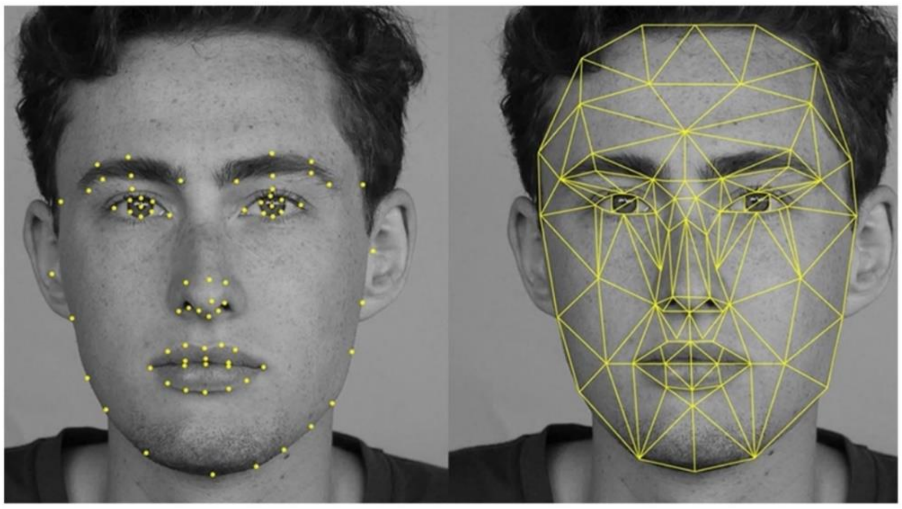


Figure 4-8. Detecting and Extracting Face Features in FD

Async class is used to exclude the GUI from all other sub-tasks, while multi-threading function is used to generate local and remote sub-tasks. Full details about the specification of the scenario and experimental devices are illustrated in Section 4.2.2.4. To generate more sub-tasks of FD, we have developed a more complex version; a FD-R which includes the recognition sub-task. As shown in Figure 4-9, FD-R includes the following sub-tasks (a) convert an image to greyscale, (b) detect faces in a given image, (c) extracts features and finally (d) compare the extracted features with a pre-build DB created during the pre-processing stage, using AWS Rekognition, as will be explained in Section 4.2.2.4.

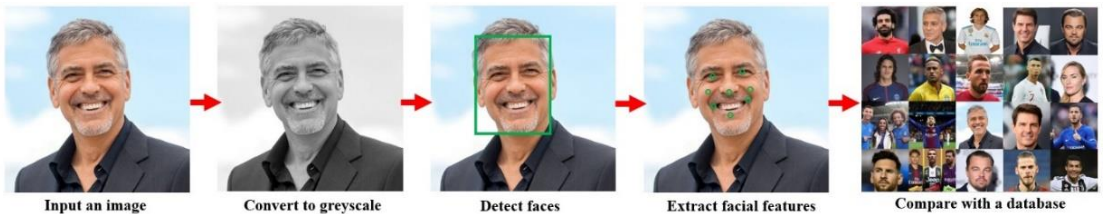


Figure 4-9. FD-R Sub-tasks Execution Flow

Finally, during the run-time of FD-R via the host-MSD, we have used the GUI to input new images, then the algorithm detects faces, extracts features and compares it with the DB. It displays the names of the faces and the extracted features, or no match if the person is not in the DB.

4.2.1.3. Fibonacci Algorithm

The Fibonacci algorithm is a sequence of a series of numbers, normally referred to as F_n sequence of numbers. It starts with (0,1), then each number in the sequence is the addition of the two previous numbers, as follows (0, 1, 1, 2, 3, 5, 8, 13, 21, 34) [89]. We have imported and launched the algorithm using Android Studio to develop a simple App UI, so to input a number and output the Fibonacci series count of that number. We have used the algorithm since it represents a memory-intensive App, rather than a processing/data-intensive App, especially when the input number is >10000, where it requires a long time to execute locally on the host-MSD. We have analysed the execution flow order and sub-tasks call hierarchy, so to check intensive functions and loops. We have found that it uses a recursion algorithm to map the input number into a mathematical equation to calculate the Fibonacci sequence series. This recursion algorithm is a powerful technique that produces accurate results, but it takes a long time to execute because it goes backwards and forwards every time it needs to calculate the equation. We have marked the recursion algorithm as an offloadable executable sub-task that has to execute remotely on the assisting-MSDs. The specification of the experiments, testing scenarios and experimental devices are detailed in Section 4.2.2.3.

4.2.2. The Six Main Scenarios to Evaluate the Performance of ECR-engine

This section details the conducted scenarios we have formed to offload the processing of CIASs sub-tasks from the host-MSD to the assisting-MSDs. We have implemented various tests to evaluate the performance of our solution, in terms of execution time, battery power consumption, connectivity and overhead costs. The tests vary to represent sub-tasks complexity, the number of assisting-MSDs and number of offloaded sub-tasks, as explained in Section 4.3. Also, the tests include evaluating our solution with two typical offloading frameworks. The specification of the tested devices including the host and assisting-MSDs is detailed in the following sections.

4.2.2.1. Offload N-Queens Sub-tasks from the host-MSD to Assisting-MSDs

In this scenario, we have decided to use 2 MSDs so that we can test the processing of CIAS among several MSDs. A Samsung Galaxy tablet S2 which acts as the host-MSD and also on a Nexus One Android SP emulator. We have implemented an Android platform running on Oracle VM VirtualBox [90], installed on a PC which acts as a nearby CS. Table 4-2 shows the full specification of all the MSDs used in this scenario. All the MSDs are connected to an AP using Wi-Fi and ethernet link in our lab at the University of Buckingham and can communicate and access the server with an IP address. After launching N-Queens on the host-MSD, we can specify the number of N to select and determine the sub-tasks to execute locally and remotely.

To evaluate the performance of our solution, we have performed several tests to cover and validate possible scenarios that might occur during the offloading process. These are varied between (i) executing all sub-tasks locally on the host-MSD, (ii) executing all sub-tasks remotely on the assisting-MSDs or (iii) execute partially among the host-MSD and the assisting-MSDs, using DFS algorithm. i.e. Sub-tasks are partitioned to execute among MSDs in parallel, and we measured sub-tasks execution time and consumed battery power. Android live monitor tool [91] is used to profiler the host-MSD local resources, such as battery level, processing capabilities, and memory requirements when executing CIASs locally on the host-MSD alone. App Tune-up kit tool is used to profile the overall findings including the average and peak consumed battery power, as well as the average processing time.

To the best of my knowledge, none of the reviewed offloading frameworks has used App Tune-up kit tool, our achieved results for this scenario are mainly based on this tool since it produces an accurate reading with a negligible error rate compared to similar tools. We profiled the network using BroadbandChecker to make sure the network is stable, using Wi-Fi with 216 Mbps uplink and 50 Mbps downlink, and 7 ms latency. Finally, the results are analysed and compared with other offloading frameworks, as detailed in Section 4.3.

Table 4-2. Experimental Devices Specifications

MSDs Specification	CPU	RAM	Battery Capacity	OS
Samsung S2 Sm-T710	1.3 GHz	3 GB	4000 mAh	Android 7.0
Nexus One Emulator	1.0 GHz	1 GB	1400 mAh	Android 7.1.1
Cloud Server	3.30 GHz	1 GB	N/A	Android 6.0

4.2.2.2. Offload FD Sub-tasks from the host-MSD to Assisting-MSDs

This scenario aims to examine sub-tasks complexity and granularity when the number of assisting-MSDs increases, as well as to evaluate the host-MSD execution time, battery power consumption and connectivity cost when executing CIASs sub-tasks. We have implemented a test to offload FD intensive sub-tasks in parallel among 4 assisting-MSDs. Table 4-3 shows the full specification of the MSDs used in this scenario. All MSDs are connected using Nearby API peer-to-peer network, as explained in Section 4.1.4. To evaluate the performance of our solution, we have performed various tests, our implementation varies from executing all sub-tasks locally on the host-MSD alone, or using assisting-MSDs to help the host-MSD processing FD in parallel. i.e. We have increased the number of assisting-MSDs in each test and observe the trade-off effect on the overall execution time, battery power, connectivity cost and offloading gain as detailed as part of our achieved results in Section 4.3. A total of 12 images are used in the test, each image is about 50 KB size, and the test is repeated 10 times. This test is further enhanced in Section 4.2.2.4 to demonstrate and evaluate the cost when the complexity of sub-tasks increases, as well as the trade-off of the overall connectivity cost.

Table 4-3. Experimental Devices Specifications

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

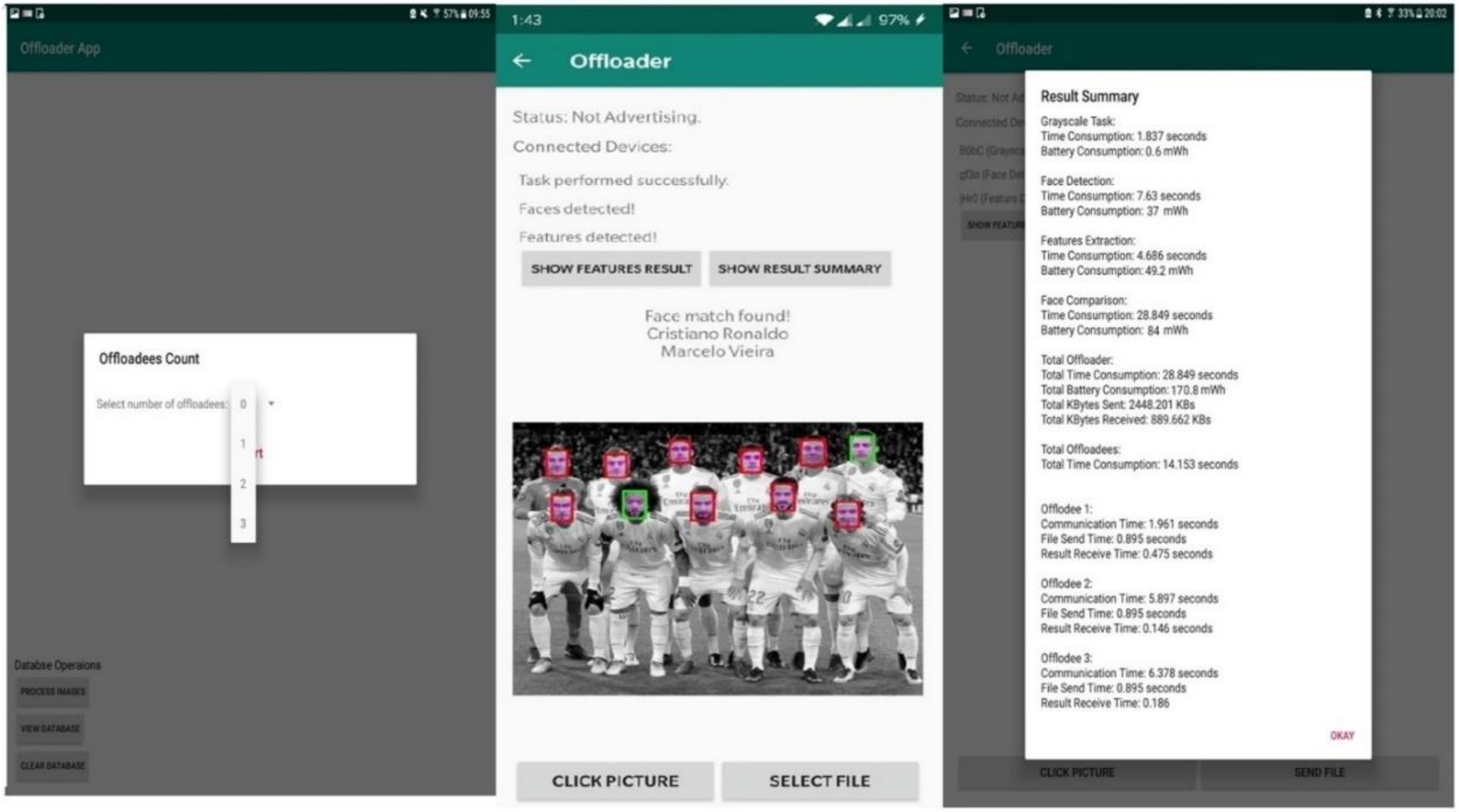
4.2.2.3. Offload FD, N-Queens and Fibonacci Sub-tasks from the host-MSD to Assisting-MSDs

This scenario aims to examine sub-tasks complexity and granularity when both the number of assisting-MSDs and sub-tasks increase, as well as to evaluate the host-MSD execution time, battery power consumption and connectivity cost when executing CIASs sub-tasks. We have tested 3 CIASs, (FD, N-Queens, and Fibonacci algorithm), so to execute in parallel among three available assisting-MSDs. To evaluate the performance of our solutions, we have performed serval tests which vary from executing all sub-tasks locally on the host-MSD alone or using the assisting-MSDs to help the host-MSD processing FD, N-Queens and Fibonacci in parallel. We have set the number of N to 8 for N-Queens, 6 images for FD, 5 runs for Fibonacci. i.e. We have increased the number of assisting-MSDs and sub-tasks in each test and observe the trade-off effect on the overall execution time, battery power, connectivity cost, and offloading gain as will be detailed in Section 4.3.

4.2.2.4. Offloading Programming Interface (OPI) to Offload FD-R Sub-tasks from the host-MSD to CS, ES and Assisting-MSDs

To mimic the functionality of ECR-engine, we have created an OPI that can perform real-time offloading trials among a pool of assisting-MSDs at the same time, including a CS, ES and assisting-MSDs. We have used Android Studio IDE, NDK and SDK libraries to create the OPI that can seamlessly be invoked on all the participated devices to form the on-the-go resource network among the MSDs. We have used 4 MSDs and 2 servers to conduct in the experiment, but we presume more devices can join the network and help with the processing, and especially when more CIAS sub-tasks are required from the host-MSD. The developed scenarios here are based on a simple distribution algorithm that shares the sub-tasks among devices. This scenario is also enhanced in Chapter 5, using a performance scoring algorithm that assigns the intensive sub-task to the device that has the highest resources and lowest load.

Figure 4-10 and Figure 4-11 show screenshots of the launching process of the OPI via the host-MSD so to perform the offloading trials, between the host and assisting-MSDs. The OPI contains three runtime scenarios, these are Offloader that acts as the host-MSD, Offloadee that acts as assisting-MSDs and server that represents Cloud and Edge servers. The Offloader scenario is to decide where to execute FD-R sub-tasks locally on Offloader or remotely on Offloadees, using a drop-down list of Offloadees between (0-3). The 0 means the FD-R sub-tasks execute locally on the Offloader, while 1-3 specify the number of Offloadees.



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

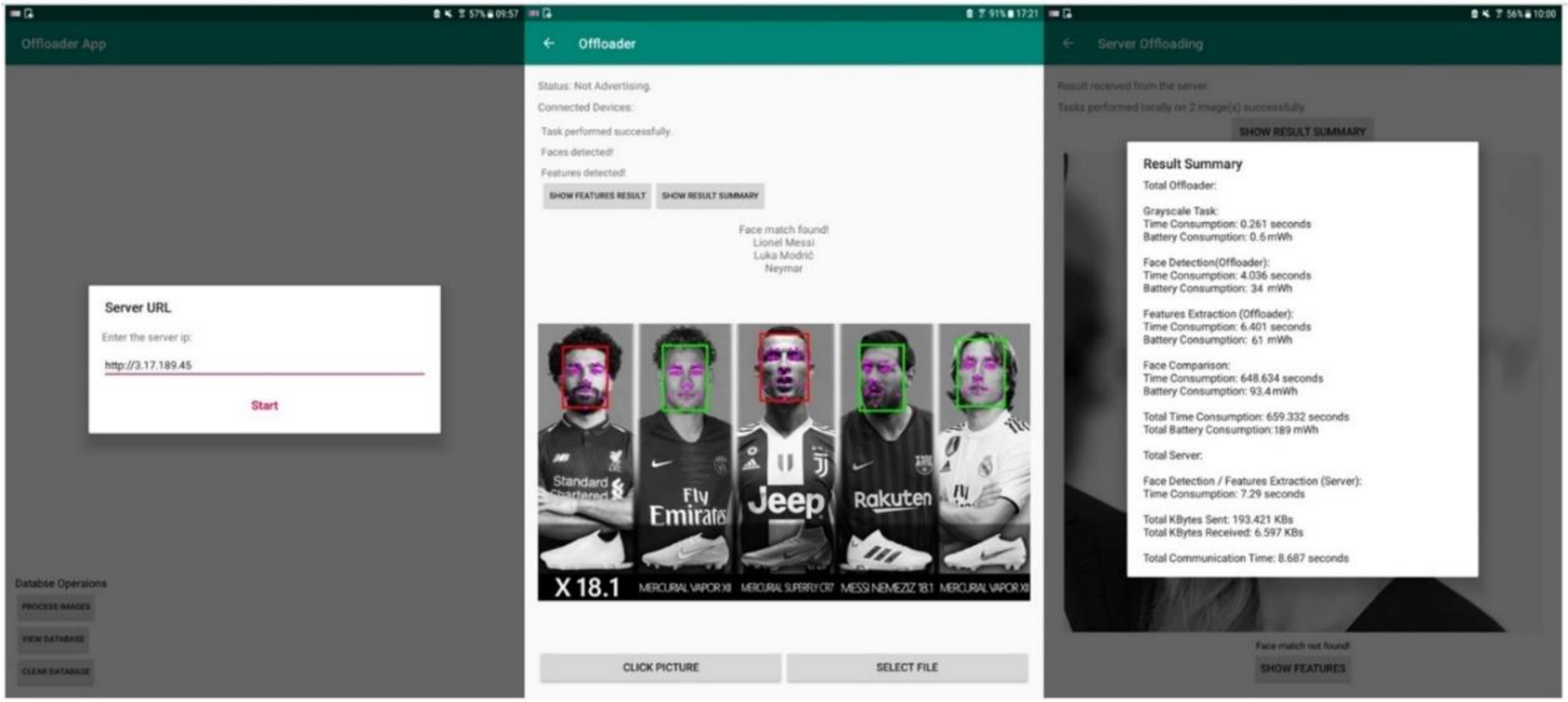
Total Offloadees:
Total Time Consumption: 14.153 seconds

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

Figure 4-10. A Screenshot of the OPI showing the Assisting-MSDs Scenario



Result Summary

Total Offloader:

Grayscale Task:
Time Consumption: 0.261 seconds
Battery Consumption: 0.6 mWh

Face Detection(Offloader):
Time Consumption: 4.036 seconds
Battery Consumption: 34 mWh

Features Extraction (Offloader):
Time Consumption: 6.401 seconds
Battery Consumption: 61 mWh

Face Comparison:
Time Consumption: 648.634 seconds
Battery Consumption: 93.4 mWh

Total Time Consumption: 659.332 seconds
Total Battery Consumption: 189 mWh

Total Server:

Face Detection / Features Extraction (Server):
Time Consumption: 7.29 seconds

Total KBytes Sent: 193.421 KBs
Total KBytes Received: 6.597 KBs

Total Communication Time: 8.687 seconds

Figure 4-11. A Screenshot of the OPI showing the CS Scenario

The Offloadee scenario is to represent the participated Offloadees. The server scenario is for executing FD-R sub-tasks remotely on the servers, (i.e. we used 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, and post API [92] to post/get sub-tasks to/from the server, as will be explained in the following sections.

1- Implementing the Local DB using AWS Rekognition Service

As shown previously in Figure 4-9, FD-R is consisting of 4 major sub-tasks, such as convert an image to greyscale, face detection, feature extraction and compare the new images with a pre-build DB. To create the local DB, we have used AWS Rekognition; a scalable DL platform developed by Amazon computer vision scientists. It offers several prebuild detection algorithms to analyse billions of images and videos. It performs accurate facial analysis and recognition to detect, analysis and compare celebrity faces, (e.g. sport, politics, business and entertainment people). It matches the upcoming images with a cluster of images, showing the level of similarities or confidence score of the new image, as shown in Figure 4-12, it also fetches face emotions, if the person is happy, sad, worried or surprised. The DB is needed later to compare the new images with the existing images in the DB, and so we have used AWS Rekognition service as a host server to recognise 100 images as will be explained later as part of the experiments. We have provided our images to the Rekognition API, then created a secret key, access key and IAM user to link the OPI to the service API. We have used the storage-based API [93] to store the images in S3-bucket, in case needed for future executions. The API uses post operations to process data/images to and from the service, it gets the images from the host-MSD local repository root. Then we call Detectface request, CallFace details and Detectfeatures functions to build a local client-side index. Then, it returns metadata includes a bounding box, facial landmarks, (i.e. the position of the mouth, ear, etc.), and face ID.

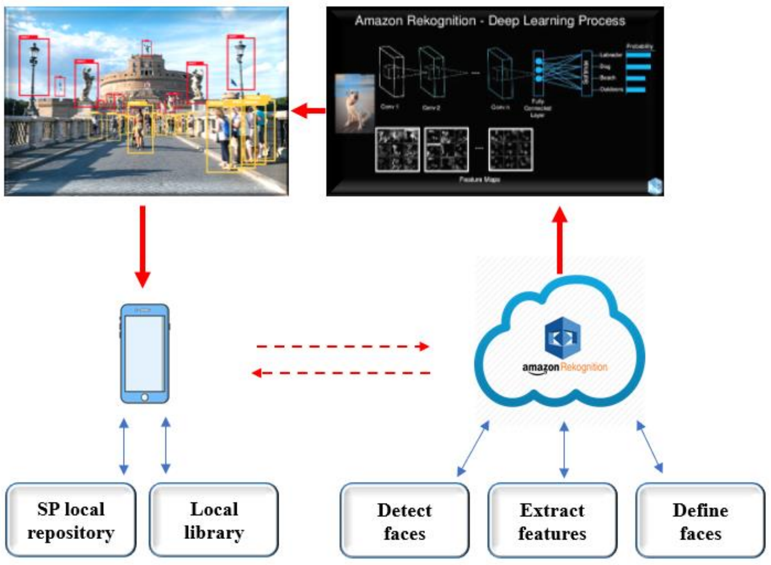


Figure 4-12. Creating a Local DB using Amazon Rekognition for FD-R

Also, the face detection function contains information about the detected faces and the time the face was detected using timestamps. The training phase is done on the host server, whereas the inference phase is done on the host-MSD, this is essential to eliminate the DB deployment overhead. Only the recognition results of the extracted faces are saved in a local DB shared repository.

2- OPI Scenarios and Offloading Trials

In this section, the various scenarios for the experiments that have been done to examine the benefits of offloading using ECR-engine, in terms of execution time, battery power consumption and connectivity costs, when FD-R sub-tasks are executed in parallel among the assisting-MSDs. These scenarios are referred to as Edge Server Scenario (ESS), Edge Offloadees Scenario (EOS) and Cloud Server Scenario (CSS).

- **Offload FD-R Sub-tasks from the host-MSD to ESS**

  In this scenario, we have created a WAMPSERVER 3.1.0 [94], which acts as a local nearby server. Both the host-MSD and server are connected through an IP address. If the decision is to execute FD-R sub-tasks by ESS, the decision engine triggers the distribution algorithm to partition the sub-tasks/images between the host-MSD and ESS. The host-MSD generates a serialisable interface and decides on the sub-task, method signature and required functions to offload. Then it invokes the remote manager to connect to the server using IP address and post API, it uses post/get commands to send/receive images to and from the server. The server waits and listens to any incoming sub-tasks, it runs the requested sub-tasks when receiving the images, records the time using timestamps, converts it to JSON format. Then sends the results back to the host-MSD upon completing the processing of FD-R sub-tasks. We have used a dynamic profiler to fetch information from (a) App Tune-up kit tool to profiler the host-MSD local resources, including battery level, CPU processing capabilities and memory requirements when running FD-R sub-tasks locally. (b) BroadbandChecker tool to profile the internet speed and make sure the network is stable during offloading.

- **Offload FD-R Sub-tasks from the host-MSD to EOS**

  In this scenario, we have performed offloading and processing the execution of FD-R sub-tasks to the assisting-MSDs. We have used a host-MSD and a maximum number of 3 assisting-MSDs, the full specification of the conducted MSDs used by EOS are detailed in Table 4-4. All MSDs are connected using a peer-to-peer nearby API, which allows the host-MSD to offload and share the processing of FD-R sub-tasks.

  We have developed a simple algorithm to distribute the images among the MSDs and the servers. Firstly, we divided the number of images (n) equally among the total MSDs, based on the number of registered devices during forming the resource network. After that we found the remaining number of images, if the remaining images 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 MSDs. For example, if the number of MSDs = 4, number of images = 10, then 10/4, so initially each device gets 2 images. For the remaining 2 images, it assigns one by one to the MSDs, so offloader = 2, offloadee1 = 3, offloadee2 = 3, offloadee3 = 2 and so on. The servers always get 75% of the total number of images, due to the unlimited resources it has, compared to the MSDs. The distribution algorithm is enhanced further in Section 5.2.4 to a performance scoring algorithm that can effectively and automatically score the performance of each MSD. This is based on monitoring the battery capacity, processing capability, network connectivity, current load, availability and RTT latency for each MSD.

The outcome of the scoring algorithm is used to create a dataset which is used to train and test the DNN model.

The assisting-MSDs wait and listen for any incoming sub-tasks, each MSD executes the grayscale, face detection and feature extraction sub-tasks when receives the images. Then records the time using timestamps and sends the results back to the host-MSD. A total of 100 images were used in the experiment, which set to have the same resolution (700X700) and a maximum size of 300 KB. Tests are repeated 5 times to examine unstable network conditions and the results are calculated, (an average of 5 runs), as explained in Section 4.3.

Table 4-4. MSDs Specifications used in EOS

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

- **Offload FD-R Sub-tasks from the host-MSD to CSS**

  In this scenario, we have created an EC2 Cloud server using AWS, namely “t2.micro Amazon Linux 2 AMI [95]”, and authenticated it with the offloading OPI using a secret and access keys and IAM credentials. We used FileZilla and Putty tool to install and migrate the necessary PHP files to the EC2 server. We created an S3 bucket to save the offloaded images, if needed for future execution and/or to train AI-engine for similar executions. If the decision is to execute FD-R sub-tasks by EC2 server, then the host-MSD connects to the server and starts to offload the images using an IP address and Post API. The server waits and listens to any incoming sub-tasks, it runs the requested sub-tasks when receiving the images, records the time using timestamps, converts it to JSON format, and sends the results back to the host-MSD.

3- The Importance Priority Weighting (IPW) Cost Model

We have developed a simple cost estimator based on a Markov Decision Process (MDP) [96], so to calculate the predicted cost overhead of our solution. The aim of using this model is to normalise the cost equation and to combine the arbitrary of different units and scales, especially when more MSDs are available to cooperate and share their unused resources. Let I denote a set of sub-tasks = {i_1, i_2, ….i_n}, let J denote a set of MSDs = {j_1, j_2, ……j_m}. To define the cost function of a sub-task i_n running on device j_m, we calculated the cost function as:

$$\boldsymbol{Cost(C)} = \sum_{\boldsymbol{j \,\epsilon\, jm}}^{\boldsymbol{i \,\epsilon\, in}} \left(\boldsymbol{P_{i,j} + T_{i,j} + L_{i,j} + D_j + B_j + E_j}\right) \tag{1}$$

Where, $P_{i,j}$ is the battery power cost of processing sub-task i on device j, $T_{i,j}$ is the time for processing sub-task i on device j, $L_{i,j}$ is the round trip time between 2 devices, D_j is the data size, B_j is the data transmission rate available at device j, and E_j is the efficiency of device j based on the CPU load. Since we have normalised the cost equation, the units of response time, latency and power consumption will not affect the trade-off. (Note that, in reference to Equation 1, the terms of time, battery power, RTT, efficiency, data size and data transmission rate refer to the cost of these expressions rather than the conventionally agreed unites of cost).

We defined a random weight factor (W) between {1,6}, that represents the probabilistic relative significance of power consumption, RTT, response time, data transmission rate and efficiency, and it is determined by:

$$\mathbf{W_P} = \frac{\mathbf{w_p}}{\mathbf{w_p + w_t + w_l + w_d + w_b + w_e}} \tag{2}$$

Considering that P is the highest priority in our implementation, followed closely by E and T respectively, they are weighted as 6, 5.5 and 5 scores being near the top of the scale. For D and B, they are regarded as a middle priority due to heavily dependent on the available assisting-MSDs and the available connectivity. So, the impact will not strongly influence the P, E and T which are the main objectives for the offloading, and both D and B score is 3. The remaining L scores 2 considering it is the least priority when compared with other weightings. Equation 2 shows the weight associated to the battery power and it is calculated by dividing the assigned weight value to the power on the total weight of all the other parameters, and this applies to (T, E, D, B and L).

After calculating the weights for all the parameters, we have modified Equation 1 by multiplying the weight of these parameters by the consumed readings of (P, T, D, L, E and B) gained during the testing, and therefore, we calculated the cost function as:

$$\boldsymbol{Cost\ (C)} = \sum_{\mathbf{j\ \epsilon\ jm}}^{\mathbf{i\ \epsilon\ in}} \left(\mathbf{w_p} \times \mathbf{P_{i,j}} + \mathbf{w_t} \times \mathbf{T_{i,j}} + \mathbf{w_l} \times \mathbf{L_{i,j}} + \mathbf{w_d} \times \mathbf{D_j} + \mathbf{w_b} \times \mathbf{B_j} + \mathbf{w_e} \times \mathbf{E_j} \right) \tag{3}$$

The cost equation is used to calculate the predicted cost overhead and efficiency gain of deploying ECR-engine. Section 4.3 presents the achieved results in comparison with the actual results gained during the implementation and testing the trials conducted using OPI.

4.3. Results and Discussion

This section presents the results obtained from implementing and testing various scenarios and experiments, including the offloading trials to execute CIASs sub-tasks in parallel among EOS, ESS and CSS. The achieved results are analysed to prove the performance of ECR-engine, so to reveal whether it benefits or degrades the offloading process.

Figure 4-13 shows the execution time of offloading and sharing the processing of N-Queens sub-tasks in parallel among the MSDs. Using our method to partition the processing of N-Queens sub-tasks reduces the execution time by 75%, it outperforms processing N-Queens sub-tasks locally on the host-MSD alone and remotely on the CS. This is especially when the number of N increases, where more iterations are needed to find more solutions to place the queens on the board, requiring more intensive processing. This is because the N-Queens is partitioned and offloaded to the assisting-MSDs at the same time. Besides, processing N-Queens sub-tasks on the CS defeats processing it locally on the host-MSD by 27%, due to CS unlimited processing resources.

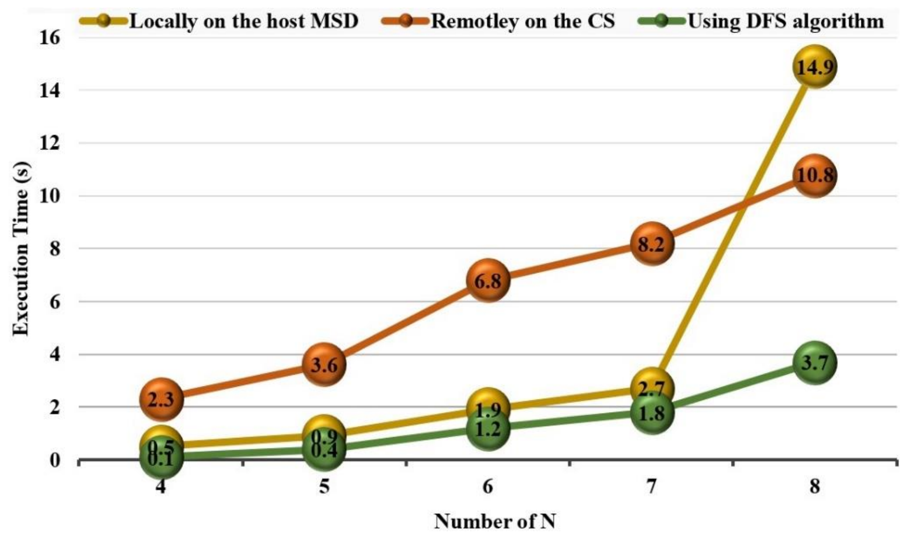


Figure 4-13. Processing Time to Execute N-Queens Puzzle Sub-tasks

Figure 4-14 shows the battery power consumption of offloading and sharing the processing of N-Queens sub-tasks in parallel among the MSDs. Using our method to partition the processing of N-Queens sub-tasks reduces the battery power by 20%. The consumed power dropped off continuously when the number of N increases, whereas the consumed power required to process N-Queens locally on the host-MSD raised steadily by around 7%. Similar to the linear trend of the execution time, when N increases, more intensive processing is necessary to process the sub-tasks.

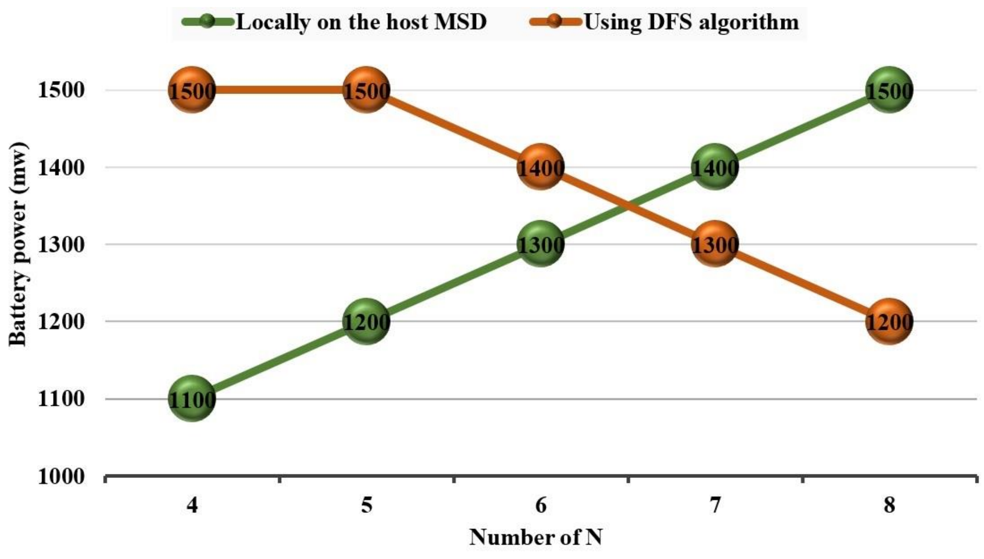


Figure 4-14. Consumed Battery Power to Execute N-Queens Puzzle Sub-tasks

Table 4-5 shows the overall findings including the consumed CPU, average and peak power for local and remote processing, using App Tune-up kit.

Table 4-5. Overall Findings for Local and Remote Execution

Runtime Environment	CPU	Average Power	Peak Power
Locally on host-MSD	22.3%	1.7 w	2.9 w
Remotely on CS	13.7%	1.0 w	1.0 w

Additionally, we have compared our solution with other offloading frameworks, so to measure the host-MSD execution time and battery power consumption, as shown in Figure 4-15 and Figure 4-16. Our solution differs in that, we have built executable remote sub-tasks from clustering a group of offloadable sub-tasks as explained in Section 4.1.2, rather than offloading sub-tasks individually in sequence order. Building executable sub-tasks is promising, as it effectively outperforms other typical coarse-grained and fine-grained techniques.

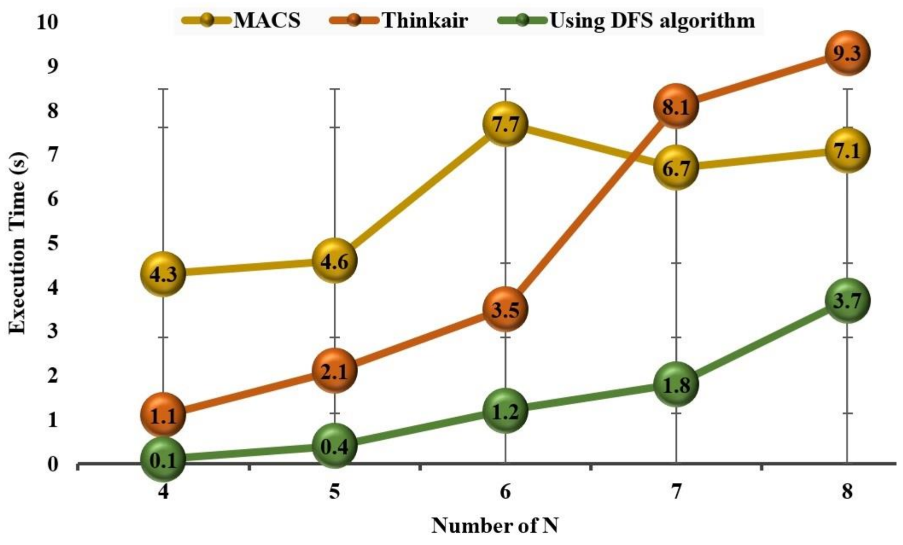


Figure 4-15. Processing Time to Execute N-Queens Sub-tasks

We have achieved 45.84% and 82.2% improvements to execution time and battery power, compared to MACS solution. In comparison to ThinkAir solution, we have achieved 57.84% improvements to execution time and much the same linear trend of battery power readings. Albeit, both the starting point of battery power is 1.5 w, but when sub-tasks complexity increase, the battery power dropped slightly using our solution, whereas for ThinkAir it fluctuated around the same behavioural trend. This will eventually change if more sub-tasks are requested by the host-MSD.

Figure 4-17 shows the executing time of processing FD sub-tasks locally on the host-MSD alone and by offloading and executing FD sub-tasks remotely on the cooperative assisting-MSDs. To cover all possible scenarios, our testing scenarios vary from executing all the images locally on the host-MSD and also by offloading and executing the images on the assisting-MSDs. i.e. We have increased the number of MSDs, (a maximum of 4 MSDs are used here), while sharing the images to execute among MSDs in parallel. i.e. For 12 images test using 4 MSDs, each device executes 3 images in parallel. This test is conducted to map a relationship between sub-tasks complexity and number of MSDs while evaluating the host-MSD execution time and battery power consumption.

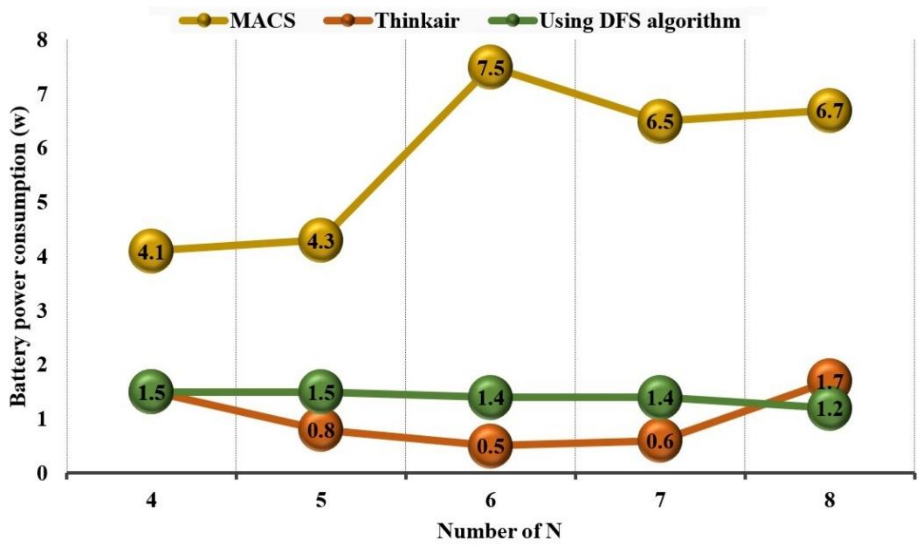


Figure 4-16. Consumed Battery Power to Execute N-Queens Sub-tasks

It is obvious when the number of MSDs increases, the overall processing time to execute FD sub-tasks has dropped significantly. When we used 3 assisting-MSDs helping the host-MSD with processing the sub-tasks in parallel, the processing time has reduced by 33.6%, compared to local processing on the host-MSD alone. Albeit, in the test of offloading to one assisting-MSD, the processing time has increased by 6%, this is due to the overhead not meeting the crossover point of being advantageous. The Figure also shows the predicted behavioural trend when more devices joined the network, (the red dotted line).

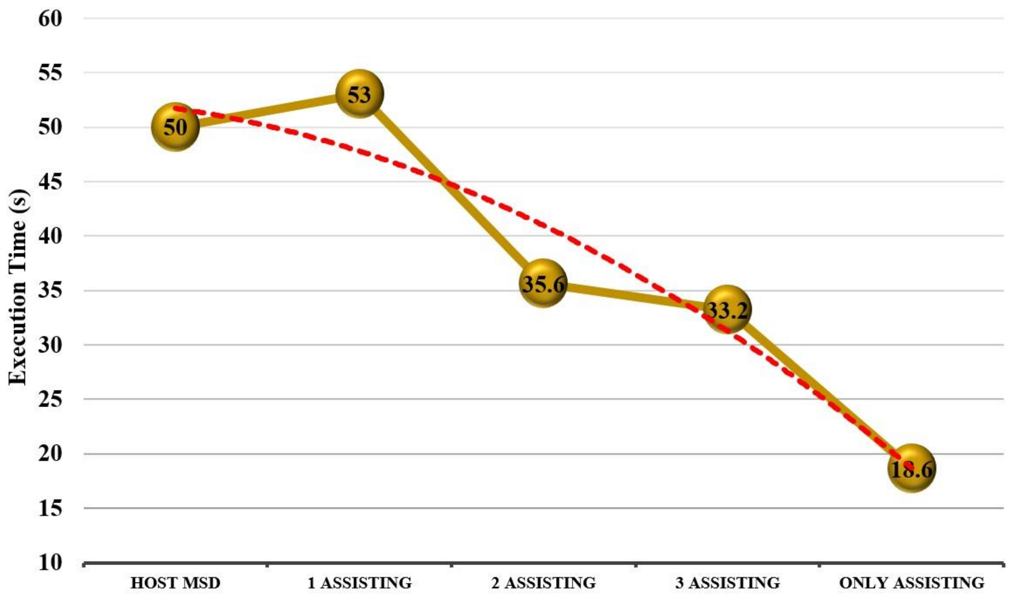


Figure 4-17. Processing Time to Execute FD Sub-tasks Locally and Remotely

Similarly, Figure 4-18 shows the battery power consumption of processing FD sub-tasks locally on the host-MSD alone and also by offloading the sub-tasks remotely on the assisting-MSDs. The behavioural trend of the consumed battery power followed much the same as the processing time results, when we used only one MSD to help the host-MSD processing the sub-tasks, the battery power has increased by 2.2%. When we used 3 assisting-MSDs helping the host-MSD with processing the sub-tasks in parallel, the battery power dropped sharply by 48%, compared to local processing on the host-MSD alone.

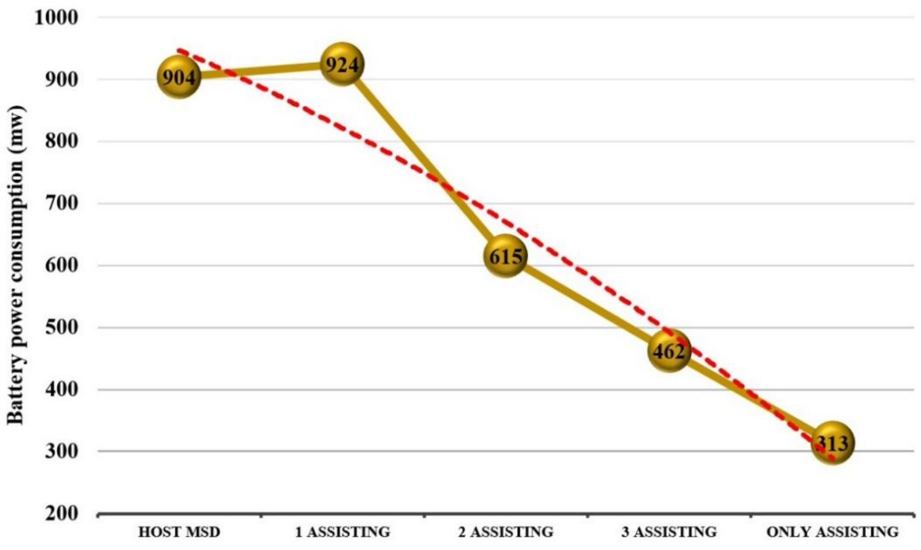


Figure 4-18. Consumed Battery Power to Execute FD Sub-tasks

To evaluate the performance of our solution, we have evaluated the processing time and battery power consumption while testing several CIASs to execute in parallel among the cooperative assisting-MSDs. In this test, we have tested three CIASs, in which case more complex sub-tasks are generated, shared and executed in parallel among the MSDs. Table 4-6 shows the processing time of executing FD, N-Queens and Fibonacci sub-tasks locally on the host-MSD alone and also by offloading and executing remotely on the assisting-MSDs. We have set the number of N to 8 for N-Queens, 6 images for FD, 5 runs for Fibonacci, as explained in Section 4.2.2.3. Executing all sub-tasks locally on the host-MSD alone consumes the most, albeit, in this test, we did not consider the connectivity cost, as the focus is to examine sub-tasks complexity by testing three CIASs. When we used 2 assisting-MSDs helping the host-MSD executing the sub-tasks in parallel, the processing time reduces by 51%, compared to 8% for one assisting-MSDs only.

Table 4-6 also shows the consumed battery power of executing FD, N-Queens and Fibonacci sub-tasks locally on the host-MSD host and also by offloading and executing remotely on the assisting-MSDs. When we used one assisting-MSDs to help the host-MSD executing the sub-tasks, the battery power did not improve but rather fluctuated much the same with the local execution. This is due to the overhead not meeting the crossover point of being advantageous at the starting point. When we used 2 assisting-MSDs helping the host-MSD executing the sub-tasks in parallel, the battery power consumption reduces by 33%, compared to the local execution by the host-MSD alone.

Table 4-6. Processing Time and Consumed Battery Power

Runtime Environment	Execution Time (s)	Consumed Battery Power (mw)
Locally on the host-MSD alone	68.9	2586
One assisting helping the host-MSD	53.3	2606
Two assisting helping the host-MSD	33.1	1834

Additionally, to evaluate the factors that impacted our solution, we did a simple test to plot sub-tasks complexity using several MSDs, while measuring the host-MSD offloading gain, actual CPU utilisation and battery power usage. Figure 4-19 shows the offloading gain when the sub-tasks complexity increases. i.e. We have increased the size of data to offload for each sub-task, (e.g. here we offloaded more images where data size varies between 300 KB to 600 KB).

It is clear that when more sub-tasks are generated, offloaded and sent through the network to execute in parallel among the assisting-MSDs, the overall gain has increased continually. This agrees with our analysis and equations explained previously in Section 3.3, that assured us, CIASs sub-tasks that require intensive processing with low RTT transfers between the host and assisting-MSDs, are always advantageous to offload.

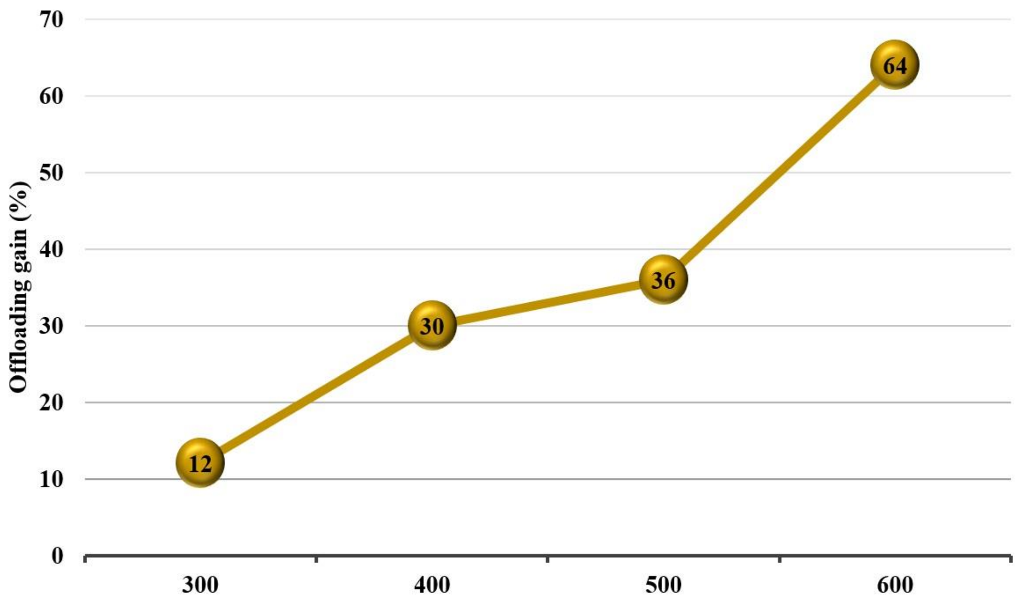


Figure 4-19. Offloading Gain when Sub-tasks Complexity Increase

Similarly, Table 4-7 shows the host-MSD overall actual CPU utilisation and battery power usage when more assisting-MSDs helping the host-MSD executing CIASs sub-tasks, using App Tune-up kit profiler.

Table 4-7. Actual CPU Utilisation and Battery Power

Host-MSD Resources	1 MSD	2 MSDs	3 MSDs	4 MSDs
Actual CPU utilisation	50%	44%	35%	18%
Actual battery power	80%	70%	60%	30%

We have increased the number of assisting-MSDs between 1-4 gradually, when more than two assisting-MSDs helping the host-MSD executing CIAS sub-tasks in parallel, both actual CPU utilisation and battery power usage have reduced constantly. Having only one assisting-MSDs executing the sub-tasks is not the ideal solution. However, the CPU utilisation and battery power have reduced by around 62% when we used 4

MSDs, this is nearly 3 times saving compared to the local execution by the host-MSD alone.

Figure 4-20 shows the execution time of processing FD sub-tasks by ESS, EOS and CSS which we described in Section 4.2.2.4. The results are testimony that forming ECR-engine is the correct decision, especially when compared with performing FD sub-tasks locally on the host-MSD alone. Offloading and sharing the sub-tasks by ESS and CSS reduce the host-MSD execution time by 83.4% due to their unlimited processing resources capability. While offloading the processing of FD sub-tasks to a single Offloadee is too costly with an increase of 14.3%, due to the overhead not meeting the crossover point of being advantageous. However, offloading FD sub-tasks to >1 Offloadee has significantly improved the host-MSD resource capability, (21.3% and 40.2% for 2 and 3 Offloadees respectively).

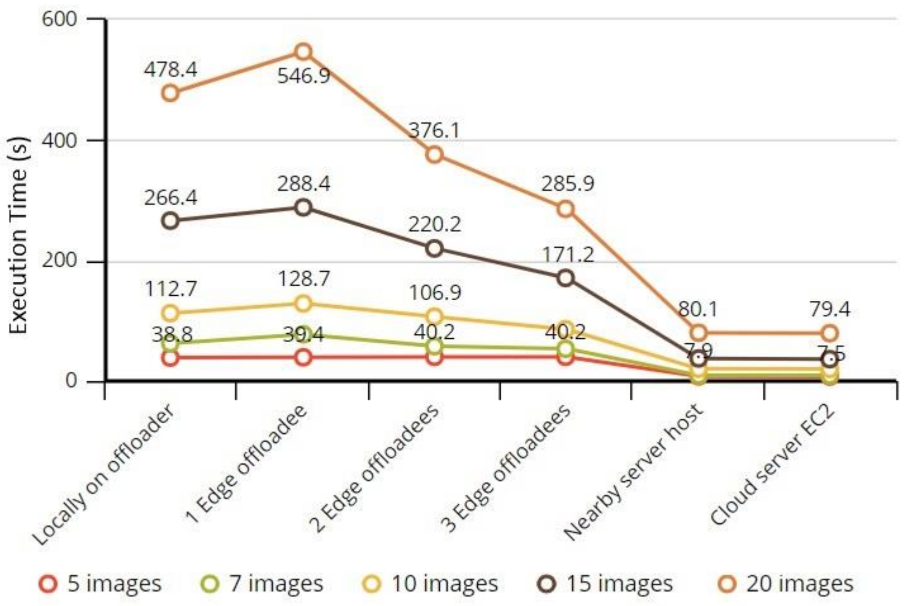


Figure 4-20. Processing Time to Execute FD Sub-tasks by EOS, ESS and CSS

Similarly, Figure 4-21 shows the execution time when processing FD-R sub-tasks for ESS, EOS and CSS. It shows an increase of the complexity of FD, by adding the recognition sub-task of matching the extracted features with a pre-build DB.

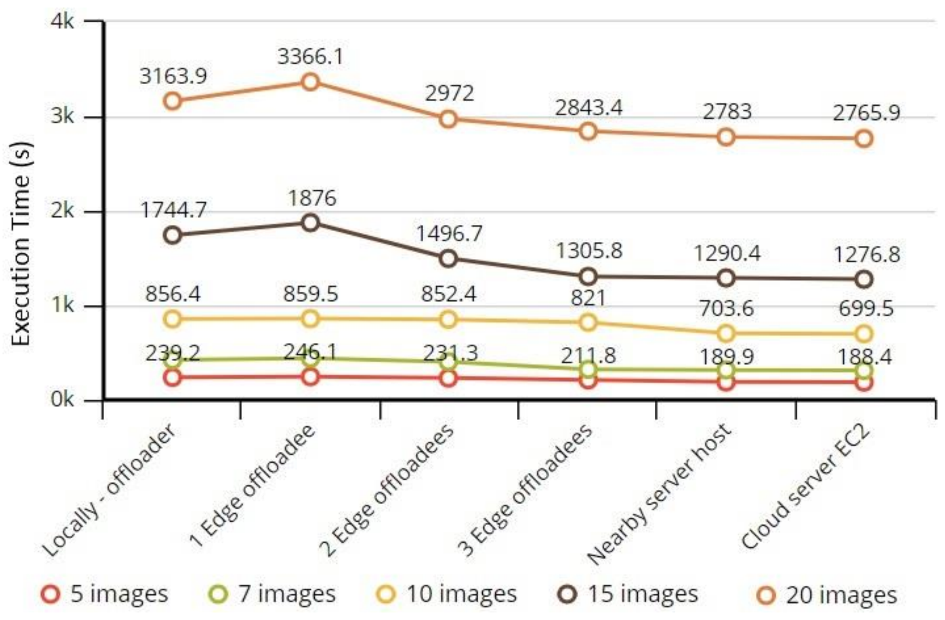


Figure 4-21. Processing Time to Execute FD-R Sub-tasks by EOS, ESS and CSS

This highlights the importance of ECR-engine, where the execution time became linear for all ESS, EOS and CSS. This means that the overall cost of ECR-engine is much less than having the offloading done by CSS, without the network traffic caused by transporting the sub-task to the CS. For 20 images with 4 MSDs, we have achieved 10.13% in comparison to processing FD-R sub-tasks locally on the host-MSD alone. While 12.1% for the CSS, which indicates the ECR-engine will outperform offloading to CSS when more intensive sub-tasks are executed on more participated assisting-MSDs. Note that the testing of the FD and FD-R CIASs is limited to controlled scenarios, (i.e. 100 images with 11 faces per image), where the taken/used images are in good quality, (e.g. without poor lighting conditions and bad face poses). This is because the purpose of the experiments is to conduct different offloading scenarios using these CIASs as benchmarks to evaluate the offloading process trend in comparison with other scenarios. Therefore, the above achieved results of FD-R for more challenging offloading scenarios, (poor lighting and poor poses), are expected to make our solution performed at least as good as the CS.

The battery power consumption is measured when executing FD-R sub-tasks by ESS, EOS, CSS and locally on the host-MSD alone as shown in Figure 4-22, it clearly shows that the same saving pattern achieved with the execution time. The behavioural trend we observed is, when only 2 devices are executing FD-R sub-tasks in parallel, the consumed battery power has increased by 19.52%. However, when more MSDs join the network as in EOS, we have recorded a battery power saving of 28.8% for 4 MSDs executing FD-R sub-tasks in parallel, which is almost similar to ESS and CSS which recorded about 31% battery power saving. None of the reviewed solutions has performed offloading of FD-R sub-tasks to nearby assisting-MSDs, by forming an on-the-go resource network. Albeit, to compare ESS and CSS with the model in [32], that performs offloading of FD sub-tasks to a CS, we have achieved improvements of 12.48% and 38.4%, in terms of processing time and battery power consumption.

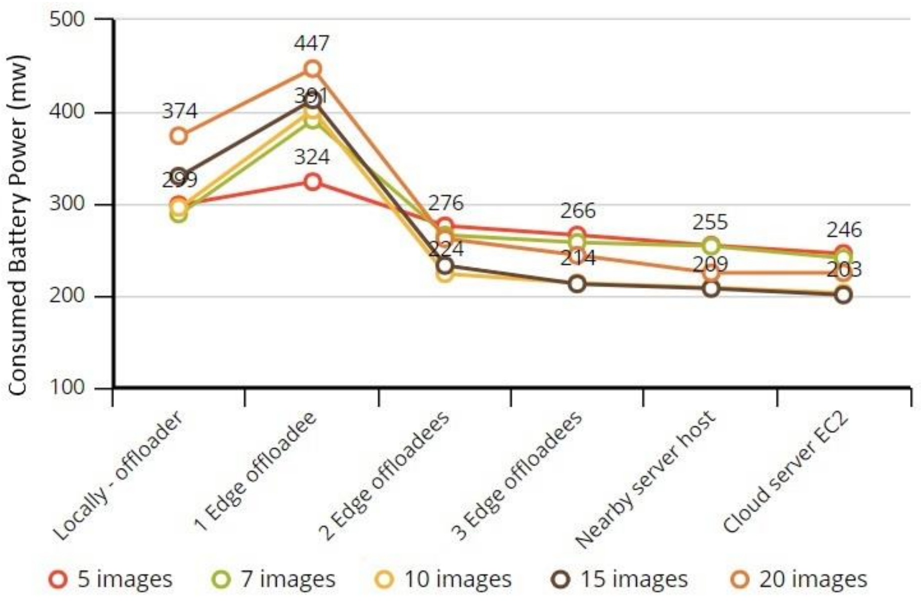


Figure 4-22. Battery Power to Execute FD-R Sub-tasks by EOS, ESS CSS

Figure 4-23 shows the execution time of processing individual FD sub-tasks by ESS, EOS, and CSS, it shows that the feature extraction sub-task is the most intensive compared to both greyscale and detect faces sub-tasks. Also, it shows, the processing time has dropped down continuously, recording a saving of 81.2% when more MSDs join the network and execute FD sub-tasks in parallel, which proves the concept and functionality of ECR-engine. To measure the accuracy of our OPI, we have 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 have achieved up to 99% accuracy as almost all the selected images and faces are recognised successfully. We also compared the accuracy rate achieved with the model in [97], that used relevant facial recognition service, we have achieved an increase of 23.75%.

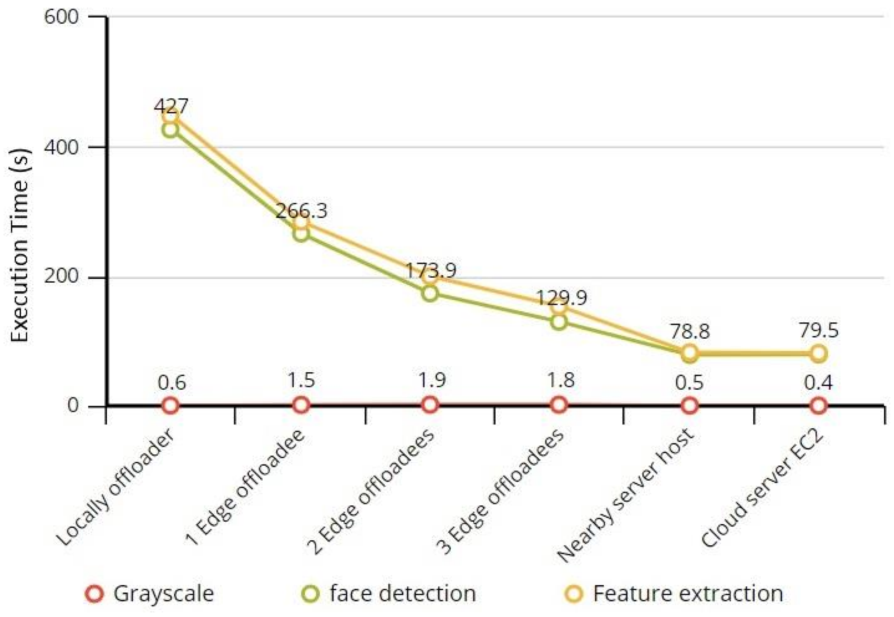


Figure 4-23. The Execution Time of Processing Individual FD Sub-tasks

Figure 4-24 and Figure 4-25 show the predicted cost overhead and efficiency gain of deploying ECR-engine, when executing the processing of FD and FD-R sub-tasks by EOS, CSS and ESS, using (IPW cost model, Equation 3 in Section 4.2.2.4.3). As well as comparing the predicted trend with the actual cost overhead and efficiency again, achieved during the experiments of the ECR-engine. Figure 4-24(a) shows the cost overhead of running FD-R sub-tasks locally on the host-MSD alone and by forming ESS, EOS and CSS. Processing the execution of FD sub-tasks on the host-MSD alone is too costly. When more sub-tasks are required by the host-MSD, the cost overhead continues to rise dramatically. i.e. Offloading and processing the execution of FD and FD-R sub-tasks to a single Offloadee with only 5 images, is too costly with an increase of 24% due to the overhead not meeting the crossover point of being advantageous. Albeit, when both the number of MSDs and the required sub-tasks granularity increase, the cost overhead has significantly improved. When we have 15 images and more, the host-MSD will suffer in terms of the consumed power the processing time, and therefore it becomes necessary to offload the sub-tasks of FD and FD-R to the assisting-MSDs that register cost saving of 40% for FD and 54% for FD-R because of adding the recognition sub-task. This highlights the importance of ECR-engine where the cost overhead became linear for all ESS, EOS and CSS. Figure 4-24(b) shows the predicted cost overhead achieved using IPW cost model, as explained in Section 4.2.2.4.3. We noticed here that the predicted cost overhead is decreasing when we have more MSDs processing more sub-tasks. At the starting point, when we used only one Offloadee, the cost overhead increases by 24.2% compared to the local execution on the host-MSD alone. For 20 images with 4 assisting-MSDs, we achieved 10.13% in comparison to the local processing, while 12.1% for the CSS scenario. Both the actual and predicted cost overhead measurements agree to have a similar behavioural trend. This indicates that forming ECR-engine is a better solution than offloading to a CS, as it reduces the shipped data/sub-tasks to the CS, which reduces the connectivity cost and the overall network traffic.

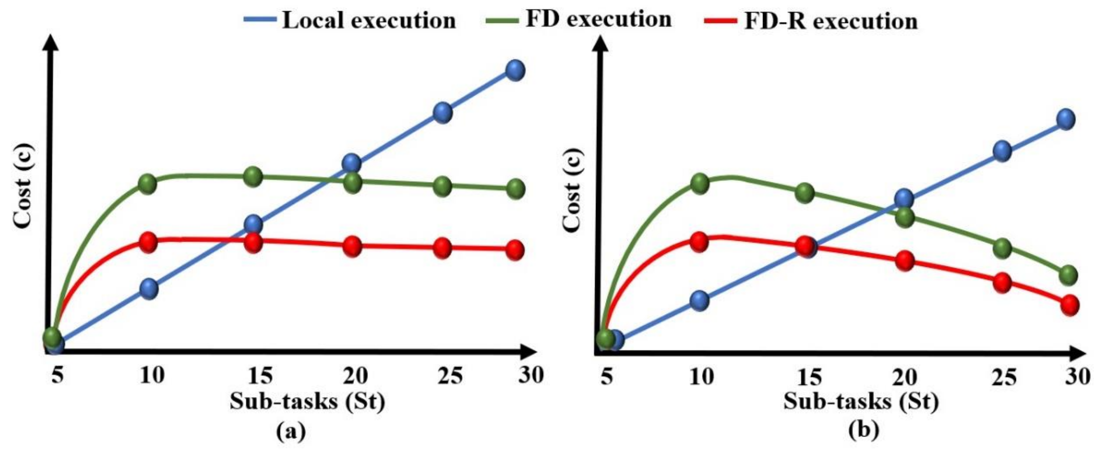


Figure 4-24. Actual and Predicted Cost Overhead from using ECR-engine

Similarly, Figure 4-25 shows both the actual and predicted efficiency gain when executing the processing of FD and FD-R sub-tasks locally on the host-MSD and by ESS, CSS and EOS. Our ECR-engine starts to be more efficient than the local execution when more MSDs join the network and help to process the FD and FD-R sub-tasks. i.e. For the 10 images test and when we used >1 offloadee, ECR-engine outperforms the local execution, and the efficiency gain continues to rise sharply. In both scenarios, we have registered an identical behavioural trend. Also, FD-R test records a better gain compared to the FD test which shows the advantages of our solution when the sub-tasks complexity/granularity increase, as well as the number of assisting-MSDs. We have also measured the RTT latency when the host-MSD communicates to ESS, EOS and CSS, compared to EC2 server average latency, the average latency of accessing 3 assisting-MSDs in parallel is equal or even less than accessing a Cloud EC2 server. EOS outperforms ESS and CSS, it achieves a decrease of 33% resulting in less latency. This proves that forming the ECR-engine that even with only 4 MSDs can reduce connectivity cost which will eliminate the overall network traffic. For sure having only a single Offloadee helping the host-MSD to execute the sub-tasks is not an option.

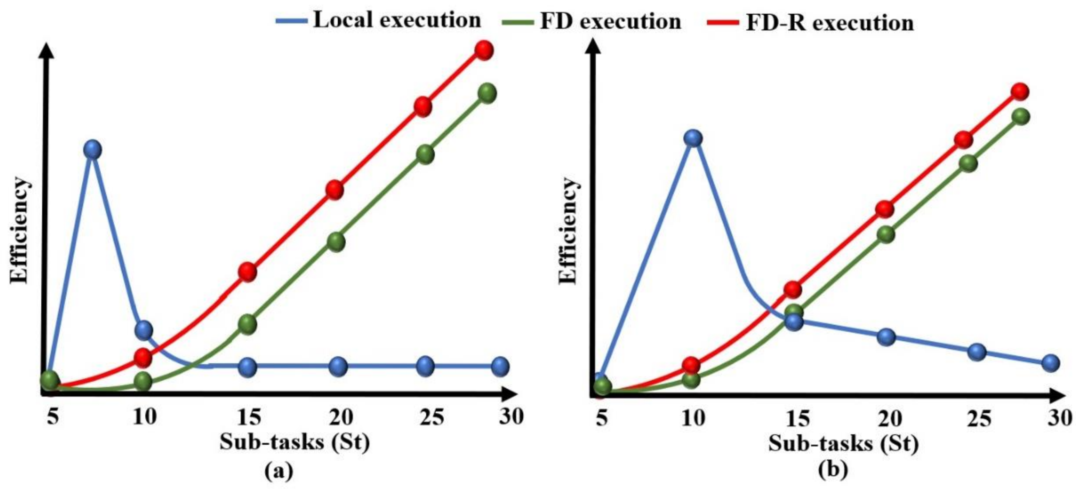


Figure 4-25. Actual and Predicted Efficiency Gain from using ECR-engine

The testing scenarios to evaluate the performance of ECR-engine show significant improvement, our solution on average have achieved 40.2% more efficient processing time, 28.8% less battery power consumption and 33% reduce latency, as well as 99% accuracy for the recognition sub-task.

Our implementation is mainly based on using nearby API connection which allows devices to communicate in a radio range of (~100m), where each device can advertise /discover other nearby devices to form a star topology network with them. This kind of network formation is also referred as near-edge computing where the resource infrastructure, (storage and processing), is deployed in a location between the far-edge computing and the Cloud DC closer to the end-users, so that it can quickly respond to delay-sensitive apps, (i.e. typically in less than 3 ms latency). This is important for us, as one of our central objectives is to minimise the latency and reduce the execution time, as explained in Section 1.1 for the required intensive CIASs sub-tasks, compared to processing these sub-tasks in the CS. Therefore, in our solution/experiments we have mainly focused on the near-edge computing scenario, so to use nearby devices in the vicinity/close proximity to the host-MSD that is hosting the app/data. On the other hand, the far-edge computing is often associated with the concept of fog computing explained previously in Chapter 1, where the processing resources are located in a middle layer between the end-users and the Cloud backbone, (normally in the gateways and/or base stations, and is located in approximately about 10’s km far from the end-users, which often takes an average of 7 ms latency). One benefit of the far-edge computing that it provides unlimited processing resources due to its proximity with the Cloud.

We have concluded that (1) the cooperative solution to offload the processing of CIASs sub-tasks in parallel to the assisting-MSDs proves to be as good as or even better than offloading to a typical CS. It reduces the high latency between the host-MSD and the CS, as well as it enhances the delay of offloading and availability of computing resources from having the focus shifts from the CS into cooperative assisting-MSDs. (2) Clustering and building a group of executable sub-tasks as a whole package is promising to eliminate the redundant sub-tasks/data pushed through the network, which will eventually eliminate the overall cost between the host and assisting-MSDs. (3) The on-the-go concept to enable adaptive resource network formation on the Edge is important, so to communicate among any nearby assisting-MSDs, without prior planning to set up the overall deployment infrastructure, especially, if a short-range peer-to-peer wireless network is used. This work is further enhanced by including a Cloud-based AI-engine that will help with the decisions of forming the ECR-engine, as will be detailed next in Chapter 5.

5. Cloud-based AI Implementation

Due to the importance and the advantages that AI adds to any database based decision engine, we have conducted an extensive review of the latest AI technologies and solutions to help us arrive at the most suitable implementation for this thesis as detailed in Section 2.3. Our Cloud-based AI-engine will host the training and validation of our implementation, so to intelligently facilitate the assisting-MSDs that can be chosen for each of our offloading sub-tasks, as well as to eliminate the cost overhead of deploying such engine on the host-MSD or the ES.

This chapter details our novel proposal solution so to maximise the performance and benefit of our ECR-engine, as well as it details our Cloud-side implementation including:

1. Explanation of our architecture for the Cloud-based AI-engine, (Section 5.1).
2. The building process of creating the training and testing datasets, (Section 5.2.2).
3. The deployment process of the DL decision-making including the chosen DNN model and other network parameters, (Section 5.2.3).
4. The implemented prediction and scheduling models to test the performance of our AI-engine, (Section 5.2.5).

5.1. Cloud-based AI-engine

Figure 5-1 shows our full solution including the AI-engine and (ECR-engine detailed previously in Chapter 4), and therefore the focus here is mainly on the functionality and implementation of the AI-engine.

Our solution uses the Cloud to host and generate the AI-engine that performs the following:

1. Determines suitable assisting-MSDs that can be used by the host-MSD to help with the processing of CIASs, so to monitor their processing resources and availability when needed and without prior planning.

2. Provides the host-MSD with decisions of the best scenario to partition and offload the CIASs sub-tasks among the assisting-MSDs using a DL decision-making model.

3. After forming all the required decisions, the AI-engine invokes the ECR-engine to offload the CIASs sub-tasks from the host-MSD to a group of assisting-MSDs, in a parallel/sharing environment.

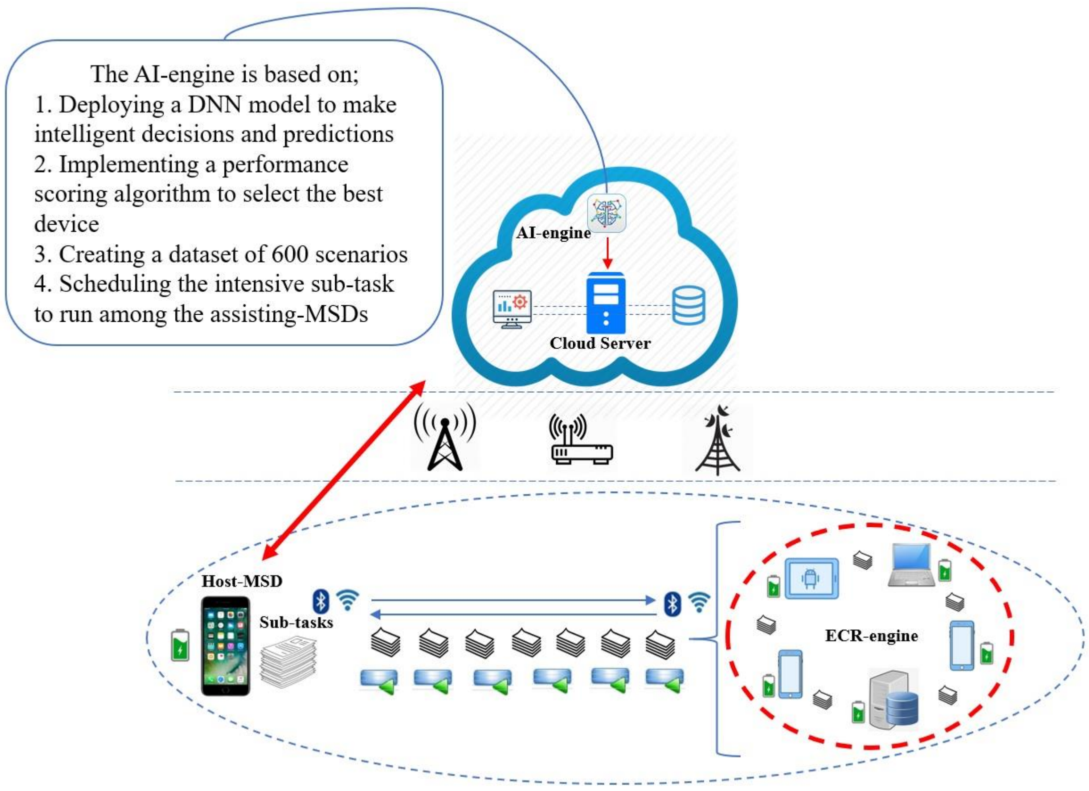

The AI-engine is based on;
1. Deploying a DNN model to make intelligent decisions and predictions
2. Implementing a performance scoring algorithm to select the best device
3. Creating a dataset of 600 scenarios
4. Scheduling the intensive sub-task to run among the assisting-MSDs
AI-engine
Cloud Server
Host-MSD
Sub-tasks
ECR-engine

Figure 5-1. Our Smart Solution: AI-engine and ECR-engine

Our offloading solution presumes as fundamental to its performance, that there will be various assisting-MSDs, including SPs, iPads and PCs that are available to cooperate in a resource network. This process is continuous, and we envisage that such devices, as a principle, are willing to contribute to help other MSDs and could be assigned certain credits, that could be used later in case they need help with executing CIASs sub-tasks. This is following on the trend to share anything to aid others: (1) voluntarily it comes to help the host-MSD, so to enhance the launching experience of such CIASs. (2) A credit scheme can be devised between grouped-users that can be exchanged for monies or other sharing schemes, as in [98].

The various tasks that needed to implement the AI-engine are:

- Using automatic sampling profiling by deploying Experitest platform [99] to profile CIAS sub-tasks. It interrupts the host-MSD CPU at certain time intervals and records the time of the currently executed instruction by debugging the metadata from the memory. Then it correlates the recorded execution time of the previous execution with the current execution at a given time. We have used sampling profiling since it is much simpler than static analysis, (i.e. no breakpoints are inserted to the source code), and is executed less often, (i.e. the execution order is not affected), resulting in negligible overhead. The output of this, as well as previous executions of the desired CIAS sub-tasks, are used to create the dataset.

- Developing a performance scoring algorithm to calculate the device performance score and select the device that has the highest performance score among others, (i.e. highest processing resources, lowest load and best connectivity protocol, etc). For example, a battery level between 80-90% scores value of 10, where a battery level of 20-30% scores value of 0 and so on for the other inputs. The performance score is a total score of 10 inputs for each MSD, as will be explained in details in Section 5.2.4. Based on the score, the most intensive sub-task is allocated to the assisting-MSDs with the highest score, any MSD scores a performance threshold of 40% or less will be discarded from the network, as being not qualified to help the host-MSD processing CIAS sub-tasks.

- Building a DL decision-making model to train and test the data, predict the next available MSD to select and make intelligent decisions to offload or not, if it decides to offload, then it schedules the sub-tasks among the participated MSDs, using a supervised DNN. The DNN is used to predict an output value from our pre-build dataset by (a) feeding the data into one or multiple nodes, each node is a function that transforms the input data into a numeric value. (b) Multiplying the values with a corresponding weight value and passing it to the intermediate layer and so on, until a decision is reached and the output values are presented. Details of the network architecture, including the used algorithm, hidden layers/neurons and the relative correlation between them are explained later in details in Section 5.2.3.1.

- Building a dataset of around 600 scenarios we used to train and test the DNN, (i.e. the dataset is split into 80% for training and 20% for testing). These scenarios are collected from possible behaviours that might occur during the offloading process, as well as actual testing scenarios conducted previously in our implementation as in Section 4.2.2.4. Each scenario inputs 10 parameters/features, (i.e. battery level, processing capability, number of assisting-MSDs, number of sub-tasks, sub-tasks granularity, network type, RTT latency, current load and availability).

- Scheduling the sub-tasks among MSDs based on complexity, highest performance score, and assign the intensive sub-task/chunk to the best MSD based on the above criteria, while managing to discard the weakest devices from the network, (e.g. such devices that have a low battery level, and/or devices that are busy performing some other tasks, etc.). This is achieved by developing a simple UI inference service that can be invoked seamlessly on the Edge to exchange features among MSDs, (e.g. battery level, processing capability, etc.), used to update the DB regularly, or for future executions.

- After completing the above and when contacted by the host-MSD, the AI-engine (a) provides a list of potential available MSDs together with their capability and advises the host-MSD with the best decision of what sub-tasks to offload and where to execute each sub-task. Then, it (b) invokes the ECR-engine to form the resource network and start the offloading process, as explained in Chapter 4. Finally, it (c) gathers the learning output gained from executing all the sub-tasks on the assisting-MSDs, so to boost up the offloading decision for further executions or to fine-tune the DNN model if needed.

Details of each of these steps will be detailed as part of the experiments section we have done to prove the concept and performance of our solution. The implementation of the AI-engine is based on a pre-build existing DNN model, (i.e. we did not build the algorithm ourselves, we only used it to train and test the dataset). Albeit, we did change some parameters for better prediction results to fulfil our testing sub-tasks, as will be detailed in Section 5.2.3.1.

Figure 5-2 shows a flowchart of these steps, and the following sections give a detailed implementation of these steps.

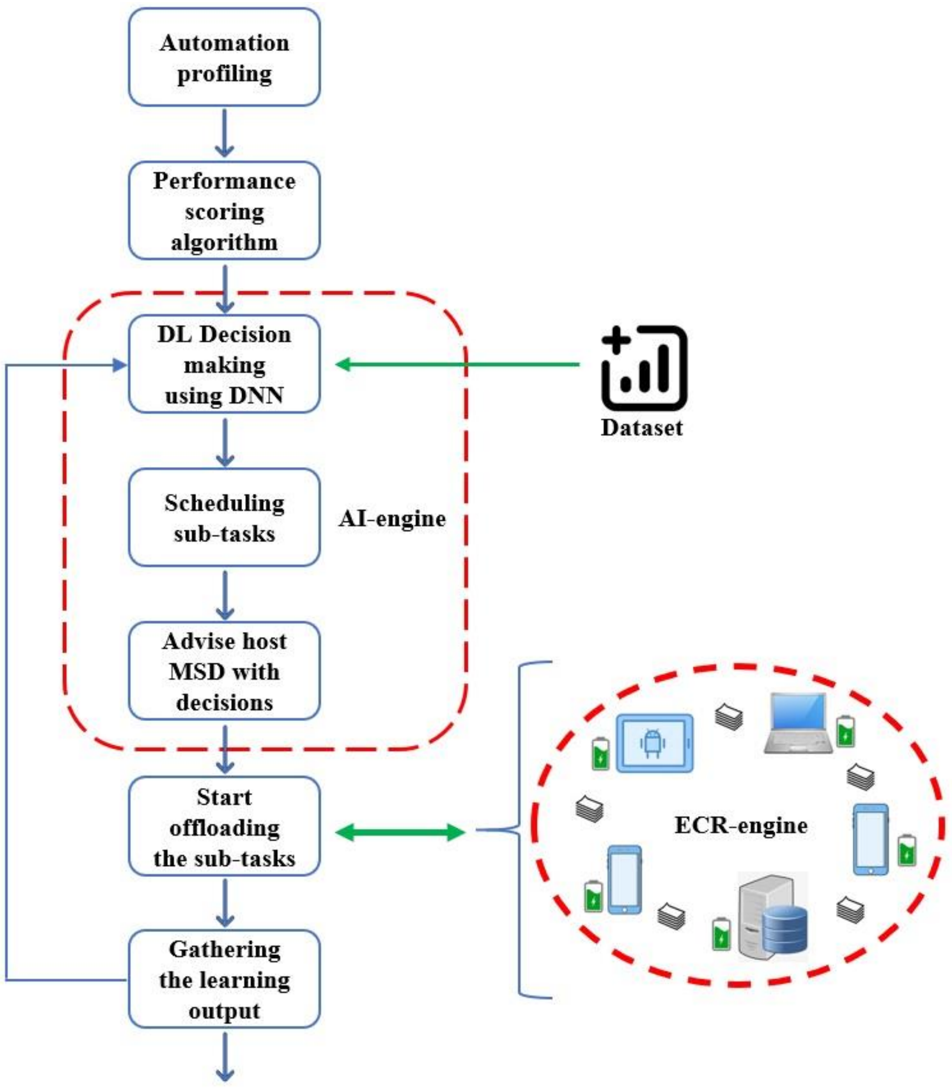

Automation profiling
Performance scoring algorithm
DL Decision making using DNN
Dataset
Scheduling sub-tasks
AI-engine
Advise host MSD with decisions
Start offloading the sub-tasks
ECR-engine
Gathering the learning output

Figure 5-2. Our AI-engine Flowchart Diagram

5.2. The Implementation and Testing of our AI-engine

This section demonstrates the implementation and testing which we have conducted to evaluate the performance of our AI-engine as follows: (a) it details the platform we used to deploy our AI-engine so to build the DL decision-making model using a DNN model. (b) The creation of the training and testing datasets, (c) the development of the performance scoring algorithm. It also explains the (d) scenarios we developed to test the accuracy of our AI-engine, including a prediction scenario to make intelligent offloading decisions and a scheduling scenario to schedule the CIAS sub-tasks among the assisting-MSDs. i.e. The scheduling is based on the device having at the time the highest processing resources and lowest load among others, any device has a performance score of 40% or less will be discarded from the network, as not being qualified to help the host-MSD executing the required sub-tasks.

5.2.1. Runtime Environment: BigML Platform to Deploy a DNN Model

The decisions of which AI algorithms and platforms to use and implement my work was a tough decision. I have investigated several algorithms, (e.g. DL, ML, RL and DRL), and used various platforms, (e.g. Collab, TensorFlow, AWS Sage Maker and BigML), so to build models/classifiers for image processing/recognition, text and voice recognition. Albeit, Evidence from the reviewed publications, (detailed in Section 2.32.3), as well as our evaluation and testing, have assured us:

1. Many researchers in the field of AI claim that DL-based algorithms outperform ML algorithms that are based on handcrafted maps [65].

DL algorithms are similar to Artificial Neural Networks, but with a much larger and complex structure which consist of several interconnected layers, (input layer, middle hidden layer(s) and output layer). These layers contain multiple nodes, each node has two functions, (weight between the input layer and the hidden layer and the time it takes to process a single input). According to [100], three-layer network (one hidden layer) is a universal standard network, these layers are connected to learn and extract features continually from observing data. As shown in Figure 5-3, the input (x) is multiplied by a weight (w), combined and summed at one hidden node, then the result is added to a bias and finally is fed into an activation function to produce the output, more details of the construction of relevant DL-based algorithms are summarised in the literature review chapter in Section 2.3. Also, details of our deployed algorithm with the number of layers, hidden unites and activation function are demonstrated in Section 5.2.3.1.

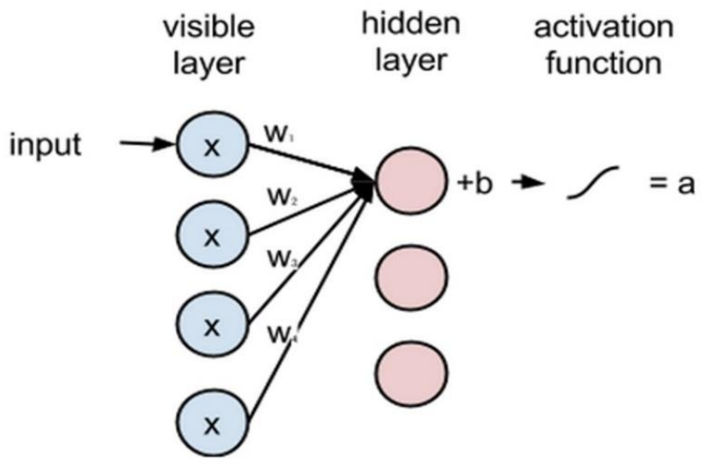


Figure 5-3. DL Standard Structure

2. BigML platform is more robust and advance compared to other AI platforms due to the following:

- It provides a powerful GUI launched as an MLaaS operated in the Cloud that includes fully automated ML and DL algorithms, in a supervised and unsupervised structure.
- It is used to build accurate models that can be used to solve complex problems, including classification, regression, time-series forecasting and clustering. As well as it provides support documentation and tutorials to help using the platform.
- Most importantly, it includes a DEEPNET model; an optimised version of DNN that enables an automatic network search of several networks and selects the best among them [101]. We believe this is important to avoid time complexity and processing requirements of manual-tuning to fit the network.

5.2.2. Building the Training and Testing Datasets

In order to build AI models that perform accurate decisions/predictions with a minimum error rate, data must be prepared and transformed to an ML-format dataset, (i.e. a structured version of the data), so that the model can understand, learn and extract features. BigML uses sources; a source is multiple instances/entities prepared to be input into a model, then the source is used to create/prepare datasets from the uploaded data, and finally, the main dataset is split into training and testing datasets/subsets. Figure 5-4 shows the process of preparing and creating the dataset, followed by implementation details about the complete process.

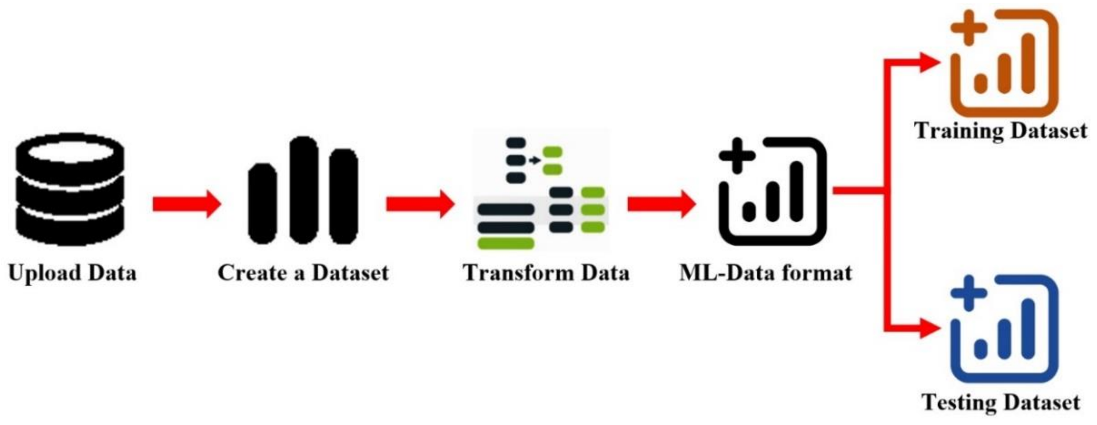
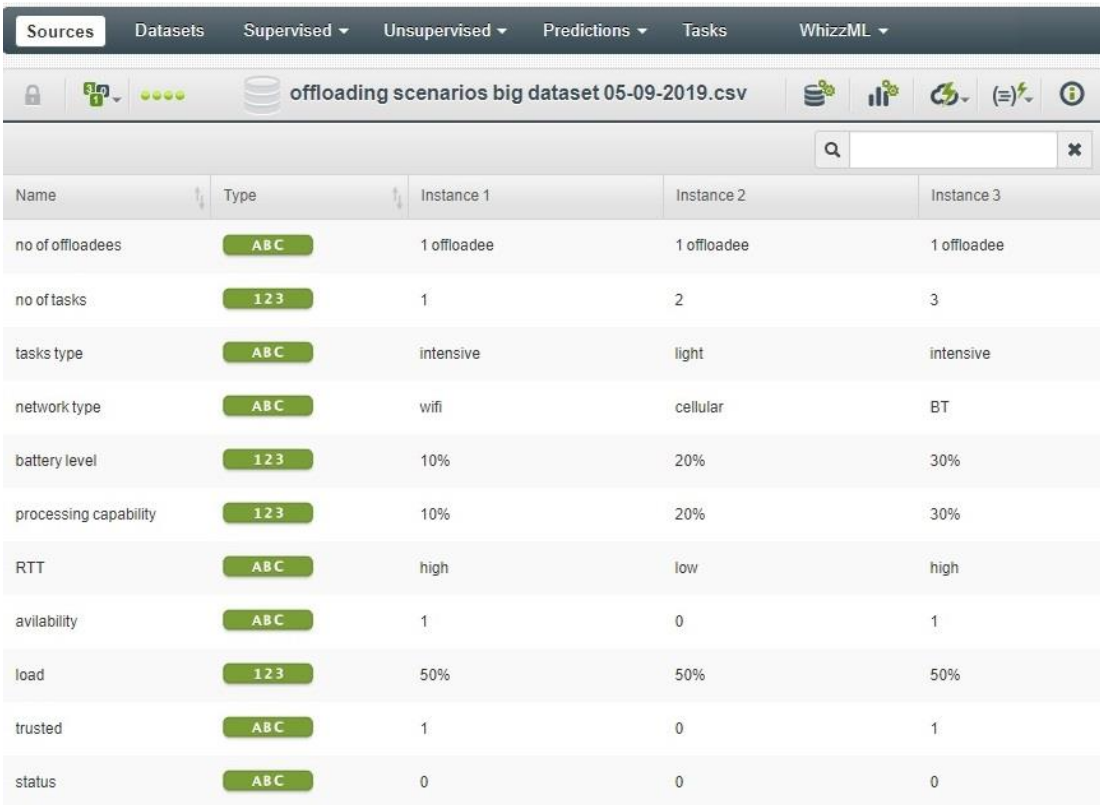


Figure 5-4. The Process of Building a Dataset using BigML

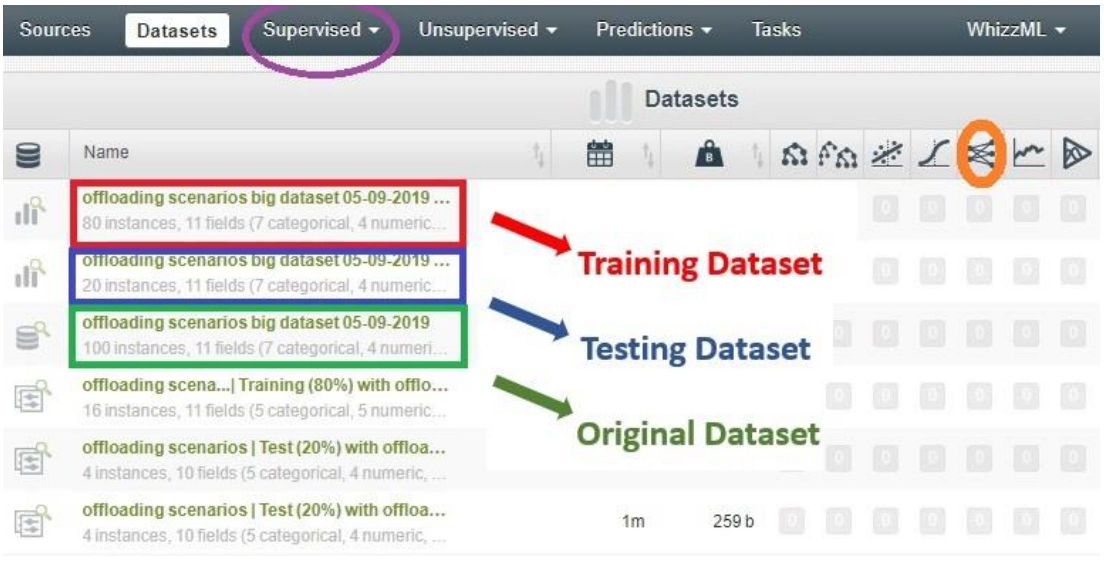
To create a source, we have prepared/labelled around 600 possible scenarios, using a Comma Separated Values format in a supervised manner, then we uploaded the source to BigML platform. The decision of creating our own dataset is because to the best of our knowledge, we did not find any relevant offloading solution that built a DL decision-making model, therefore, no dataset is available. These scenarios are collected from using the sampling profiling, performance scoring algorithm and sub-tasks previous executions. Then we transformed it to a structured ML-format so the model can understand all the data accurately by (1) classifying the data into various categories, (e.g. categorical, numeric, text, etc), (2) configuring instances sizes, missing values and data repetition, then finally (3) converting the instances/entities to vectors. We have created 2 datasets, the first dataset is used to create the prediction scenario, while the second dataset is used for the scheduling scenario of sub-tasks among MSDs, as will be explained in Section 5.2.5. i.e. For the prediction scenario, we have labelled our scenarios with 0/1 values as in a typical supervised manner [16], where 1 represents a suitable offloading candidate, and 0 if not, based on the input features. The engagement status is the objective field which we want the model to be able to predict/decide. E.g. We have 13 scenarios with a 60% battery level and 51 scenarios with a high RTT latency and so on for other features. Figure 5-5 shows a screenshot of the creation of the source/dataset.



Name	Type	Instance 1	Instance 2	Instance 3
no of offloadees	ABC	1 offloadee	1 offloadee	1 offloadee
no of tasks	123	1	2	3
tasks type	ABC	intensive	light	intensive
network type	ABC	wifi	cellular	BT
battery level	123	10%	20%	30%
processing capability	123	10%	20%	30%
RTT	ABC	high	low	high
avilability	ABC	1	0	1
load	123	50%	50%	50%
trusted	ABC	1	0	1
status	ABC	0	0	0

Figure 5-5. A Screenshot of the Process of Preparing the Data Source

To measure the actual performance of our decision-making model, especially if new scenarios appeared, a dataset splitting to training and testing subsets is required. The splitting percentage varies based on various models/problems and correlations between data. Albeit, the (80/20) or (70/30) splitting percentage is a good start to create prediction models [102]. As shown in Figure 5-6, we have used 80% of the dataset for training and 20% for testing. We have selected the random splitting to cover all the scenarios, rather than the liner splitting which assigned the first scenarios for training, while the last scenarios for testing. Nevertheless, in the random splitting, duplicated instances may have been used in both training and testing subsets, which will lead to unrealistic or incorrect predictions. We have addressed this by removing all the duplicated instances during the testing phase.



Name		
offloading scenarios big dataset 05-09-2019 … 80 instances, 11 fields (7 categorical, 4 numeric…		
offloading scenarios big dataset 05-09-2019 … 20 instances, 11 fields (7 categorical, 4 numeric…		
offloading scenarios big dataset 05-09-2019 100 instances, 11 fields (7 categorical, 4 numeri…		
offloading scena…| Training (80%) with offlo… 16 instances, 11 fields (5 categorical, 5 numeric…		
offloading scenarios | Test (20%) with offloa… 4 instances, 10 fields (5 categorical, 4 numeric, …		
offloading scenarios | Test (20%) with offloa… 4 instances, 10 fields (5 categorical, 4 numeric, …	1m	259 b

Figure 5-6. Original, Training and Testing Datasets

5.2.3. DL Decision-making Model

This section demonstrates the implementation details of designing our DL decision-making model, so to make intelligent decisions/predictions of what and where to offload, as shown in Figure 5-7. It details the deployed DNN model, including decisions of the network structure, (i.e. hidden layers, normalisation weights, the used algorithm and activation function).

5.2.3.1. Our deployed DNN Model

We have used BigML to launch a DEEPNET model; an optimised version of DNN that enables an automatic network search of various networks so to select the best among them that is likely to give the most accurate results. DNN consists of one or more layers, each layer contains multiple nodes, each node is a function that transforms an input into another value by multiplying it with a certain/random weight, as will be explained in the following paragraphs. Then it passes the values to the intermediate hidden layers, the process continues until a decision is reached and the output is presented.

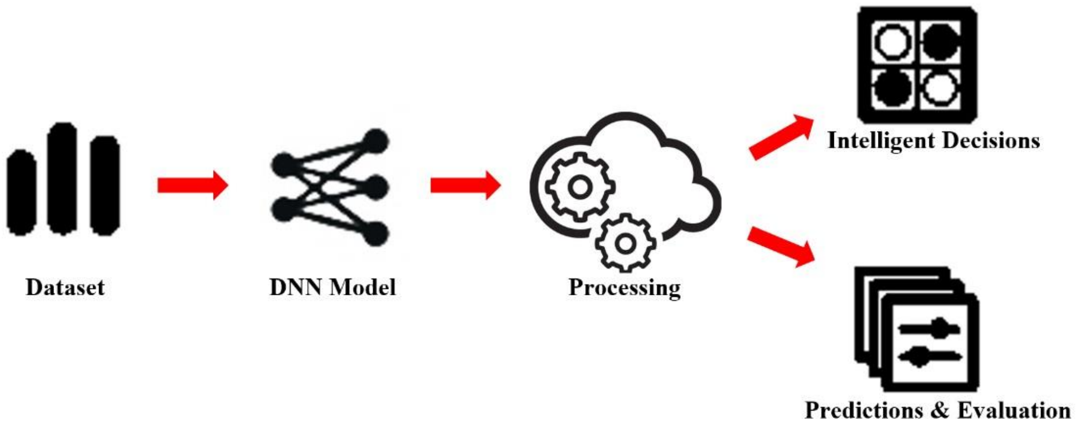
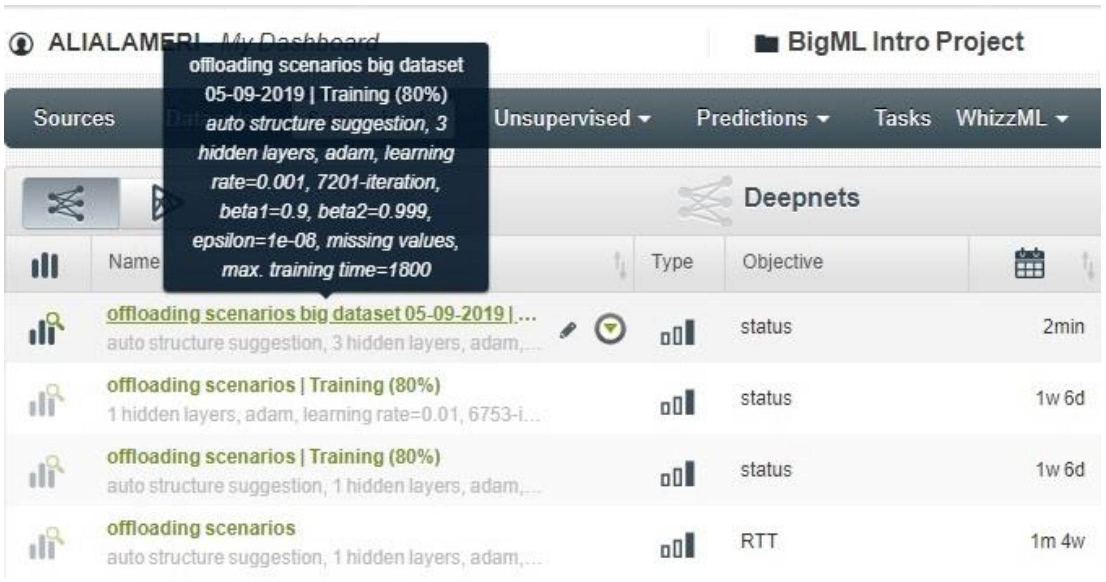


Figure 5-7. Our DL Decision-making Model

To configure the structure of DNN, we have used the automatic network search to select/form the best network decisions so to eliminate the time complexity of manual fine-tuning, which eventually gave us a better prediction score. As a result, we have used three hidden layers, Adam algorithm and random weight for normalisation, as shown in Figure 5-8 and explained below in the following subsections. We have trained the model for three hours, although the network may stop earlier if the performance does not show further improvements, we found that using the training time choice has given us better prediction results, compared to setting a specific number of iterations.



offloading scenarios big dataset 05-09-2019 | Training (80%)
auto structure suggestion, 3 hidden layers, adam, learning rate=0.001, 7201-iteration, beta1=0.9, beta2=0.999, epsilon=1e-08, missing values, max. training time=1800

Name	Type	Objective	
offloading scenarios big dataset 05-09-2019 | ... auto structure suggestion, 3 hidden layers, adam,...		status	2min
offloading scenarios | Training (80%) 1 hidden layers, adam, learning rate=0.01, 6753-i...		status	1w 6d
offloading scenarios | Training (80%) auto structure suggestion, 1 hidden layers, adam,...		status	1w 6d
offloading scenarios auto structure suggestion, 1 hidden layers, adam,...		RTT	1m 4w

Figure 5-8. DNN Model Network Structure

1. Field of Importance

The objective field is the field we want the model to be able to predict, (i.e. by default, the last input field is normally assigned as the objective field), we have assigned the engagement status as our objective field, and therefore it is labelled with values of 1 if it is an offloading candidate and 0, else wise. The DNN model observes all the input fields and establishes which input field has the most influence on the objective field, by defining the field of importance. The field of importance is the relative contribution of each of the other input fields, (e.g. battery capacity, processing capability, RTT latency, etc), on the objective field. It is calculated by taking a weighted average for each input field to reduce the predicted error. Each field is normalised to take a value between (0% – 100%), where all the input fields sum a total of 100%. The DNN model continually processes back and forth all the time to discover this, it stops once it reached the lowest predicted error rate, and therefore no further improvement is discovered. Some inputs, (e.g. availability), have higher importance on the objective field than others since it has a value of 0/1 on the prediction model.

2. Hidden Layers

In DEEPNET, each node is a function that transforms an input into another value by multiplying it with a certain/random weight, then it passes the values to the hidden layers, the hidden layers are intermediate layers between the input layer and the output layer, these layers are connected to learn and extract features continually from observing data. The DEEPNET allows to use up to 32 hidden layers with 8,192 nodes/neurons per layer. The decision of how many layers to use is crucial and varies based on testing, and therefore an optimisation balance should be considered, to achieve a good prediction performance, while eliminating overfitting. After testing various configurations and to fit our solution scenarios, we have used three hidden layers with 64, 128 and 64 nodes for the first, second and third hidden layers respectively. This is because (a) the higher number of nodes, the higher possibility of overfitting, and (b) few nodes will eventually lead to poor performance or incorrect results.

3. Used Algorithm

The algorithm is used to optimise the network and minimise the loss function, DEEPNET provides many gradient descent algorithms, (e.g. Momentum, Adagrad, RMSProp, Adam and FTRL) We have used Adaptive Moment Estimation (Adam) to optimise our model, it is an adaptive algorithm that is used to solve prediction problems [16]. Adam normally outperforms other algorithms, it converges itself faster and can solve various problems that other algorithms may not be able to solve. According to [103] "Adam outperforms the rest of DEEPNET algorithms due to its bias correlation".

4. Activation Function

The activation function is used to represent a linear or non-linear relationship between the input and output fields, that is necessary to solve complex problems. DEEPNET provides several activation functions, (e.g. Tanh, Sigmoid, ReLU, Softplus and SoftMax). We have used ReLU activation function in our testing because (a) it is the most common function used in any DNN model [104], (b) it performs well with Adam algorithm and is likely to give accurate results for supervised structure models, and (c) it converts an input into an output, which then can be used again as an input for the next layer of the network and so on.

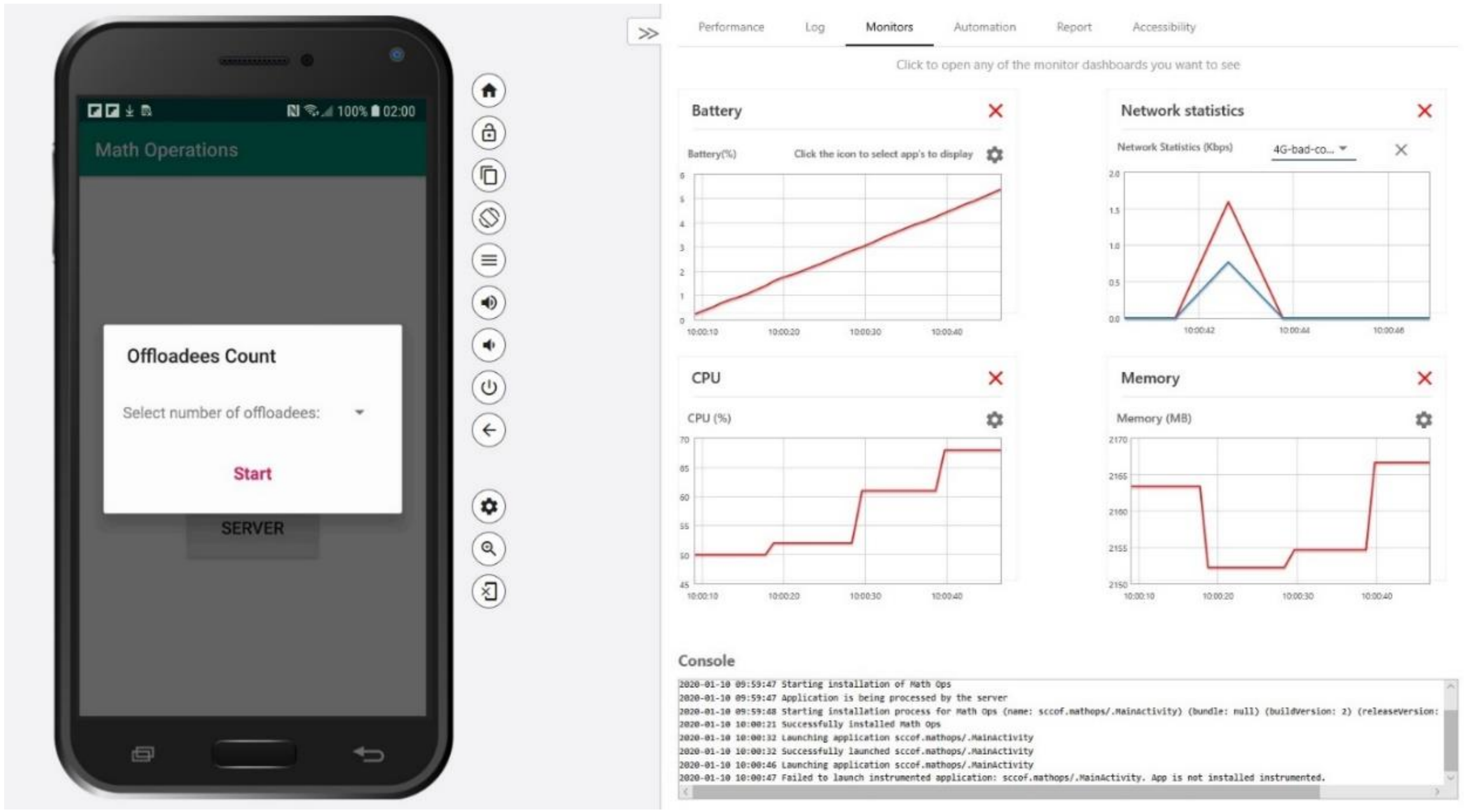
5. Normalisation Weight

The weight factor is used to normalise the input values before fed into the activation function, and it is normally assigned a value between (0,1). We have randomly initialised the weight values, which are then multiplied by the input values before fed to the hidden layer. The learning rate is set to 0.001 so that the data can fit the training process, normally <1% gives better prediction results. Larger values may increase the training time as the network will have to repeat itself again and again until a certain level of acceptance is satisfied, as well as to prevent nodes from dropping out during the training process.

5.2.4. Performance Scoring Algorithm

We have developed a simple algorithm to calculate the performance score of the assisting-MSDs used in the experiments. We have created about 600 scenarios to emulate real-time trials. These scenarios are collected from possible behaviours that might occur during the offloading process and actual testing scenarios conducted previously in our implementation. As well as decisions of sub-tasks complexity, offloadability and granularity achieved during the sampling profiling, as shown in Figure 5-9. i.e. Each scenario inputs 10 parameters/features, (battery level, processing capability, number of assisting-MSDs and sub-tasks, sub-tasks granularity, network type, RTT latency, current load and availability). Some parameters are fetched directly from the host-MSD physical hardware, (e.g. battery capacity and CPU processor cycles/load). i.e. We have defined a threshold for each of these, for battery capacity and processing capability, we set three thresholds which depend on the current level, if between (10-30%), then it scores 0, (31-60%) it scores 5 and (61-90%) it scores 10. These performance thresholds are used to calculate the total performance score of each of the assisting-MSDs that joined the network and want to participate to help the host-MSD processing CIAS sub-tasks. Based on the score, the most intensive sub-task is allocated to the MSD with the highest score, any device scores a performance threshold of 40% or less will be discarded from the network, as not being qualified to help the host-MSD to process the sub-tasks.

We have defined a performance threshold of 40% or less, as to avoid assisting-MSDs running out of battery or involved in some other activities while helping the host-MSD, which will eventually disbenefit our solution. For example, if we have 4 assisting-MSDs with a performance score as follow: MSD1 = 60, MSD2 = 72, MSD3 = 39 and MSD4 = 54. So, the most intensive sub-task or the biggest portion of the sub-tasks is allocated to MSD2, followed by MSD1 and MSD4, while MSD3 will be discarded from the network. For simplicity, we set 10% of the sub-tasks to execute on the host-MSD at all times.



```
2020-01-10 09:59:47 Starting installation of Math Ops
2020-01-10 09:59:47 Application is being processed by the server
2020-01-10 09:59:48 Starting installation process for Math Ops (name: sccof.mathops/.MainActivity) (bundle: null) (buildVersion: 2) (releaseVersion:
2020-01-10 10:00:21 Successfully installed Math Ops
2020-01-10 10:00:32 Launching application sccof.mathops/.MainActivity
2020-01-10 10:00:32 Successfully launched sccof.mathops/.MainActivity
2020-01-10 10:00:46 Launching application sccof.mathops/.MainActivity
2020-01-10 10:00:47 Failed to launch instrumented application: sccof.mathops/.MainActivity. App is not installed instrumented.
```

Figure 5-9. A Screenshot of Running Sampling Profiling

We have created a simple inference service to be positioned at the Edge by the host-MSD, the inference service is an offloading UI that can be invoked seamlessly on all the MSDs, so to exchange a list of features among the devices, (e.g. battery level and processing capability), and to update the DB regularly. The AI-engine makes all the decisions automatically and then invokes the ECR-engine to perform the offloading as explained in Chapter 4. We have used 4 MSDs and 2 servers to conduct the experiment, but we presume more devices can join the network especially when more intensive sub-tasks are required by the host-MSD. We have enabled 4 scenarios in the UI, where all the sub-tasks execute locally on the host-MSD, or generating the ECR-engine to execute the sub-tasks in parallel among the assisting-MSDs, and also by using a CS and an Edge server. Albeit, multiple servers can be used in the experiments, it only requires a server IP address to start the connection, and post API to post/get sub-tasks to/from the server. Using the performance scoring algorithm instead of the distribution algorithm explained in Section 4.2.2.4, for the same testing scenarios of the OPI as detailed in Section 4.3, have improved the achieved results. Our achieved results obtained by deploying the ECR-engine alone have been improved by 16% when including the AI-engine in the test. This is important because the AI-engine will make all the required decisions before invoking the ECR-engine that will form the resource network and start the offloading process. The CS scenario is included for evaluation purpose to compare our solution with other relevant implementations.

5.2.5. Testing the DNN Model and Achieved Results

After creating the runtime environment including the training and testing datasets, performance scoring algorithm, offloading UI, DNN model and configuring the network structure/parameters successfully. This section details the process of building and testing the prediction and scheduling scenarios, followed by the results achieved from conducting the scenarios.

5.2.5.1. A prediction Scenario to Offload or Not

First, we have created a dataset of 100 offloading scenarios, the scenarios emulate real-time trials that happened during the offloading process, including various MSDs and servers. As mentioned earlier each scenario inputs 10 parameters, (i.e. battery level, processing capability, number of assisting-MSD and sub-tasks, sub-tasks granularity, network type, RTT latency, current load and availability). The dataset is split as 80% for training and 20% for testing as explained in Section 5.2.2. Then we have deployed a prediction model to decide dynamically to offload or not, based on the correlations and importance of the inputs on the objective field. After creating the prediction model, we then input the training dataset to train the model as shown in Figure 5-10.

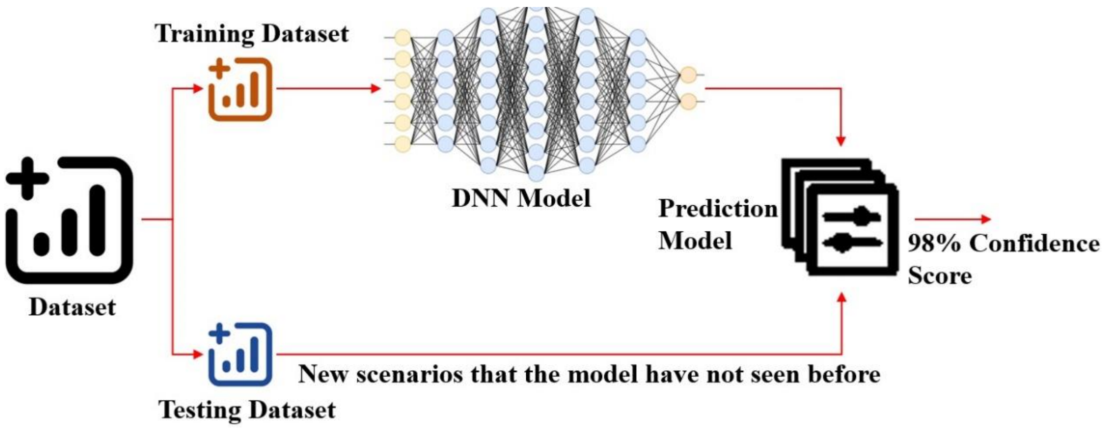
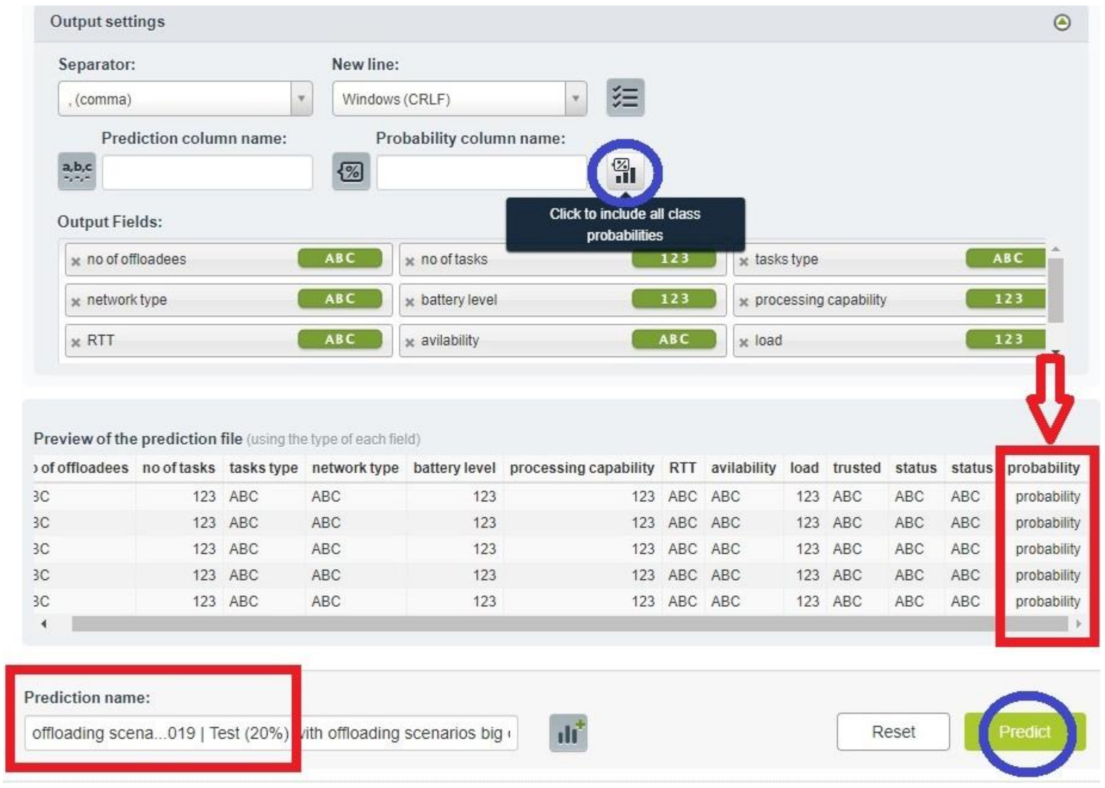


Figure 5-10. A Prediction Scenario to Perform Intelligent Decisions

After the training process is completed, the model was able to predict accurate results with up 98.2% as all the scenarios were labelled beforehand. Then we tested the model with scenarios that the model did not see before, and check if the model can produce accurate predictions based on the learning process gained during the training phase. We then input the testing dataset and removed all the labelling from the objective field, also we added the probability class to observe individual prediction scores, as shown in Figure 5-11. The model was able to learn and predict the best offloading scenario amongst other scenarios, or advise to not offload at all if it encounters scenarios with poor inputs, (e.g. a device has a battery of 10%), the prediction is based on the gained learning obtained during the training process, with up to 99% accuracy.


Output settings

Separator: , (comma)

New line: Windows (CRLF)

Prediction column name:

Probability column name:

Click to include all class probabilities

Output Fields:

no of offloadees ABC	no of tasks 123	tasks type ABC
network type ABC	battery level 123	processing capability 123
RTT ABC	avilability ABC	load 123

Preview of the prediction file (using the type of each field)

of offloadees	no of tasks	tasks type	network type	battery level	processing capability	RTT	avilability	load	trusted	status	status	probability
3C	123	ABC	ABC	123	123	ABC	ABC	123	ABC	ABC	ABC	probability
3C	123	ABC	ABC	123	123	ABC	ABC	123	ABC	ABC	ABC	probability
3C	123	ABC	ABC	123	123	ABC	ABC	123	ABC	ABC	ABC	probability
3C	123	ABC	ABC	123	123	ABC	ABC	123	ABC	ABC	ABC	probability
3C	123	ABC	ABC	123	123	ABC	ABC	123	ABC	ABC	ABC	probability

Prediction name: offloading scena...019 | Test (20%) vith offloading scenarios big

Figure 5-11. Adding the Probability Class to the Testing Dataset

5.2.5.2. A Schedular Scenario to decide What and Where to Offload

We have created a more complex dataset of 600 scenarios that emulate real-time trails of previous executions as explained previously in Section 5.2.4. In this scenario, we have built a scheduler model as shown in Figure 5-12, to schedule CIAS sub-tasks among the assisting-MSDs, based on a developed performance scoring algorithm. The scheduling is determined based on the device having at the time the highest performance score while discarding the device with a threshold performance score of 40% or less as not being qualified to help the host-MSD.

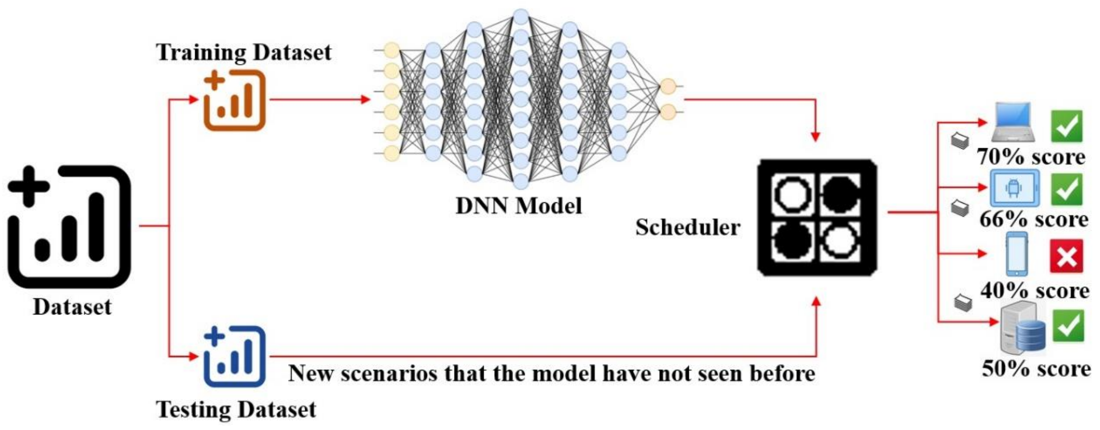
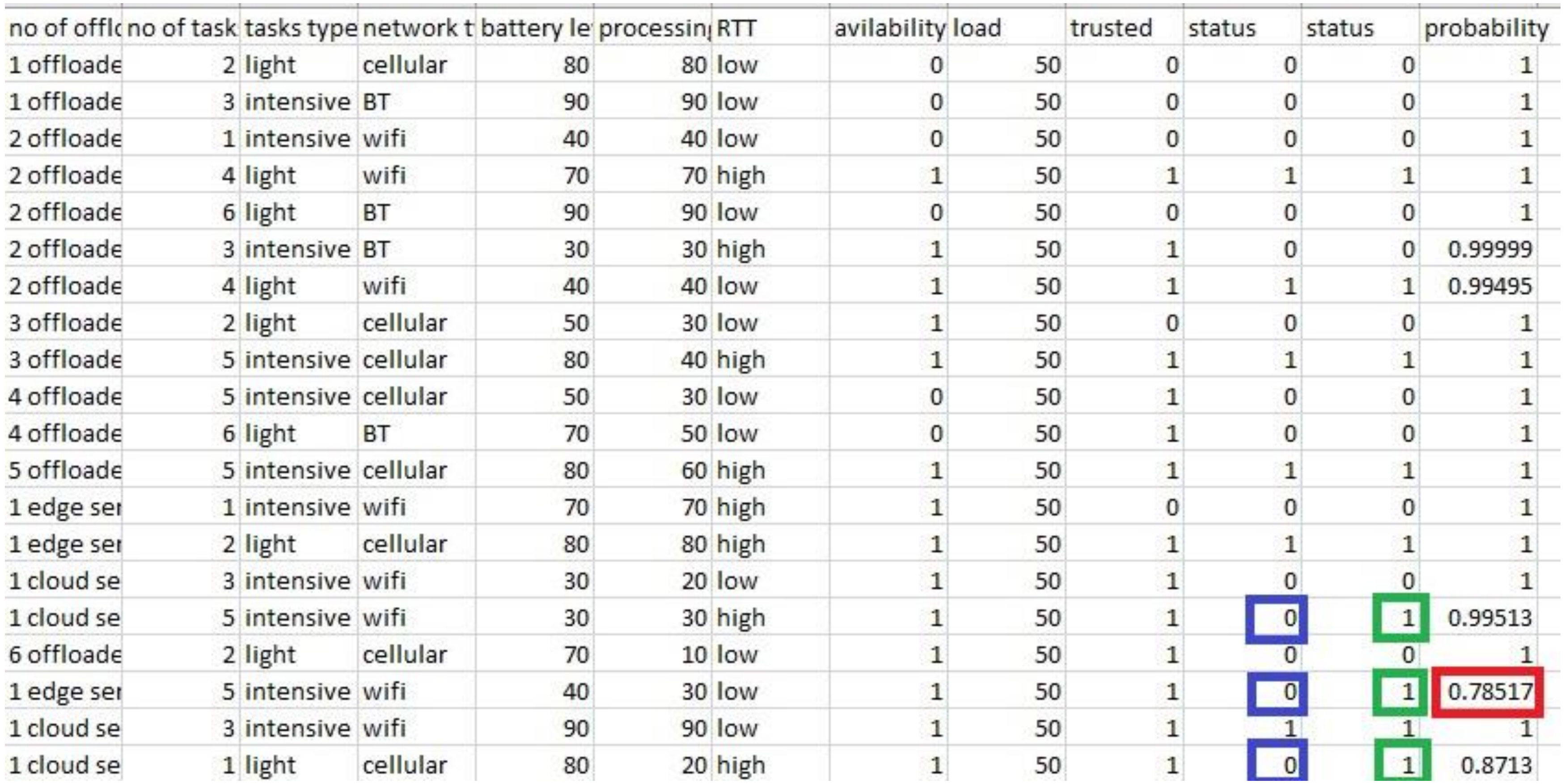


Figure 5-12. Scheduling Sub-tasks to execute among the Assisting-MSDs

Similar to the prediction scenario, we have trained the model using the training dataset (480 scenarios), and then tested the model using the testing dataset (120 scenarios). We have used up to five assisting-MSDs, two servers and ten intensive sub-tasks. The model was able to schedule all the sub-tasks to be assigned to the assisting-MSDs successfully with up to 99% accuracy.

Figure 5-13 shows the output of the prediction model, the model was able to predict the new scenarios with a maximum of 99% and a minimum of 78% confidence score. Only 3 of the scenarios were predicted incorrectly, this is because we used a dataset of 100 scenarios only. i.e. The model needs more scenarios to enhance the accuracy prediction while reducing the error rate. This is then addressed in the scheduler model testing, where we created a dataset of 600 scenarios and fed it to the model while observing the performance of the model.



no of offlc	no of task	tasks type	network t	battery le	processin	RTT	avilability	load	trusted	status	status	probability
1 offloade	2	light	cellular	80	80	low	0	50	0	0	0	1
1 offloade	3	intensive	BT	90	90	low	0	50	0	0	0	1
2 offloade	1	intensive	wifi	40	40	low	0	50	0	0	0	1
2 offloade	4	light	wifi	70	70	high	1	50	1	1	1	1
2 offloade	6	light	BT	90	90	low	0	50	0	0	0	1
2 offloade	3	intensive	BT	30	30	high	1	50	1	0	0	0.99999
2 offloade	4	light	wifi	40	40	low	1	50	1	1	1	0.99495
3 offloade	2	light	cellular	50	30	low	1	50	0	0	0	1
3 offloade	5	intensive	cellular	80	40	high	1	50	1	1	1	1
4 offloade	5	intensive	cellular	50	30	low	0	50	1	0	0	1
4 offloade	6	light	BT	70	50	low	0	50	1	0	0	1
5 offloade	5	intensive	cellular	80	60	high	1	50	1	1	1	1
1 edge ser	1	intensive	wifi	70	70	high	1	50	0	0	0	1
1 edge ser	2	light	cellular	80	80	high	1	50	1	1	1	1
1 cloud se	3	intensive	wifi	30	20	low	1	50	1	0	0	1
1 cloud se	5	intensive	wifi	30	30	high	1	50	1	0	1	0.99513
6 offloade	2	light	cellular	70	10	low	1	50	1	0	0	1
1 edge ser	5	intensive	wifi	40	30	low	1	50	1	0	1	0.78517
1 cloud se	3	intensive	wifi	90	90	low	1	50	1	1	1	1
1 cloud se	1	light	cellular	80	20	high	1	50	1	0	1	0.8713

Figure 5-13. Confidence Score for the Prediction Model

Similarly, Figure 5-14 shows the output correlation of the schedular model, the model was able to schedule all the sub-tasks to be assigned to the assisting-MSDs successfully, based on the learning gained during the training phase.

The best assisting-MSD with the highest performance score was selected while the device with a 40% or less performance score was discarded. The model achieved up to 99% accuracy, especially when the size of the training dataset increases.

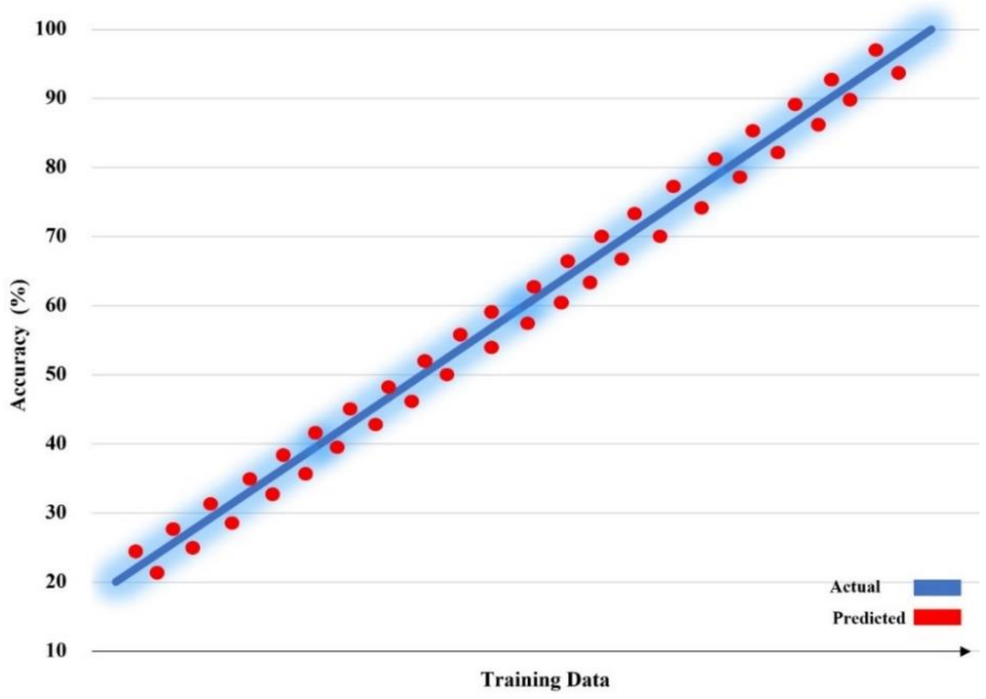


Figure 5-14. The Accuracy of the Actual and Predicted Scenarios

Also, the output results of the prediction/schedular model can be downloaded and used as a new input dataset to be fed again to the model to improve the learning. This process is called propagation, to fine-tune the model for further testing if needed. We did not test this process, as we did not find it necessary to fine-tune the model again, as almost 99% of the scenarios were predicted correctly, with <1% error rate.

We have concluded that (1) deploying an AI offloading engine is important to perform dynamic and adaptive decisions without prior planning, so to boost-up the static decision of what and where to offload made by typical offloading frameworks. (2) A DNN model that can form an automatic network search to select the best network structure is essential, so to reduce the time complexity and processing requirements introduced by the manual fine-tuning process. (3) A CS is necessary to host the training and validation of the DNN model, so to enhance the host-MSD performance/functionality by eliminating the cost overhead of deploying such model on the host-MSD or the ES.

The implementation of the AI-engine is based on a pre-build existing DNN model. i.e. We did not program the algorithm, we only used it to train and test the dataset, albeit, we did change some parameters/structure for better prediction results and to fulfil our solution sub-tasks. The implementation and testing prove that our model schedules all the CIAS sub-tasks to be allocated to the best assisting-MSDs, based on the learning gained during the training phase. The best assisting-MSDs with the highest performance score was selected while the MSD with a 40% or less performance score was discarded. The model successfully achieved/formed all the decisions with up to 99% accuracy while <1% error rate, especially when the size of the training dataset increases.

6. Conclusion and Future Work

This chapter concludes the research work done in this thesis, as well as:

1. A summary of the main tasks/contributions that are unique to this thesis only, (see Section 6.2).

2. The future work that highlights potential future research directions, (see Section 6.3).

6.1. Conclusion

We visualised that Apps for millions of instances at any one second needs to be processed, and is driven by (a) smart cities with the use of the Internet of Everything (IoE) where people, data, process and things are connected all together [105]. (b) Heterogonous environments where it is challenging to pre-predict the computing resources, as well as the processing and/or network requirements, (i.e. if the Cloud server is inaccessible due to network availability/latency, server outage or even cost).

We believe the trend of more and more mobile Cloud services and/or IoT Apps will continue to rise, especially to cope with the rollout of (i) AI models, (ii) Edge deployments and (iii) the future of wireless network capabilities. Our vision on such trend, is the future networks, (e.g. 5G and beyond), will enable and simplify the implementation of AI models on local Edge devices without the need of the Cloud, so to eliminate the extra overhead cost, latency and network traffic.

We can equally visualise that our solution, when re-deployed commercially, can elevate the execution of such sub-tasks by forming and dismantling the "computing resource on the Edge near the host/end-node" in millions everyone second to solve user's computational needs where the app needs the help of nearby devices. Our contribution, being a research study, was focused on establishing a "proof of concept" for this vision. We do realise that major efforts are required to take this level of work into a commercially viable product, but we believe that this concept and its associated concepts of "devices working cooperatively to aid achieving bigger sub-tasks" and "forming resources on the go when and where needed" are realistic and will find their way into most of our future computational needs.

The sharing of resources thus far has been solely based on infrastructure deployments of which is typically pre-planned on metrics for so use-scenarios. The only way out of this is to form an on-the-go resource network, when it is needed, for a specific sub-task(s) and where the sub-task(s) is, these are points which have been ignored by the previous infrastructure-based provision.

The cooperation issues are also revolutionary as people tend to be reserved when sharing their devices, however, we are taking the "sharing theme" beyond its boundaries, some would say. However, like the early day's vision of sharing photos on Facebook was perceived at the beginning. While we were deploying these concepts, Google has released their 1-meter accuracy localisation service, (i.e. using emerging location APIs and/or ML for Wi-Fi locations [106]), which is amongst others, uses information from other SP users to achieve this accuracy, which otherwise is impossible based solely on GNSS.

In the same vein, we expect that our concept will be acceptable to users once they learn that they equally can be the beneficiary of such resource and when confidence gained in the "AI-engine" we proposed that is being fare, cautious and secure/private. There is no doubt that AI and automated systems are the new revolutions that will open the light to deploy more and more intelligent systems, which will eventually take us to a different level of thinking and forming predictions/decisions in seconds. Albeit, more work needs to be done to tailor AI in different ways so to fit/solve more tasks and problems, (e.g. RL and federated learning).

Part of my research was the efficiency of wireless networks to instantaneously form/deform service networks as we require for our proof of concept. The issue of offloading on-the-go using wireless connectivity is so vital for our future diverse tasks, this issue needs to be addressed properly. One of the thoughts was to perhaps come up with a unique "offloading networking model" and one of its main layers needs to focus on addressing the issue of forming/deforming for this quick in/out to simplify helping nearby devices performing offloading. I have covered researching for offloading granularity, however, this does not mean it is ideal, and one of the issues that I still think about is "why having big tasks in the first place" and "when do programming codes" can partition their selves automatically without manual finetuning, or "why such service knowledge is not available online from the outset so to eliminate the issue of understanding and monitoring these services constantly".

6.2. Thesis Summary

The "issue" that this research work is set out to address is that offloading "the processing of CIASs workload" from a host-MSD to a CS for execution causes (a) extra time delay, (b) extra connectivity cost and (c) extra network traffic congestion, thus resulting in rapid battery power consumption and degradation in performance and/or functionality.

To resolve this "issue", we have developed a novel solution of assembling a local "Computing Resource" when needed, formed by on-the-go networking of nearby assisting-MSDs that have been adopted cooperatively in a previous step, (we call this newly formed ECR-engine). That is, we have deployed a Cloud-based AI-engine, that will have a record of all the MSDs subscribed for this cooperative scheme, but most importantly the AI-engine will advise the host-MSD about the device's availability, processing resources and engagement status. Based on the instructions taken by the AI-engine, the ECR-engine will then form the resource network and start to offload the sub-tasks among the assisting-MSDs and retrieve the results upon completion.

This solution has taken five years to accomplish putting and experimenting with the various components of it. A summary of the main tasks/contributions that we needed to achieve so to build our solution are as follow:

- Practically validating four Cloud offloading frameworks, which have helped us to build our first framework that offloads CIASs sub-tasks from a host-MSD to a CS for execution. This includes (a) conducting a pilot study to offload various types and sizes of sub-tasks using different wireless network technologies and (b) measuring the latency of offloading for various wireless connectivity. All of these are detailed in Chapter 3.
- Developing an intelligent solution for profiling CIAS, as well as available computing resources in recruiting assisting-MSDs, and partitioning the CIAS workload in suitable chunks/sub-tasks, so to share these sub-tasks amongst the assisting-MSDs. This is achieved by (a) implementing a peer-to-peer connectivity. (b) Using a DFS algorithm to cluster a group of offloadable sub-tasks together that

can be executed on the assisting-MSDs. (c) Developing an OPI that can be invoked on all the MSDs, so to perform offloading trials. (d) Implementing a simple cost estimator by defining IPW values to estimate the cost overhead of the processing time, battery power, latency and efficiency. All of these are detailed in Chapter 4.

- To ensure that I deploy the most efficient AI-engine, we conducted a study to investigate various ML and DL algorithms. It was concluded by building a DL decision-making model to (a) train and test the dataset, (b) predict the next available MSD to select and make intelligent decisions to offload or not, (c) if it decides to offload, then it schedules the sub-tasks among the assisting-MSDs, using a DNN and a performance scoring algorithm. Please see Chapter 5 for details.

- The dataset had to explore as many offloading scenarios as possible. To support this versatility, we built a dataset of 600 scenarios from profiling CIAS sub-tasks and the processing resources of assisting-MSDs, as well as by using a prebuild DB of sub-tasks previous executions. The dataset is split to training and testing datasets used to train and test the DNN model. Each scenario inputs (battery level, processing capability, number of assisting-MSDs and sub-tasks, sub-tasks granularity, network type, RTT latency, current load and availability). Please see Chapter 5 for details.

- On the personal development side, (i) I have participated in teaching CC practical sessions for 3 autumn terms during my research, (2017, 2018 and 2019). (ii) I have published three papers in highly-ranked events that enabled me to network with like-minded scientists. (iii) I have attended and participated in various academic and industrial conferences, workshops and seminars, so to engage with like-minded researchers and developers to expand my skills.

Our implemented solution proves to be reliable, accurate and efficient, compared to other static Cloud frameworks that are formed based on a pre-planning of the overall infrastructure. The results show that on average our solution achieved 40.2% more efficient processing time, 28.8% less battery power consumption and 33% reduce latency. As well as 99% confident score for performing the recognition task and 98% prediction accuracy for the deployed DNN model.

6.3. Future Work

This thesis opens the light for potential future directions, and so our novel solution can be extended to consider the following:

1. Our solution does not claim to introduce a security solution, albeit, it provides a secure deployment by implementing secure techniques such as AWS Rekognition and Nearby API, as explained previously in Section 2.1. However, to make our solution even more robust, further measures are necessary so to avoid some malicious activities that might occur during the offloading process. Our solution can be extended by adding a security engine to the main offloading elements; profiling, partitioning and decision making. The security engine can be used to secure the communication and authenticate the devices, and therefore authentication protocols and/or session keys as in [46] and [48] can be used and exchanged before offloading and sharing the sub-tasks through the network. Therefore, the decision of offloading or not can only be allowed if the "resource network, its connectivity and granularity" satisfy the security engine as well as the offloading criteria. For commercial viability, such secure undertaking has to take place.

2. As discussed in my conclusion, it is vital that the processing of the AI-engine does not cause overhead on any side of the "resource network". Therefore, a re-deployment of the AI-Engine based on RL algorithms is paramount to ensure that a fair load balance achieved with maximum accuracy based on the ever-reformats of the resource network. Therefore, the AI-engine can learn from experience without the need to train and test pre-build dataset, (e.g. in the absence of the training dataset). The typical advantageous of these algorithms are:

- Inspire by the behavioural state so to make dynamic decisions within/from the deployed environment.
- Try all possible runs, test multiple scenarios randomly and repeatedly until they reach the best outcome, and therefore form the structure of the algorithm based on the gained learning, using mathematical equations to calculate the State, Action and Reward functions.

Q learning algorithm [107] is a good example of RL that takes a suitable action to make a reward, this also can be used to implement a reward system that can be devised between MSDs and can be exchanged for monies or other sharing schemes, so to encourage more MSDs to participate and share their unused resources. A further investigation to evaluate the prediction performance of such algorithm is necessary, as this type of algorithms are mainly based on experience, as well as further implementation and testing are required, so to discover if such algorithms can be fine-tuned if the initial prediction results are inaccurate.

3. One of the main issues that have constantly played on my mind is the applications that are most suited to our end2end solution. Obviously, as I come across/learn any new developing technology, I tend to form a scenario for having this solution handling the new application area. For this part, I include some scenarios are:

- Wearable Devices; are being very important topic and becoming essential in handling various types of tasks which we performed on a daily basis. These devices, (e.g. Google Glass, Smart Galaxy Smartwatch, etc.), are helping us to perform a plethora of applications such as health monitoring, sports activities, reality augmentation and many more [108]. However, such devices are normally occupied with limited processing resources, (e.g. battery capacity, processor capability and storage), which makes it impossible to execute some intensive sub-tasks. We believe our solution can be tailored and deployed to address this issue, as to offload such sub-tasks to the host-MSD with the help of nearby assisting-MSDs to handle the execution, or even by using the host-MSD as a middleware layer to offload the sub-tasks to the CS.
- Vehicular Networks; a new emerging topic and yet it still has limitations that need to be addressed. Vehicular networks use advance communication technologies to connect with other devices, so to exchange data among them or with the cellular BS or other fixed infrastructures. We believe our solution could help vehicles to communicate with other nearby vehicles and exchange data/sub-tasks in a sharing environment. This is very important, to eliminate the time-wasting from going to the Cloud, as well as to learn/benefit from the knowledge/experience of other nearby vehicles.

References

[1] "State of the IoT 2018: Number of IoT devices now at 7B – Market accelerating." https://iot-analytics.com/state-of-the-iot-update-q1-q2-2018-number-of-iot-devices-now-7b/ (accessed May 06, 2020).

[2] P. Sethi and S. R. Sarangi, "Internet of Things: Architectures, Protocols, and Applications," *Journal of Electrical and Computer Engineering*. 2017, doi: 10.1155/2017/9324035.

[3] "With Internet Of Things And Big Data, 92% Of Everything We Do Will Be In The Cloud." https://www.forbes.com/sites/joemckendrick/2016/11/13/with-internet-of-things-and-big-data-92-of-everything-we-do-will-be-in-the-cloud/#5f4e3bfc4ed5 (accessed Apr. 23, 2020).

[4] "The Growth in Connected IoT Devices Is Expected to Generate 79.4ZB of Data in 2025, According to a New IDC Forecast." https://www.idc.com/getdoc.jsp?containerId=prUS45213219 (accessed Apr. 23, 2020).

[5] S. Saad and S. Nandedkar, "Energy Efficient Mobile Cloud Computing," *Ijcsit.Com*, vol. 5, no. 6, pp. 7837–7840, 2014, [Online]. Available: http://www.ijcsit.com/docs/Volume 5/vol5issue06/ijcsit20140506206.pdf.

[6] K. Kim, J. Lynskey, S. Kang, and C. S. Hong, "Prediction based sub-Task offloading in mobile edge computing," *Int. Conf. Inf. Netw.*, vol. 2019-Janua, pp. 448–452, 2019, doi: 10.1109/ICOIN.2019.8718183.

[7] "Amazon EC2." https://aws.amazon.com/ec2/ (accessed May 06, 2020).

[8] K. Kumar, "Oriented Applications of Big Data Sentiment Analysis," *2018 3rd Int. Conf. Internet Things Smart Innov. Usages*, 2018.

[9] C. Song *et al.*, "Hierarchical edge cloud-based traffic offloading enabling low-latency in 5G optical and radio network," *Opt. InfoBase Conf. Pap.*, vol. Part F83-A, pp. 6–8, 2017, doi: 10.1364/ACPC.2017.M3B.6.

[10] M. Satyanarayanan, “Edge Computing,” *Computer*. 2017, doi: 10.1109/MC.2017.3641639.

[11] “The Benefits and Potential of Edge Computing.” https://www.vxchnge.com/blog/the-5-best-benefits-of-edge-computing (accessed Jul. 02, 2020).

[12] M. S. Mahdavinejad, M. Rezvan, M. Barekatain, P. Adibi, P. Barnaghi, and A. P. Sheth, “Machine learning for internet of things data analysis: a survey,” *Digital Communications and Networks*. 2018, doi: 10.1016/j.dcan.2017.10.002.

[13] Y. Lecun, Y. Bengio, and G. Hinton, “Deep learning,” *Nature*. 2015, doi: 10.1038/nature14539.

[14] Y. Liu, Q. Cui, J. Zhang, Y. Chen, and Y. Hou, “An Actor-Critic Deep Reinforcement Learning Based Computation Offloading for Three-Tier Mobile Computing Networks,” *2019 11th Int. Conf. Wirel. Commun. Signal Process. WCSP 2019*, pp. 1–6, 2019, doi: 10.1109/WCSP.2019.8927911.

[15] Y. Zhang, H. Liu, L. Jiao, and X. Fu, “To offload or not to offload: An efficient code partition algorithm for mobile cloud computing,” *2012 1st IEEE Int. Conf. Cloud Networking, CLOUDNET 2012 - Proc.*, pp. 80–86, 2012, doi: 10.1109/CloudNet.2012.6483660.

[16] BigML, *Classification and Regression with the BigML Dashboard*. 2016.

[17] “Nearby | Google Developers.” https://developers.google.com/nearby (accessed May 06, 2020).

[18] A. Al-ameri and I. A. Lami, “SCCOF : Smart Cooperative Computation Offloading Framework for Mobile Cloud Computing Services,” in *the 8th Annual International Conference on Big Data, Cloud and Security (ICT-BDCS 2017)*, 2017, pp. 9–13, doi: 10.5176/2251-2136.

[19] A. Al-ameri and I. A. Lami, “SOSE: Smart Offloading Scheme Using Computing Resources of Nearby Wireless Devices for Edge Computing Services,” in *Lecture Notes of the Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering, LNICST*, 2019, doi:

10.1007/978-3-030-23943-5_5.

[20] "Welcome To Colaboratory - Colaboratory." https://colab.research.google.com/notebooks/intro.ipynb (accessed May 06, 2020).

[21] "TensorFlow." https://www.tensorflow.org/ (accessed May 06, 2020).

[22] "BigML.com." https://bigml.com/ (accessed May 06, 2020).

[23] "Amazon SageMaker." https://aws.amazon.com/sagemaker/ (accessed May 06, 2020).

[24] I. A. Lami and A. Al-ameri, "DEO: A Smart Dynamic Edge Offloading Scheme Using Processing Resources of Nearby Wireless Devices to Form an Edge Computing Engine," in *the 3rd IEEE International Conference on Cloud and Fog Computing Technologies and Applications (IEEE Cloud Summit 2019)*, 2020, pp. 58–64, doi: 10.1109/cloudsummit47114.2019.00016.

[25] Y. S. Mezaal, H. H. Madhi, T. Abd, and S. K. Khaleel, "Cloud computing investigation for cloud computer networks using cloudanalyst," *J. Theor. Appl. Inf. Technol.*, 2018.

[26] "AWS Elastic Beanstalk – Deploy Web Applications." https://aws.amazon.com/elasticbeanstalk/ (accessed May 06, 2020).

[27] H. Y. Chen, Y. H. Lin, and C. M. Cheng, "COCA: Computation offload to clouds using AOP," *Proc. - 12th IEEE/ACM Int. Symp. Clust. Cloud Grid Comput. CCGrid 2012*, pp. 466–473, 2012, doi: 10.1109/CCGrid.2012.98.

[28] M. S. Ali, M. Ali Babar, L. Chen, and K. J. Stol, "A systematic review of comparative evidence of aspect-oriented programming," *Information and Software Technology*. 2010, doi: 10.1016/j.infsof.2010.05.003.

[29] E. Cuervoy *et al.*, "MAUI: Making smartphones last longer with code offload," *MobiSys'10 - Proc. 8th Int. Conf. Mob. Syst. Appl. Serv.*, pp. 49–62, 2010, doi: 10.1145/1814433.1814441.

[30] B.-G. Chun, S. Ihm, P. Maniatis, M. Naik, and A. Patti, "CloneCloud: Elastic

Execution between Mobile Device and Cloud," *Eur. Conf. Comput. Syst. Proc. Sixth Eur. Conf. Comput. Syst. EuroSys 2011, alzburg, Austria - April 10-13, 2011*, p. 301, 2011, doi: 10.1145/1966445.1966473.

[31] "NET | Free. Cross-platform. Open Source." https://dotnet.microsoft.com/ (accessed May 31, 2020).

[32] S. Kosta, A. Aucinas, P. Hui, R. Mortier, and X. Zhang, "ThinkAir: Dynamic resource allocation and parallel execution in the cloud for mobile code offloading," *Proc. - IEEE INFOCOM*, pp. 945–953, 2012, doi: 10.1109/INFCOM.2012.6195845.

[33] "The N-queens Problem | OR-Tools | Google Developers." https://developers.google.com/optimization/cp/queens (accessed May 31, 2020).

[34] M. Akram and A. Elnahas, "Energy-aware offloading technique for Mobile cloud computing," *Proc. - 2015 Int. Conf. Futur. Internet Things Cloud, FiCloud 2015 2015 Int. Conf. Open Big Data, OBD 2015*, pp. 349–356, 2015, doi: 10.1109/FiCloud.2015.45.

[35] D. Kovachev and R. Klamma, "Framework for Computation Offloading in Mobile Cloud Computing," *Int. J. Interact. Multimed. Artif. Intell.*, vol. 1, no. 7, p. 6, 2012, doi: 10.9781/ijimai.2012.171.

[36] J. Oueis, E. C. Strinati, and S. Barbarossa, "Multi-parameter decision algorithm for mobile computation offloading," *IEEE Wirel. Commun. Netw. Conf. WCNC*, vol. 3, pp. 3005–3010, 2014, doi: 10.1109/WCNC.2014.6952959.

[37] N. Of and N. Of, "M OBILE C LOUD C OMPUTING M IGRATE OR N OT ? E XPLOITING D YNAMIC T ASK M IGRATION IN M OBILE C LOUD C OMPUTING S YSTEMS," no. June, pp. 24–32, 2013.

[38] X. Wei *et al.*, "MVR: An Architecture for Computation Offloading in Mobile Edge Computing," *Proc. - 2017 IEEE 1st Int. Conf. Edge Comput. EDGE 2017*, pp. 232–235, 2017, doi: 10.1109/IEEE.EDGE.2017.42.

[39] P. Asghari, A. M. Rahmani, and H. H. S. Javadi, "Internet of Things

applications: A systematic review," *Comput. Networks*, 2019, doi: 10.1016/j.comnet.2018.12.008.

[40] S. B. Calo, M. Touna, D. C. Verma, and A. Cullen, "Edge computing architecture for applying AI to IoT," *Proc. - 2017 IEEE Int. Conf. Big Data, Big Data 2017*, vol. 2018-Janua, pp. 3012–3016, 2017, doi: 10.1109/BigData.2017.8258272.

[41] H. Li, K. Ota, and M. Dong, "Learning IoT in Edge: Deep Learning for the Internet of Things with Edge Computing," *IEEE Netw.*, vol. 32, no. 1, pp. 96–101, 2018, doi: 10.1109/MNET.2018.1700202.

[42] "7.1. Deep Convolutional Neural Networks (AlexNet) — Dive into Deep Learning 0.8.0 documentation." https://d2l.ai/chapter_convolutional-modern/alexnet.html#alexnet (accessed May 31, 2020).

[43] X. Chen, Q. Shi, L. Yang, and J. Xu, "ThriftyEdge: Resource-Efficient Edge Computing for Intelligent IoT Applications," *IEEE Netw.*, vol. 32, no. 1, pp. 61–65, 2018, doi: 10.1109/MNET.2018.1700145.

[44] A. Mtibaa, K. A. Harras, and A. Fahim, "Towards computational offloading in mobile device clouds," *Proc. Int. Conf. Cloud Comput. Technol. Sci. CloudCom*, vol. 1, pp. 331–338, 2013, doi: 10.1109/CloudCom.2013.50.

[45] K. Xiao, Z. Gao, Q. Wang, and Y. Yang, "A heuristic algorithm based on resource requirements forecasting for server placement in edge computing," *Proc. - 2018 3rd ACM/IEEE Symp. Edge Comput. SEC 2018*, pp. 354–355, 2018, doi: 10.1109/SEC.2018.00043.

[46] K. Ko, Y. Son, S. Kim, and Y. Lee, "DisCO: A distributed and concurrent offloading framework for mobile edge cloud computing," *Int. Conf. Ubiquitous Futur. Networks, ICUFN*, pp. 763–766, 2017, doi: 10.1109/ICUFN.2017.7993896.

[47] W. T. Su, C. S. Liang, and C. Y. Dai, "Secure computation offloading based on social trust in mobile networks," *Int. Conf. Ubiquitous Futur. Networks, ICUFN*, pp. 75–80, 2014, doi: 10.1109/ICUFN.2014.6876754.

[48] D. Shibin and G. J. W. Kathrine, “A comprehensive overview on secure offloading in mobile cloud computing,” *Proc. 2017 4th Int. Conf. Electron. Commun. Syst. ICECS 2017*, vol. 17, no. 1, pp. 121–124, 2017, doi: 10.1109/ECS.2017.8067851.

[49] “Amazon Rekognition – Video and Image - AWS.” https://aws.amazon.com/rekognition/ (accessed May 31, 2020).

[50] B. G. Chun and P. Maniatis, “Dynamically partitioning applications between weak devices and clouds,” *Proc. 1st ACM Work. Mob. Cloud Comput. Serv. Soc. Networks Beyond, MCS’10, Co-located with ACM MobiSys 2010*, 2010, doi: 10.1145/1810931.1810938.

[51] K. Peng, M. Zhu, Y. Zhang, L. Liu, V. C. M. Leung, and L. Zheng, “A multi-objective computation offloading method for workflow applications in mobile edge computing,” *Proc. - 2019 IEEE Int. Congr. Cybermatics 12th IEEE Int. Conf. Internet Things, 15th IEEE Int. Conf. Green Comput. Commun. 12th IEEE Int. Conf. Cyber, Phys. So*, pp. 135–141, 2019, doi: 10.1109/iThings/GreenCom/CPSCom/SmartData.2019.00044.

[52] “Eclipse Platform Overview | The Eclipse Foundation.” https://www.eclipse.org/eclipse/eclipse-charter.php (accessed May 31, 2020).

[53] D. Sulaiman and A. Barker, “MAMoC-Android: Multisite adaptive computation offloading for android applications,” *Proc. - 2019 7th IEEE Int. Conf. Mob. Cloud Comput. Serv. Eng. MobileCloud 2019*, pp. 68–75, 2019, doi: 10.1109/MobileCloud.2019.00017.

[54] A. C. Olteanu and N. Ţăpuş, “Offloading for mobile devices: A survey,” *UPB Sci. Bull. Ser. C Electr. Eng.*, vol. 76, no. 1, pp. 3–16, 2014.

[55] S. Saha, M. A. Habib, and M. A. Razzaque, “Compute intensive code offloading in mobile device cloud,” *IEEE Reg. 10 Annu. Int. Conf. Proceedings/TENCON*, pp. 436–440, 2017, doi: 10.1109/TENCON.2016.7848036.

[56] A. Mtibaa, K. A. Harras, K. Habak, M. Ammar, and E. W. Zegura, “Towards Mobile Opportunistic Computing,” in *Proceedings - 2015 IEEE 8th International Conference on Cloud Computing, CLOUD 2015*, 2015, doi:

10.1109/CLOUD.2015.163.

[57] I. Hwang, “Design and implementation of cloud offloading framework among devices for web applications,” *2015 12th Annu. IEEE Consum. Commun. Netw. Conf. CCNC 2015*, pp. 41–46, 2015, doi: 10.1109/CCNC.2015.7157944.

[58] “PeerJS - Simple peer-to-peer with WebRTC.” https://peerjs.com/ (accessed May 31, 2020).

[59] C. Lampe and T. Marshall, “CLOUD COMPUTING FOR MOBILE USERS: CAN OFFLOADING COMPUTATION SAVE ENERGY?,” *Publ. by IEEE Comput. Soc.*, no. April, pp. 51–56, 2010.

[60] H. Wu, Q. Wang, and K. Wolter, “Tradeoff between performance improvement and energy saving in mobile cloud offloading systems,” *2013 IEEE Int. Conf. Commun. Work. ICC 2013*, pp. 728–732, 2013, doi: 10.1109/ICCW.2013.6649329.

[61] I. A. Elgendy, M. El-Kawkagy, and A. Keshk, “Improving the performance of mobile applications using cloud computing,” *2014 9th Int. Conf. Informatics Syst. INFOS 2014*, pp. PDC109–PDC115, 2015, doi: 10.1109/INFOS.2014.7036687.

[62] F. Qiu, B. Zhang, and J. Guo, “A deep learning approach for VM workload prediction in the cloud,” *2016 IEEE/ACIS 17th Int. Conf. Softw. Eng. Artif. Intell. Netw. Parallel/Distributed Comput. SNPD 2016*, pp. 319–324, 2016, doi: 10.1109/SNPD.2016.7515919.

[63] “Deep Belief Networks — all you need to know.” https://medium.com/@icecreamlabs/deep-belief-networks-all-you-need-to-know-68aa9a71cc53 (accessed Jul. 02, 2020).

[64] T. Goyal, A. Singh, and A. Agrawa, “Cloudsim: Simulator for cloud computing infrastructure and modeling,” in *Procedia Engineering*, 2012, doi: 10.1016/j.proeng.2012.06.412.

[65] Nam Tuan Nguyen, Yichuan Wang, Husheng Li, Xin Liu, and Zhu Han, “Extracting typical users’ moving patterns using deep learning,” *GLOBECOM*

- *IEEE Glob. Telecommun. Conf.*, pp. 5410–5414, 2012, doi: 10.1109/GLOCOM.2012.6503981.

[66] B. Del Monte and R. Prodan, “A scalable GPU-enabled framework for training deep neural networks,” *2016 2nd Int. Conf. Green High Perform. Comput. ICGHPC 2016*, pp. 1–8, 2016, doi: 10.1109/ICGHPC.2016.7508071.

[67] “Grafana: The open observability platform | Grafana Labs.” https://grafana.com/ (accessed May 31, 2020).

[68] V. Mnih *et al.*, “Human-level control through deep reinforcement learning,” *Nature*, 2015, doi: 10.1038/nature14236.

[69] S. Tarkoma, M. Siekkinen, E. Lagerspetz, and Y. Xiao, *Smartphone energy consumption: Modeling and optimization*. 2014.

[70] “ns-3 | a discrete-event network simulator for internet systems.” https://www.nsnam.org/ (accessed Jun. 24, 2020).

[71] “NetAnim - Nsnam.” https://www.nsnam.org/wiki/NetAnim (accessed Jul. 02, 2020).

[72] “The CLOUDS Lab: Flagship Projects - Gridbus and Cloudbus.” http://www.cloudbus.org/cloudsim/ (accessed Jun. 24, 2020).

[73] B. Wickremasinghe, R. N. Calheiros, and R. Buyya, “CloudAnalyst: A cloudsim-based visual modeller for analysing cloud computing environments and applications,” *Proc. - Int. Conf. Adv. Inf. Netw. Appl. AINA*, pp. 446–452, 2010, doi: 10.1109/AINA.2010.32.

[74] “Global Infrastructure Regions & AZs.” https://aws.amazon.com/about-aws/global-infrastructure/regions_az/ (accessed Jul. 02, 2020).

[75] B. Wickremasinghe and others, “Cloudanalyst: A cloudsim-based tool for modelling and analysis of large scale cloud computing environments,” *MEDC Proj. Rep.*, vol. 22, no. 6, pp. 433–659, 2009.

[76] . T. M., “an Optimum Service Broker Policy for Selecting Data Center in Cloudanalyst,” *Int. J. Res. Eng. Technol.*, vol. 05, no. 09, pp. 76–84, 2016, doi:

10.15623/ijret.2016.0509011.

[77] "Amazon Web Services (AWS) - Cloud Computing Services." https://aws.amazon.com/ (accessed Jun. 24, 2020).

[78] "Cloud Computing Services | Microsoft Azure." https://azure.microsoft.com/en-gb/ (accessed Jun. 24, 2020).

[79] "Internet Speed Test: 3G, 4G, LTE, and Wifi — Who Wins?" https://www.bandwidthplace.com/internet-speed-test-3g-4g-lte-and-wifi-who-wins-article/ (accessed Jul. 02, 2020).

[80] K. Kumar, J. Liu, Y. H. Lu, and B. Bhargava, "A survey of computation offloading for mobile systems," *Mob. Networks Appl.*, 2013, doi: 10.1007/s11036-012-0368-0.

[81] "Android Studio and SDK tools | Android Developers." https://developer.android.com/studio (accessed Jun. 24, 2020).

[82] "PowerTutor." http://ziyang.eecs.umich.edu/projects/powertutor/ (accessed Jun. 24, 2020).

[83] "Cisco GCI Projects Cloud Traffic to Nearly Quadruple | The Network." https://newsroom.cisco.com/press-release-content?articleId=1804748 (accessed Jun. 24, 2020).

[84] "App Tune-up Kit 1.3 Download Android APK | Aptoide." https://app-tune-up-kit.en.aptoide.com/app (accessed Jun. 24, 2020).

[85] "Google Play Services: Using the Nearby Connections API." https://code.tutsplus.com/tutorials/google-play-services-using-the-nearby-connections-api--cms-24534 (accessed Jun. 24, 2020).

[86] "Google Play | Android Developers." https://developer.android.com/distribute/best-practices/engage/nearby-interactions (accessed Jun. 24, 2020).

[87] "Timeout: The story of N-Queens' time complexity - Jon Mohon - Medium." https://medium.com/@jmohon1986/timeout-the-story-of-n-queens-time-

complexity-c80636d92f8b (accessed Jun. 24, 2020).

[88] "Facial landmarks with dlib, OpenCV, and Python - PyImageSearch." https://www.pyimagesearch.com/2017/04/03/facial-landmarks-dlib-opencv-python/ (accessed Jun. 24, 2020).

[89] "GitHub - RonakKosamia/NdroidNDK-Fibonacci-count: NDK library sample to get fibonacci count." https://github.com/RonakKosamia/NdroidNDK-Fibonacci-count (accessed Jun. 24, 2020).

[90] "Oracle VM VirtualBox." https://www.virtualbox.org/ (accessed Jun. 24, 2020).

[91] "Measure app performance with Android Profiler | Android Developers." https://developer.android.com/studio/profile/android-profiler (accessed Jun. 24, 2020).

[92] "What is the POST Method? | POST Method Definition | API Glossary." https://rapidapi.com/blog/api-glossary/post/ (accessed Jun. 24, 2020).

[93] A. Mishra, "Amazon Rekognition," *Mach. Learn. AWS Cloud*, pp. 421–444, 2019, doi: 10.1002/9781119556749.ch18.

[94] "WampServer, la plate-forme de développement Web sous Windows - Apache, MySQL, PHP." https://www.wampserver.com/en/ (accessed Jun. 24, 2020).

[95] "Amazon Linux 2." https://aws.amazon.com/amazon-linux-2/ (accessed Jul. 03, 2020).

[96] S. Goudarzi, M. H. Anisi, A. H. Abdullah, J. Lloret, S. A. Soleymani, and W. H. Hassan, "A hybrid intelligent model for network selection in the industrial Internet of Things," *Appl. Soft Comput. J.*, 2019, doi: 10.1016/j.asoc.2018.10.030.

[97] J. Luzuriaga, J. C. Cano, C. Calafate, and P. Manzoni, "Evaluating Computation Offloading Trade-offs in Mobile Cloud Computing : A Sample Application," no. c, pp. 138–143, 2013.

[98] X. Wang, X. Chen, W. Wu, N. An, and L. Wang, "Cooperative Application

Execution in Mobile Cloud Computing: A Stackelberg Game Approach," *IEEE Commun. Lett.*, vol. 20, no. 5, pp. 946–949, 2016, doi: 10.1109/LCOMM.2015.2506580.

[99] "Experitest: Mobile App & Cross-Browser Testing End-to-End." https://experitest.com/ (accessed Apr. 23, 2020).

[100] "Introduction to Deep Learning - Towards AI—Multidisciplinary Science Journal - Medium." https://medium.com/towards-artificial-intelligence/deep-learning-series-chapter-1-introduction-to-deep-learning-d790feb974e2 (accessed Jun. 25, 2020).

[101] M. Kesarwani, V. Arya, B. Mukhoty, and S. Mehta, "Model extraction in MLAAS paradigm," in *ACM International Conference Proceeding Series*, 2018, doi: 10.1145/3274694.3274740.

[102] A. Gholamy, V. Kreinovich, and O. Kosheleva, "Why 70 / 30 or 80 / 20 Relation Between Training and Testing Sets : A Pedagogical Explanation," pp. 1–6, 2018.

[103] "Deepnets: Behind The Scenes | The Official Blog of BigML.com." https://blog.bigml.com/2017/10/04/deepnets-behind-the-scenes/ (accessed Jun. 25, 2020).

[104] "A Gentle Introduction to the Rectified Linear Unit (ReLU)." https://machinelearningmastery.com/rectified-linear-activation-function-for-deep-learning-neural-networks/ (accessed Jun. 25, 2020).

[105] "The Internet of Everything (IoE) | OpenMind." https://www.bbvaopenmind.com/en/technology/digital-world/the-internet-of-everything-ioe/ (accessed Jun. 30, 2020).

[106] "How to achieve 1-meter accuracy in Android : GPS World." https://www.gpsworld.com/how-to-achieve-1-meter-accuracy-in-android/ (accessed Jun. 30, 2020).

[107] K. Arulkumaran, M. P. Deisenroth, M. Brundage, and A. A. Bharath, "Deep reinforcement learning: A brief survey," *IEEE Signal Processing Magazine*.

2017, doi: 10.1109/MSP.2017.2743240.

[108] "What is wearable tech? Everything you need to know explained." https://www.wareable.com/wearable-tech/what-is-wearable-tech-753 (accessed Jun. 30, 2020).