AN OPTIMAL APPLICATION PARTITIONING AND COMPUTATIONAL OFFLOADING FRAMEWORK FOR MOBILE CLOUD COMPUTING

A THESIS SUBMITTED TO
THE GRADUATE SCHOOL OF INFORMATICS
OF
THE MIDDLE EAST TECHNICAL UNIVERSITY

BY

MAHİR KAYA

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE
OF DOCTOR OF PHILOSOPHY
IN
THE DEPARTMENT OF INFORMATION SYSTEMS

JANUARY 2016

AN OPTIMAL APPLICATION PARTITIONING AND COMPUTATIONAL OFFLOADING FRAMEWORK FOR MOBILE CLOUD COMPUTING

Submitted by **MAHİR KAYA** in partial fulfillment of the requirements for the degree of **Philosophy of Doctorate in Information Systems, Middle East Technical University** by,

Prof. Dr. Nazife Baykal
Director, Informatics Institute ________________________________

Prof. Dr. Yasemin Yardımcı Çetin
Head of Department, Information Systems ________________________________

Assoc. Prof. Dr. Altan Koçyiğit
Supervisor, Information Systems, METU ________________________________

Examining Committee Members:

Assoc. Prof. Dr. Alptekin Temizel
Modeling and Simulation, METU ________________________________

Assoc. Prof. Dr. Altan Koçyiğit
Information Systems, METU ________________________________

Assist. Prof. Dr. P. Erhan Eren
Information Systems, METU ________________________________

Assist. Prof. Dr. Sadık Eşmelioğlu
Computer Engineering, Çankaya University ________________________________

Assist. Prof. Dr. Ömer Özgür Tanrıöver
Computer Engineering, Ankara University ________________________________

Date: 13.01.2016

I hereby declare that all information in this document has been obtained and presented in accordance with academic rules and ethical conduct. I also declare that, as required by these rules and conduct, I have fully cited and referenced all material and results that are not original to this work.

Name and Surname : **Mahir Kaya**

Signature : ____________________

ABSTRACT

AN OPTIMAL APPLICATION PARTITIONING AND COMPUTATIONAL OFFLOADING FRAMEWORK FOR MOBILE CLOUD COMPUTING

Kaya, Mahir
PhD, Department of Information Systems
Supervisor: Assoc. Prof. Dr. Altan Koçyiğit

January 2016, 111 pages

The use of mobile applications is increasing every day and they offer more functionality on mobile devices. However, these devices are inferior to server computers in terms of memory and processor capacity. Furthermore, rapid depletion of mobile devices' energy resources is still a major problem. Performance and energy shortcomings of mobile devices can be improved by using surrogate or cloud computing technologies. In this thesis, an offloading framework is proposed to improve the performance and efficiency of mobile applications. The framework seamlessly handles offloading and provides distribution transparency via the Inversion of Control mechanism. In particular, computation intensive components of an application are run on a remote server. It is possible to migrate different combinations of components to remote servers. Indeed, offloading some combinations of components are productive and others are counterproductive. Experimental results show that offloading the optimal combination of components to remote servers reduces the execution time and energy consumption of mobile devices. Hence, a call graph model is proposed to decide on the components to be offloaded. Offloading decisions are made by finding the best partitioning in the graph. The graph model has been validated by extensive experiments.

Keywords: Code offloading, distribution transparency, application graph partitioning, mobile cloud computing.

ÖZ

MOBİL BULUT BİLİŞİM İÇİN EN İYİ UYGULAMA BÖLME VE HESAPLAMA AGIRLIKLI TAŞIMA ÇERÇEVESİ

Kaya, Mahir
Doktora, Bilişim Sistemleri Bölümü
Tez Yöneticisi: Doç. Dr. Altan Koçyiğit

Ocak 2016, 111 sayfa

Mobil uygulamaların kullanımı her geçen gün artmakta ve bu uygulamalar mobil cihazlarda daha fazla işlevsellik sunmaktadır. Bununla birlikte, bu cihazlar, bellek ve işlemci kapasitesi açısından sunucu bilgisayarlardan daha düşükler. Ayrıca, mobil cihazların enerji kaynaklarının hızla tükenmesi hala önemli bir sorundur. Mobil cihazların performans ve enerji eksiklikleri yerel sunucular veya bulut bilişim teknolojileri kullanılarak iyileştirilebilir. Bu tezde, mobil uygulamaların performansını ve verimliliğini artırmak için bir kod taşıma çerçevesi önerilmektedir. Bu çerçeve kod taşımayı kesintisiz bir şekilde ele almakta ve kontrol mekanizmasının çerçeve yazılıma verilmesi yoluyla dağıtım şeffaflığı sağlamaktadır. Özellikle, bir uygulamanın hesaplama yoğunluklu bileşenleri uzak bir sunucuda çalıştırılmaktadır. Uzak sunuculara uygulama bileşenlerinin farklı kombinasyonlarını göndermek mümkündür. Gerçekten, bazı bileşenlerin kombinasyonlarının sunucuya taşınması kazançlı iken diğerleri için kazançlı olmamaktadır. Deneysel sonuçlar, bileşenlerin optimum kombinasyonun uzak sunuculara taşınmasının işlem süresini kısaltığını ve mobil cihazların enerji tüketimini azalttığını göstermektedir. Bu nedenle, taşınacak bileşenlere karar vermek için bir çağrım çizge modeli önerilmiştir. Taşıma kararları çizgedeki en iyi bölümleme bulunarak yapılmaktadır. Çizge modeli kapsamlı deneyler ile doğrulanmıştır.

Anahtar Kelimeler: Kod taşıma, dağıtım saydamlığı, uygulama çizgesini bölme, mobil bulut bilişim.

dedicated to my wife and son

ACKNOWLEDGEMENTS

First and foremost, I would like to express my sincere thanks to my supervisor Dr. Altan Koçyiğit for his guidance, support and patience throughout this study. I am very grateful for his inspiring ideas, he always provides insightful discussions about the research.

I thank the thesis monitoring committee members Dr. P. Erhan Eren and Dr. Alptekin Temizel for their guidance and feedback throughout this study. I would also like to thank the examining committee members Dr. Sadık Eşmelioğlu and Dr. Ömer Özgür Tanrıöver for their valuable comments and suggestions.

I also wish to thank my colleagues at the institute for an excellent working atmosphere. Special thanks to Serhat Peker, Kerem Kayabay, Ali Mert Ertuğrul, Ahmet Coşkunçay, Ahmet Faruk Acar, Nurcan Alkış, Bilge Sürün and Okan Bilge Özdemir for their personal and scholarly interactions, and Sibel Gülnar and Sibel Ergin for her support in administrative procedures.

Finally, I would like to thank my family for their love, and support. I would like to express my deepest gratitude to my beloved wife Yasemin for her love, encouragement and support, and for not losing her patience throughout this study. The birth of our son Yaman was a great experience that we lived in this period.

TABLE OF CONTENTS

ABSTRACT iv
ÖZ v
DEDICATION vi
ACKNOWLEDGEMENTS vii
TABLE OF CONTENTS viii
LIST OF TABLES xi
LIST OF FIGURES xii
LIST OF LISTING xiv
LIST OF ABBREVIATIONS xv
CHAPTERS
1. INTRODUCTION 1
1.1 Motivation 2
1.2 Objectives and Scope 3
1.3 Research Questions 4
1.4 Thesis Structure 6
2. BACKGROUND AND RELATED WORK 7
2.1 Application Offloading Approaches 7
2.1.1 Offloading Virtual Machine of Mobile Devices 7
2.1.2 Offloading Application Components 8
2.2 Related Work 9
2.3 Proxy-Based Method 14
2.3.1 Transparency of Proxy-based method 15
2.3.2 Proxy Inheritance 15
2.3.3 Proxy Instantiation 17
2.3.4 Field Access 17
2.3.5 Static Members and Private Methods 17
2.3.6 Dynamic Proxies 18
2.3.7 Inversion of Control 19
2.3.8 Java RMI 19

2.3.9 OSGi 20
2.3.10 Android IDL 21
2.3.11 DCOM and CORBA 22
2.3.12 Bytecode Instrumentation 23
2.4 Graph Partitioning 23
2.4.1 Multilevel Algorithm 24
2.4.2 (k+1) Algorithm 26
2.4.3 Min-Cut Algorithm 26
2.4.4 Runtime vs Compile Time Partitioning 27
2.5 Mobile Cloud Computing 27
2.5.1 Cloud Computing 28
2.5.2 Cloudlet Approach 31
2.6 Discovery of Local Machines and Services 32
2.6.1 Jini Service Discovery 32
2.6.2 UPnP and DPWS 32
2.6.3 DNS-SD 32
2.7 Security 33
2.7.1 Secure Sockets Layer (SSL) 33
2.7.2 Open Authorization (OAuth) 33
3. CODE OFFLOADING 35
3.1 Offloading Approach 35
3.2 Offloading Programming Model 35
3.2.1 Proxy and Object Creation 36
3.2.2 Method Call 37
3.2.3 Callback Mechanism 39
3.2.4 Processing Requests 40
3.3 Discussion on Programming Model 42
4. OFFLOADING DECISION MODEL 43
4.1 Offloading Decision Making 43
4.1.1 Defining the weights of vertices and edges 45
4.1.2 Verification of the Graph Model 46
4.1.3 Graph Construction Algorithm 49
4.2 Decision heuristic for offloading classes 50
5. OFFLOADING FRAMEWORK ARCHITECTURE 53

5.1 High Level Architecture of the Offloading Framework 53
5.2 Flowchart of a mobile application development with the offloading framework .. 55
5.2.1 Determining the network cost function coefficient on-the-fly 58
5.3 Fault Tolerance Mechanism of the Framework 59
5.4 Comparison with Other Framework Approaches 60
5.5 Extensibility of the Framework 61
5.6 Sample Application 62
5.7 Using Mobile GPU for General-Purpose Computing 65
6. EXPERIMENTAL EVALUATION 67
6.1 Experiment 1 67
6.1.1 Execution time results 68
6.1.2 Energy consumption results 71
6.2 Experiment 2 (Synthetic Applications) 72
6.2.1 Synthetic Application 1 72
6.2.2 Synthetic Application 2 75
6.3 Speed up and network cost functions using history-based profiles 79
6.4 Experiment 3 82
6.5 Experiment 4 85
6.5.1 Results on the execution time (response time) 88
6.5.2 Power consumption results 91
6.6 Performance Comparison of Frameworks 91
6.7 Discussion on Roaming, SSL connection, Failure and Load on Server 92
7. CONCLUSION 93
7.1 Summary 93
7.2 Contributions 95
7.3 Limitations and Future Work 95
REFERENCES 97
APPENDICIES 105
APPENDIX A: 105
APPENDIX B: 106
CIRCULUM VITAE 109

LIST OF TABLES

Table 1 Comparison of offloading models ... 9
Table 2 Evaluation of offloading models ... 11
Table 3 Comparison of the well-known cloud providers [68] ... 30
Table 4 Key differences: Cloudlet vs. cloud [15] ... 31
Table 5 Offloading gain calculation ... 44
Table 6 Graph model verification (two classes) ... 46
Table 7 Graph model verification on a callback ... 47
Table 8 Graph model verification (three classes) ... 48
Table 9 Comparison of the frameworks ... 61
Table 10 The OCR precision values ... 71
Table 11 Delay for each method of Synthetic Application 1 ... 72
Table 12 Application call graph of Synthetic Application 1 and offloading results 1 ... 73
Table 13 Application call graph of Synthetic Application 1 and offloading results 2 ... 74
Table 14 Application call graph of Synthetic Application 1 and offloading results 3 ... 74
Table 15 Application call graph of Synthetic Application 1 and offloading results 4 ... 75
Table 16 Delay for each method of Synthetic Application 2 ... 76
Table 17 Application call graph of Synthetic Application 2 and offloading results 1 ... 77
Table 18 Application call graph of Synthetic Application 2 and offloading results 2 ... 78
Table 19 Comparison of Offloading gains for a LAN server ... 83
Table 20 Comparison of Offloading gains for the cloud ... 83
Table 21 Offloading cases for the OR application ... 86
Table 22 Graph partitioning results ... 87
Table 23 Offloading gains for LAN server ... 88
Table 24 Offloading gains for Cloud Server ... 89

LIST OF FIGURES

Figure 1 (a) Single machine computation, (b) VM migration [10] 8
Figure 2 Application partitioning taxonomy [36] 12
Figure 3 A class diagram of proxy 15
Figure 4 An object diagram of proxy [46] 15
Figure 5 The extending approach for a proxied class [32] 16
Figure 6 The extending approach for a non-proxied class [32] 16
Figure 7 Dynamic proxy dispatch 18
Figure 8 (a) Local invocation, (b) Remote method invocation (RMI) [3] 20
Figure 9 Android IDL [54] 21
Figure 10 Android service binder [54] 22
Figure 11 CORBA architecture [9] 23
Figure 12 Different ways to coarsen a graph [20] 25
Figure 13 A graph with edge weights [58] 27
Figure 14 The graph after the first minimum cut phase [58] 27
Figure 15 Mobile cloud computing structure [36] 28
Figure 16 Abstract protocol flow [79] 34
Figure 17 An overview of the offloading programming model [1] 36
Figure 18 The class diagram of a sample application 36
Figure 19 The sequence diagram of object creation using a proxy on the smartphone 37
Figure 20 The sequence diagram of a proxy method call 38
Figure 21 The sequence diagram of a proxy method call with object parameter 38
Figure 22 The sequence diagram of a callback 40
Figure 23 Application call graphs 43
Figure 24 Energy model 45
Figure 25 Calculating execution times of the vertices and edges 45
Figure 26 Graph representation of the execution times 46
Figure 27 Graph representation of the energy model 46
Figure 28 An overview of the offloading framework [2] 54
Figure 29 Flowchart of a mobile application development 55
Figure 30 The activity diagram of the runtime behavior of the mobile application 57
Figure 31 Transmission time 58
Figure 32 Logging and recovery mechanism of the framework 60
Figure 33 The class diagram of the OR application 62
Figure 34 The sequence diagram of the OR application 1 63
Figure 35 The sequence diagram of the OR application 2 64
Figure 36 An OCR application 68
Figure 37 The OCR image set taken by the smartphone camera 68
Figure 38 The OCR control image set taken by an image-processing tool 69
Figure 39 The OCR execution time for 400x800 resolution images 69
Figure 40 The offloading execution time for 480x800 resolution images 69
Figure 41 The OCR execution time for 1232x2048 resolution images 69
Figure 42 The offloading execution time for 1232x2048 resolution images 69

Figure 43 The OCR execution time for different network connection (400x800 pixels) 70
Figure 44 The OCR execution time for different network connections (1232x2048 pixels). 70
Figure 45 The OCR execution time for 1232x2048 resolution control images 70
Figure 46 The offloading execution time for 1232x2048 resolution control images............ 70
Figure 47 The OCR power consumption 71
Figure 48 The offloading power consumption... 71
Figure 49 The OCR power consumption for different network connections........................ 71
Figure 50 The class diagram of Synthetic Application 1 72
Figure 51 The sequence diagram of Synthetic Application 1 73
Figure 52 The class diagram of Synthetic Application 2 75
Figure 53 The sequence diagram of Synthetic Application 2 76
Figure 54 The regression analysis of the processing time in the smartphone........................ 80
Figure 55 The regression analysis of the processing time in the LAN server 80
Figure 56 The regression analysis of the speed up function in the LAN server 80
Figure 57 The regression analysis of the processing time in the cloud server....................... 80
Figure 58 The regression analysis of the speed up function in the cloud server.................... 80
Figure 59 The regression analysis of network cost (Wi-Fi connection) in the LAN server .. 80
Figure 60 The regression analysis of the network cost (Wi-Fi connection) in the cloud server 81
Figure 61 The regression analysis of the network cost (3G connection) in the cloud server 81
Figure 62 An image filter application 82
Figure 63 Comparison of execution times on local, LAN server and cloud.......................... 82
Figure 64 Scatter Plot of the gain of the model and the measured gain for LAN server 84
Figure 65 Scatter Plot of the gain of the model and the measured gain for the cloud 84
Figure 66 A graph representation of the OR application 87
Figure 67 The screenshots of the OR application using our offloading framework 87
Figure 68 The offloading cases of the OR application 88
Figure 69 Scatter Plot of the gain of the model and the measured gain for LAN server 89
Figure 70 Scatter Plot of the gain of the model and the measured gain for the cloud 90
Figure 71 Execution time of the OR 90
Figure 72 Execution time of offloading 90
Figure 73 The energy consumption of the OR application cases.. 91
Figure 74 Scatter Plot of the gain of the model and the measured gain for LAN server 91
Figure 75 Comparison of the frameworks 92

LIST OF LISTING

Listing 1 InvocationHandler interface 18
Listing 2 A dynamic proxy factory that wraps a class 18
Listing 3 (a) Concrete implementation, (b) IoC constructor injection 19
Listing 4 The pseudo code of the offloading factory create-method 37
Listing 5 The pseudo code of the proxy method handler 39
Listing 6 The pseudo code of the offloading factory processing the request message 41
Listing 7 (a) Raw and (b) filtered method call stack 49
Listing 8 Application Call Graph Construction Algorithm 50
Listing 9 FM heuristic for the graph partition 51
Listing 10 KL based partitioning heuristic 51
Listing 11 Heuristic solution extension point of the framework 62
Listing 12 A code snippet from the OR application 1 63
Listing 13 A code snippet from the OR application 2 64

LIST OF ABBREVIATIONS

AIDL	:	Android Interface Definition Language
AMI	:	Amazon Machine Image
AOP	:	Aspect Oriented Programming
API	:	Application Programming Interface
AR	:	Augmented Reality
ASM	:	A SMifier
AspectJ	:	Aspect Java
BCEL	:	Byte Code Engineering Library
BIC	:	Bytecode Instructions Count
CA	:	Certificate Authority
CLR	:	Common Language Runtime
CORBA	:	Common Object Request Broker Architectures
CPU	:	Central Processing Unit
DaaS	:	Data Storage as a Service
DCOM	:	Distributed Component Object Model
DHCP	:	Dynamic Host Configuration Protocol
DNS	:	Domain Name System
DNS SRV	:	Domain Name System Service
DNS TXT	:	Domain Name System Text
DNS-SD	:	Domain Name System -Service Discovery
DPWS	:	Devices Profile for Web Services
EC2	:	Elastic Compute Cloud
EJB	:	Enterprise JavaBean
EWMA	:	Exponentially Weighted Moving Average
FM	:	Fiduccia and Mattheyses
FT-CORBA	:	Fault Tolerant- Common Object Request Broker Architectures
GGGP	:	Greedy Graph Growing Partitioning
GGP	:	Graph Growing Partitioning
GPS	:	Global Position System
GUI	:	Graphical User Interface
HELVM	:	Heavy Edge Light Vertex Matching
HEM	:	Heavy Edge Matching
HTTP	:	Hypertext Transfer Protocol
IaaS	:	Infrastructure as a Service
ICT	:	Information and Communication Technologies
IDE	:	Integrated Development Environment
IDL	:	Interface Definition Language

ILP	:	Integer Linear Programming
IoC	:	Inversion of Control
IPC	:	Inter Process Communication
ITU	:	International Telecommunication Union
J2SE	:	Java 2 Platform, Standard Edition
JVM	:	Java Virtual Machine
KB	:	Kilobyte
KL	:	Kernighan and Lin
LAN	:	Local Area Network
LEM	:	Light Edge matching
MAUI	:	Mobile Assistance Using Infrastructure
MCCF	:	Mobile Cloud Computing Frameworks
MP	:	Megapixel
ms	:	Millisecond
NP-Hard	:	Non-deterministic Polynomial-time Hard
OASIS	:	Organization for the Advancement of Structured Information Standards
OAuth	:	Open Authorization
OCR	:	Optical Character Recognition
OMG	:	Object Management Group
OR	:	Object Recognition
ORB	:	Object Request Broker
OS	:	Operating System
OSGi	:	Open Services Gateway initiative
PaaS	:	Platform as a Service
PDA	:	Personal Digital Assistant
PKCS	:	Public-Key Cryptography Standards
RGB	:	Red Green Blue
RM	:	Random Matching
RMI	:	Remote Method Invocation
R-OSGi	:	Remote Services for OSGi
S3	:	Simple Storage Service
SaaS	:	Software as a Service
SOAP	:	Simple Object Access Protocol
SQL	:	Structured Query Language
SSL	:	Secure Socket Layer
TCP	:	Transmission Control Protocol
TCP/IP	:	Transmission Control Protocol/Internet Protocol
TCP/UDP	:	Transmission Control Protocol/User Datagram Protocol
TSL	:	Transport Layer Security
UPnP	:	Universal Plug and Play
URL	:	Uniform Resource Locator

VLSI	:	Very Large Scale Integration
VM	:	Virtual Machine
VMM	:	Virtual Machine Manager
WAN	:	Wide Area Network
WLAN	:	Wireless Local Area Network
XML	:	Extensible Markup Language

CHAPTER 1

INTRODUCTION

Parallel to the developments in information technologies, the use of mobile devices (such as smartphones, tablets) has considerably increased in the past decade. With the improvements in mobile communication technologies (such as Wi-Fi and 3G), users can now easily and instantly access information from a variety of sources. According to the development reports on the Information and Communication Technologies (ICT) of the International Telecommunication Union (ITU), subscription to mobile and fixed broadband is increasing all around the world [1]. In addition, the ITU [1] estimated that the rate of global internet use would reach 40.4% by the end of 2014. The rapidly developing technologies offer many benefits to users in terms of time and cost effectiveness. More recently, smartphones are widely used not only for relatively simple applications such as displaying and sending e-mails, capturing, displaying and sending photos, and instant messaging, but also for more complex and computation intensive applications.

As mobile applications such as image processing, object recognition, augmented reality applications and mobile games are gaining exponential growth, the need for more powerful mobile devices becomes main concern for mobile software developers. On the other hand, user satisfaction requirements such as thickness and weight constrain the capabilities of these devices in terms of processing power and battery. In addition, running these applications solely on smartphones can be impractical with respect to battery lifetime and responsiveness. During the last decade, in order to overcome the resource limitation of mobile devices, mobile software developers used two major approaches to relieve the constrained mobile devices. First is to optimize complex algorithms that are memory and computation intensive to be implemented in the constrained mobile devices. The second approach is to use client-server architecture based on delegation where the computation intensive components are run on a resourceful server, then at runtime, mobile devices request the services provided by the remote server. In the first approach, the software developer concentrates on optimizing the complex algorithms instead of focusing on the business logic of the applications. The second approach also requires designing a communication mechanism between clients and the server, in which low-level network issues and errors in network communication should be handled. Moreover, creating fixed remote services is feasible only if the computational resource configurations such as energy consumption and network characteristic will not change at runtime.

Offloading is an invaluable contribution since it is utilized in mobile computing environments in order to enhance the capability of resource-constrained mobile devices by migrating the components of applications such as classes, objects, services or methods to resourceful servers that are nearby machines (called surrogates) or the virtual machines of the cloud [2]. Mobile devices generally use two offloading mechanisms to benefit from a nearby server or cloud computing infrastructure. In the first method, a virtual machine (VM) of the smartphone is entirely moved to the remote server, re-started, resource

intensive tasks are performed and the VM is brought back to the mobile device. In this method, not only the network cost is too expensive, but also problems occur during calculations that require smartphone' resources such as sensors. The second method is the application partitioning mechanism. This method can be grouped under three sub-headings; the proxy-based methods like Remote Method Invocation (RMI) [3], preparing computation intensive parts as a service using the Interface Definition Language (IDL) [4], and the OSGi service-based method [5]. Partitioning an application and sending the components to be offloaded to remote servers incur less overhead, but require various degrees of program restructuring; and in both cases, problems can arise when an application runs processes that are dependent on the resources of a smartphone.

Recent studies based on services (application components) that require the use of IDL and an OSGi middleware [6], [7] implement the computation intensive parts as services and migrate these services to the remote server. However, these services should be independent of the resources of smartphones such as sensors and cameras. Another problem is that even if services run locally, the application communicates with these services using Inter-Process Communication (IPC) via the network stack, which is time-consuming and thus limits the goal of the computation offloading in terms of increasing the overall performance and reducing the energy consumption. Marshalling (serialization) arguments of the services also create argument inconsistency if the remote operation modifies the arguments passed. Microsoft's DCOM [8], [9] and OMG's CORBA [8], [9] that are well-known architectures for distributed application development, also suffer from such argument inconsistencies. Hence, it is assumed that arguments are passed by value or they are immutable.

Although significant research has been conducted on the mobile cloud computing systems [2], [6], [7], [10]–[14], there are still several challenges to be addressed, as stated above, concerning the design and implementation of a widely adopted framework and the selection of components to be offloaded in current smartphone applications. The limited bandwidth in wireless networks as well as high and changing network latencies in a WAN environment also need to be considered [15]. Therefore, an adaptive, seamless offloading strategy should be implemented without resulting in any extra overhead for the smartphone applications.

1.1 Motivation

There are many offloading solutions in the literature, which allow migrating computation intensive components of an application to remote servers. However, most of them are based on offloading specific components that do not depend on local components such as sensors, camera, Global Position System (GPS) of the mobile device; in other words, they do not support callbacks. Apart from the component granularity level of the existing frameworks for offloading, sending application state such as method parameters, especially large objects, to the remote server leads to extra network overhead as well as argument inconsistency. Therefore, a framework which integrates remote resources as part of mobile devices by overcoming these limitations is required in order to enable them to become dominant powerful computing devices

Offloading usually becomes counterproductive if components which incur higher communication cost than processing cost savings are offloaded to remote servers. An application consists of several components some of which are dependent on each other either highly or loosely. In distributed computing, partitioning an application is a major problem in terms of detecting profitable partitions for remote execution. Most of the existing studies leave marking components to be offloaded to software developers. Hence, an effective model which dynamically presents application behavior at runtime as well as determining optimal partition by which both overall performance increases and energy consumption decreases for mobile applications enhances mobile computing development. In both cases where mobile devices are enhanced help mobile devices industry to satisfy the highest user expectations.

1.2 Objectives and Scope

In this section, the scope and main directions of this study are presented. This thesis focuses on offloading objects of an application through the application partitioning which is based on a call graph based model. Designing an offloading framework to bridge the gap between mobile devices and resourceful servers is a complex process. The major properties of such a framework are determined and solutions are provided. These properties are listed as follows:

- A transparent object offloading technique that supports callback functionality is provided.
- The offloading technique takes into account execution times (or energy consumption) and network bandwidth that are collected and dynamically updated.
- An application partitioning mechanism based on a call graph is modeled
- An optimal partition containing computation intensive classes is migrated to a remote server at runtime.
- Resource-rich local servers and services providing processing capability are discovered and maintained.
- Security related issues such as Secure Socket Layer (SSL) and authentication mechanism are taken into account.

The offloading technique presented in this thesis is based on the Inversion of Control (IoC) [16], [17] mechanism. This technique seamlessly synchronizes resource access on both the smartphone and the surrogate/cloud side of an application and eliminates the limitations of the existing offloading approaches [18], [19]. First, the software modules that are not suitable for offloading; such as software modules providing the Graphical User Interface (GUI), utilizing local resources such as sensors and network components are ignored. Other components are candidates to be offloaded to resourceful servers. The selection of the components to be offloaded depends on certain factors; such as the amount of computation and data required for the method call, and bandwidth of the available wireless network.

The proposed framework delegates an object creation to a factory in which when an object creation is requested, it checks the object type to determine whether it is eligible to be offloaded. If the framework decides that an object type (class) is the component to be offloaded, it returns a proxy to access the services of the object created in the server. In the mobile device, methods invoked from the proxy are delegated to the object residing in the surrogate/cloud. When the execution is completed, the returned values are sent back to the mobile device. There may be cases, where the object that is run on a server requires the callback functionality or the resources in the mobile device, which need to be handled with extra caution. In such cases, reverse proxies are provided in the server to access objects residing in the local device.

The dynamic proxies also allow profiling the method calls and collecting measurement data; such as the execution time of the called methods, methods' parameter types, the size of the parameter values. Once an object creation is requested from the framework, a proxy is locally executed to collect such information. During offloading, remote execution and network times are also collected and updated for each class. This information which is collected at object level accumulated at the class level. A profiling algorithm is developed to collect the cumulative statistics for each class including (method execution times and number of method calls). Since estimating the method execution time statically from the source code [12] is not easy and lacks run time behavior of applications, the desired metrics have been collected at run time. In order to provide an input to the optimal partitioning algorithm, a call graph is constructed. For each method call, the method call stack is monitored and the call graph is updated. In this graph, the vertices stands for classes and the edges present the class dependency in terms of method calls.

In this study, in addition to the offloading technique, a framework is presented to discuss the offloading decision-making. To this end, a novel graph model is proposed to collect the

profiling information and then to decide on the parts of the application to be offloaded and to be executed on the remote server. Constructing the call graph, the offloading decision making problem is converted to the graph partitioning (min-cut) problem [20]. The graph partitioning approaches are suitable to make such offloading decisions [2], [21]. However, finding an optimal solution for the graph partition is NP-Hard [20], [22], therefore in this study, a well-known graph partitioning heuristic; Fiduccia and Mattheyses (FM) heuristic [20], [23], was implemented to determine the minimum edge-cut that is the best offloading decision. The proposed framework uses a modular approach, and therefore is suitable for the inclusion of a new heuristic algorithm. Different combinations of application classes were offloaded to identify the cases where offloading can be counterproductive. This is important in terms of finding an optimal solution for offloading to decrease the network cost involved. The quality of the offloading decision-making is measured with respect to achieving a specific goal which is improvement of the application performance and decrease on energy consumption. As a result, partitioning an application effectively at runtime is one of the research questions of this thesis.

Network discovery allows services and computers to learn the availability of networked devices and consume their services. The goal of service discovery protocol is to decrease or eliminate explicit administration [24]. In this study, DNS-SD [25] protocol has been implemented to be aware of the availability of local machines. The network resources are classified according to the DNS-SD naming structure. DNS SRV and DNS TXT records are used to facilitate this protocol [26]. A DNS query in a specific format is sent to a local network in order to find available services.

Security related issues are also handled in the framework, Secure Socket Layer (SSL) socket connection is utilized [27], [28]. SSL regulates authentication of the client and server. The communication between parts is encrypted to provide security. The software developer can indicate the SSL connection at the object creation phase for specific objects. The software developer can also make all communication with SSL; however, this situation would result in costly offloading which is not worth for large data transmission. In addition, Twitter OAuth mechanism [29] was used to authenticate users to access server resources. In OAuth, the client initiates an access request to protected resources that are managed by a resource owner and hosted by a resource server. During authorization, the credentials of the resource owner are not used. The client acquires an access token that indicates various attributes such as scope and life time. The token is provided by an authorization server [30]. Once the authorization is achieved, the client can access all offloading servers.

In order to support development transparency which is to minimize the burden on the software developer in terms of application partitioning and proxy injection that connects objects residing in different virtual machines, Android platform on the mobile device and Java (J2SE) which is a cross-platform technology on the server side are the main focus of this thesis to implement proposed solutions. To evaluate the offloading framework, several experiments were carried out and the performance of offloading was assessed on real time and synthetic applications. An Object Recognition (OR) [31], an Optical Character Recognition (OCR) and an image filtering applications were used since they show the effectiveness of the proposed framework. Using these applications, the offloading technique and the decision model were implemented. The graph model allowed determining the best offloading decision and the results were validated conducting several experiments on real time applications. In addition, to handle all these issues automatically, the burden and the error prone development tasks on software developers are relieved and this framework would also reduce software development efforts by providing generic solutions.

1.3 Research Questions

Although there have been many researches for several years, offloading or cyber foraging models still need to be improved to provide a holistic computation offloading approach including remote server discovery, gathering context information, application partitioning,

remote execution, error handling and security to be practically usable in real life. In this study, the main focus is the computation offloading models for surrogate/cloud architectures and the following research questions have been pointed out:

1. Can offloading improve performance of the mobile devices in terms of responsiveness (execution time) and energy consumption?
   Answering this question, the existing approaches and applications were reviewed while the smartphone capabilities were considered. As expected, smartphones have certain limitations when compared to their desktop equivalents in terms of CPU processing and memory capacity. The battery lifetime is also another important constraint. To see whether offloading improves the responsiveness of an application or not, several experiments were conducted on an OCR application. The application was run on the smartphone locally and some computation intensive parts of the same application were migrated to the remote server. In the experiments, which are presented in chapter 6, the responsiveness of the smartphone applications was considerably improved by offloading. In addition, the overall reduction in execution times led to significant decrease in the energy consumption of the smartphones.

2. In what ways application partitioning can be achieved efficiently at runtime? Is it possible to perform partitioning without the need of developers' intervention?
   These questions are answered through existing application partitioning approaches. The existing frameworks heavily depend on explicitly defining computation intensive parts of the application by annotations as discussed in Section 2.2. Thus the software developer needs to define methods and services to be offloaded. Application partitioning is handled through whether sending the annotated methods and services or not. To decide the annotated methods to be offloaded is made via integer linear programming that is too costly for each method. Therefore, some studies handled this costly operation on the server side by communicating to the server to send the updated state after which they gathered the results. On the other hand, an overall optimal partitioning algorithm considering dependency to other application components is the main focus of the thesis. After metrics collection, an application call graph was constructed. Since the optimal partitioning solution for the call graph is costly, an optimal partitioning heuristic is proposed to find the best offloading containing classes. Consequently, the decision model finds productive classes to be offloaded at runtime.

3. Is it possible to have a streamlined offloading architecture? And how can the distribution transparency of application partitioning be improved?
   To answer these questions, existing offloading mechanisms proposed by Verbelen et al. [6] and Kemp et al. [7] were investigated. In addition, current object oriented approaches [3], [32], [33] were reviewed in terms of injecting proxies transparently, which does not break the existing application code. A number of limitations in these approaches were identified and addressed. After this analysis, the proposed offloading technique was designed, as presented in Chapter 3. The solution preserves the polymorphic behavior of methods and invocations. The framework collects metrics through method calls of the proxy objects. Collected metrics are also updated during each method call. A service discovery mechanism and security issues are also handled. Furthermore, other approaches require some processes such as a pre-compile phase, change of smartphone' virtual machine and third party middleware such as Apache Flex for OSGi-based service, which increases the framework dependency to the specific mobile application platforms. On the other hand, since our framework only uses dynamic proxies and reflection in object oriented languages, it does not depend on underlying smartphone OS and other middleware.

1.4 Thesis Structure

This thesis is organized as follows:

Chapter 2 presents a detailed background of the study. Application offloading approaches are investigated. The taxonomy of the application offloading and granularity of application components for offloading are discussed. Application partitioning heavily depends on proxy-based method. Thus, transparency of proxy injection to an existing application without a developer intervention and change of existing application structure is presented. The graph partitioning algorithms and mobile cloud computing approaches are detailed. Lastly, service discovery protocols and security issues are reviewed. This section also involves an overview of the related work on offloading methods and offloading decisions. A detailed comparison of the offloading methods is presented. The literature review helps to draw specific issues and limitations in previous works, which contains distribution transparency, application partitioning and execution of application components on the remote server.

Chapter 3 presents the offloading programming model. How issues related with distribution transparency are handled is explained. The pseudo-code of the offloading technique is presented. How the injection of proxies to the application and method call through proxies are explained. In addition, the reverse proxy mechanism for callback is explained.

Chapter 4 presents the application partitioning model which is based on a call graph model. The proposed graph model and offloading decision algorithm are also described. The metrics collections and graph construction algorithm are presented.

Chapter 5 presents the offloading framework. The offloading framework structure both on the mobile device and the server side is presented. The components of the framework are explained. The runtime behaviour of a mobile application using the framework is presented. A prototype implementation of the framework through an example application is discussed.

Chapter 6 presents the performance evaluation of the framework. The evaluation goal of each example application is addressed. The examples show whether offloading is productive or not. It has been shown that finding the optimal graph partition containing classes to send the remote server for execution can significantly increase the overall application performance. The most important part in this section is the verification of the graph model. The result of the graph model and data measured in the experiments are compared.

Finally, Chapter 7 concludes the present work discussing the limitations of our study, contributions of the thesis and possible future work.

CHAPTER 2

BACKGROUND AND RELATED WORK

2.1 Application Offloading Approaches

In this section, main approaches related to application offloading are presented. There are two main approaches for offloading. First approach is migrating a whole VM of a smartphone to a server and the latter is to only send specific components of an application to the server for utilizing extensive resources of remote machines such as clouds.

2.1.1 Offloading Virtual Machine of Mobile Devices

Recent developments in virtual machine technology and cloud computing architecture bring a demand for utilizing the remote computers by allocating the clone of the smartphone VM. Three approaches for VM migration were proposed. In first approach, the VM of the smartphone, which is already executed on a smartphone, is suspended and then migrated to the remote server with its memory and disk state, then this VM is launched in a remote server. After execution of the requested task, it returns the VM to the smartphone [15]. The second approach is to load a base VM of the smartphone in the remote server at the initiation phase. During the application execution, a dynamic VM synthesis which is a small VM overlay is sent to be integrated with the base VM in the remote server. Although, the second approach relatively decreases the state transfer, the migrated data is also very large when considering constrained smartphones and network bandwidth. Synchronization of both sides can also be very complex. Satyanarayanan [15] proposed this solution for one hop away cloudlet architectures which have high bandwidth WiFi connection. On the other hand, the third approach [10], [34] is based on sending an application-level VM which is an abstract computing machine. The bytecodes of the method area are executed through this VM. The method area contains stack information and heap objects. This method requires considerable change of the smartphone VM structure and also sends the stack information and all heap objects at runtime, which is very costly in terms of network overhead. Figure 1 presents the VM migration architecture for the third approach.

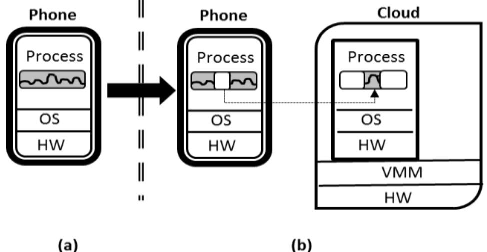


Figure 1 (a) Single machine computation, (b) VM migration [10]

2.1.2 Offloading Application Components

An adaptation of mobile applications via component (classes and services) mobility has presented a solution to address the limitations of constrained mobile devices. By offloading components of an application to the cloud, computational power and battery lifetime of mobile devices can be enhanced. Component granularity is a major issue affecting the benefits of offloading. The partition granularity level is usually handled at object, class and method level. Therefore, it is important to analyze the advantages and disadvantages of granularity level of the offloading decision. The computation intensive components are executed on the remote server and the results return to the smartphone. Component offloading is generally done through proxy-based methods as explained in Section 2.3.

2.1.2.1 Object offloading

An object can be offloaded by investigating the bytecode of an application and it is converted to the component to be offloaded. The analysis of the application can also be handled at runtime. There are algorithms [35] to determine the dependencies between the objects of a program. The objects of applications are monitored and statistics are collected to compute a usage graph. The edges between objects represent communication and nodes represent objects' sizes or memory usage. This graph can be partitioned in terms of computation related objects. Moreover, Static analysis has a certain shortcoming compared to execution analysis, since we cannot exactly depict the objects' usage by analyzing the source code.

2.1.2.2 Class offloading

Class offloading is based on tracing all objects of a class to partition and send all of them to a surrogate server. Class offloading analysis can also be done either via implementation or at runtime by computing the consumption graph. Since such a graph partitioning problem is NP-complete, a graph partitioning heuristic can be applied to choose the classes to offload with respect to their interactions. This method requires a special virtual machine implementation to monitor all objects of desired class.

2.1.2.3 Method offloading

Method offloading is not as much complex as other two methods. The call graph can be computed by static analysis and communications such as method arguments' return values between nodes and number of times the method is executed is collected by execution analysis. Since the quantity of methods in an application is more than the quantity of classes, application analyzers can spend more time in method offloading. In method offloading, all methods are sent to a surrogate with their dependent methods. In addition, the method of an application should be wrapped to be sent and executed on a remote server. On the other

hand, software developers may want to send only specific methods that do not depend on any other method or smartphone resource.

2.2 Related Work

The studies related to offloading in the literature have been investigated via systematic review approach. This study has especially focused on the researches between 2010 and 2015. Scopus, Web of Science and IEEE Xplore were employed in searches. Only the studies focusing on “Computation Offloading”, “Mobile Computing”, “Mobile Cloud Computing” and “Application Partitioning” were analyzed. Table 1 presents the comparison of offloading studies with respect to a taxonomy containing methods, partitioning time, dependent platform, offloading component granularity and offloading decision. Table 2 shows the evaluation of each study and applications that are implemented to validate the studies. As a result, image processing and augmented reality applications are mostly implemented for evaluation.

Table 1 Comparison of offloading models

App/Author	Method	Partitioning	Partitioning Algorithm	Platform	Granularity	Offloading Decision
Verbelen et al., (2010, 2011)	OSGi	Static, compile time	Convert annotated classes to the service	Java	Bundle	ILP + runtime
Verbelen et al., (2012)	OSGi	Static partitioning at compile time	Convert annotated methods to service	Android	Bundle	History based profile
Kemp et al., (2011) Cuckoo	AIDL	Static partitioning at compile time	Convert methods to AIDL services	Android	Service	-
Kovachev and Klamma (2012) MACS	AIDL	Static partitioning at compile time	Convert methods to AIDL services	Android	Service	ILP + runtime
Cuervo et al. (2010) MAUI	Microsoft. Net byte code instrumen tation	Dynamic	Method annotation as remotable	Microsoft .Net	Method	Profiling metrics, ILP
Kristensen and Bouvin (2010, 2012) Scavenger	RPC and decorator pattern	Dynamic	Method annotation	Python	Method	Task and peer centric profiling (history based profiling)
Chen et al., (2012)	AIDL	Static partitioning at compile time	Create wrapper for remote services	Android	Service	Execution time, energy consumption and remaining battery metrics
Zhang et al., (2012)	Byte code instrumen tation	Dynamic	Call graph model, depth-first search algorithm	Android	Method	-
Yang et al. (2008)	Byte code instrumen tation	Dynamic	(k+1) graph partition algorithm	Java	Class	Resource metrics; memory, bandwidth

Table 1 (Cont.)

Kosta et al. (2012) ThinkAir	Proxy-based	Dynamic	Method annotation as remotable	Android	Method	Profiling metrics at runtime
Chun et al. (2011) CloneCloud	Java byte code instrumen tation	Static compile time	Static and dynamic profiling are used to detect expensive methods	Android	Firstly send mobile VM, then send thread includes methods to be offloaded	Profiling metrics,
Zhang et al. (2010)	weblet	Dynamic	-	-	weblet	-
Hung et al. (2011)	Android state transfer	Dynamic	VM migration	Android	Android component state migration	-
Gu et al. (2004)	Java RPC	Dynamic	Min-Cut graph partitioning and Fuzzy offloading decision engine are implemented	Java	Class	Online profiling
Geoffray et al. (2006)	Java RPC	Dynamic	Byte code instrumentatio n	Java	Method	Online profiling
Rim et al. (2006)	Java RPC	Static, reads configuration file	Byte code instrumentatio n	Java	Method	Configuratio n file
Giurgiu et al. (2009) AlfredO	OSGi + proxy	Dynamic	All and K-step partitioning algorithm	Java	Module called bundles (functional)	Offline profiling
Rellermeyer et al. (2008)	OSGi + proxy	Dynamic service call	Service based	Java	Module called bundles	-
Han et al. (2008)	Java RPC	Dynamic	Max-flow Min-cut algorithm	Java	Functional requirements	Online profiling
Abebe and Ryan (2012)	Byte code injection	Dynamic	Distributed local application graph	Android	Class	Online profiling
Ou et al. (2007)	Byte code instrumen tation	Dynamic	K+1 partitioning algorithm	Java	Class	Online profiling
Gao et al. (2012)	Task flow partition	Dynamic	Graph based task partition and resource allocation algorithm	Simulation	Task	-
Flore et al (2015)	AIDL	Static partitioning	Create wrapper for remote services	Android	Service	Fuzzy Logic

Table 2 Evaluation of offloading models

App/Author	Evaluation / Validation	Applications
Verbelen et al., (2012, 2011)	Compare execution time on mobile device and offloading, This is the first application that uses the OSGi method.	Augmented reality shopping assistant application
Verbelen et al., (2012)	Comparing execution time on mobile device, LAN offloading and Cloud offloading	Chess and Photo Editor Application
Kemp et al., (2011) Cuckoo	Comparing execution time on mobile device and server.	eyeDentify: a multimedia content analysis application.
Kovachev and Klamma (2012) MACS	Comparing execution time and energy consumption on mobile device and offloading	N-Queens problem and process a video file, detect faces from video file, cluster them and provide video point in terms of faces
Zhang et al. (2012)	Compare execution time of their method that is Call link with ILP method.	Face recognition and Natural language processing
Yang et al. (2008)	Compare execution time on mobile device and offloading using one and two surrogate.	Combined OCR and translation app
Kosta et al. (2012) ThinkAir	Comparing execution time and energy consumption in case of mobile device, WiFi local, WiFi WAN.	N-queens problem, a face detection program, a virus scanning app and an image merging app.
Chen et al. (2011) CloneCloud	Comparing execution time and energy consumption with respect to offloading file size or image size, WiFi and 3G connection.	Virus scanning and Image search
Cuervo et al. (2010) MAUI	Execution time, energy consumption and cpu usage are compared in terms of WiFi and 3G.	Face recognition, interactive video game.
Kristensen and Bouvin(2012) Scavenger	Comparing running time on mobile device and offloading.	Image manipulation app
Zhang et al. (2010)	Comparing running time, energy and memory on mobile device and offloading	Image processing and AR app.
Hung et al. (2011)	Comparing running time on mobile device and on server in terms of file or image size.	P2P file exchange and face recognition
Gu et al. (2004)	Execution time is compared in case of memory constraints when offloading or not	Java İmage Editor, Graphical molecular editor, Java text editor.
Geoffray et al. (2006)	Execution time is compared according to offloading or not	an interactive bookstore implemented with servlets
Rim et al. (2006)	Execution time is compared according to downloaded application's byte code size.	Complex scientific applications
Giurgiu et al. (2009) AlfredO	Execution time is compared according to whether to offload or not	3D home design from images
Rellermeyer et al. (2008)	Execution time is compared according to whether to offload or not	AlfredOShop shopping application
Han et al. (2008)	Whole application runs on the mobile device, except mobile device dependent components are moved to the server and their partitioning algorithm are implemented and energy consumption is compared	3D game application
Abebe and Ryan (2012)	Performance and battery consumption are compared.	A java based n-body simulator using the Barnes-Hut algorithm, a Hospital System Simulator, and NASA World Wind demo application.
Ou et al. (2007)	Execution time saving is compared.	PiCalculator and MP4GenPlayer application.
Gao et al. (2012)	Energy and execution time savings are compared.	Simulation
Flore et al (2015)	Energy and execution time savings are compared.	NQueens problem

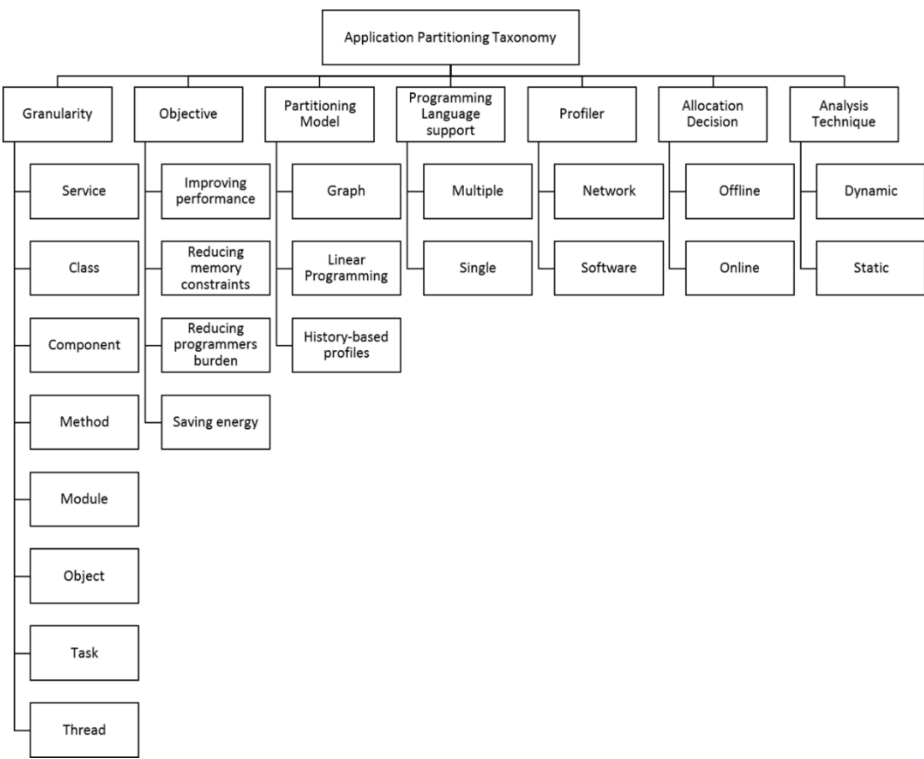


Figure 2 Application partitioning taxonomy [36]

Figure 2 presents an application partitioning taxonomy. The application components for partitioning can be different granularity such as class, method, service and task. The aim of offloading components of an application can be varied. The performance improvement and energy saving can be aim of the offloading. For partitioning an application, the parameters related with software and network can be collected and used. An allocation decision, which is related with whether to send the object or not, can be made offline (at compile time) or online (at runtime). The analysis of the application can also be made at runtime or at compile time.

For application partitioning, graph based approaches can be used to partition an application graph into a number of disjoint subsets that contain the list of classes to be offloaded to resourceful servers. The cumulative weight of edges whose incident vertices are located in different virtual machines is called the edge-cut of the partition. The main goal of application partitioning is to minimize the edge-cut that determines the network cost. Although determining the optimal partitioning to distribute the components of applications is an NP-Hard problem, there are various classical heuristics offering solutions [20]–[23]; however, these heuristics need to be adapted to mobile environment in terms of costs and balance constraints. In addition, Ou et al. [2] proposed a different multi-level graph partitioning heuristic for mobile applications coarsening the graph based on the heavy edge-light vertex algorithm. The coarsening phase continues until the subsets/partitions that are suitable for component distribution are achieved. The algorithm randomly chooses vertices and merges them with their neighbors that have low vertex cost and high edge weight. The edge weight of this heuristic is the frequency of the method call among different classes. The vertex weight represents memory and CPU processing cost. The runtime complexity of this heuristic is $O(|V|^3)$ (|V| is the number of vertices) and the computation process for finding suitable partitions to offload is also expensive. Abebe and Ryan [37] implemented the same heuristic but maintained the distributed partitions on the cloud side to decrease the memory-related costs.

Flores et al. [14] discussed the importance of the decision-making process in partitioning a mobile application. To increase the responsiveness of an application or decrease the energy consumption, components that with low or no dependency on the locally executed part should be determined with caution at runtime to be offloaded. If not, code offloading usually becomes a more costly process than locally running the application. Flores et al. proposed a method level code offloading using the java reflection library for pure functions that do not require callback to mobile device resources, so this approach decreases distribution transparency.

Chun et al. [10] and Satyanarayanan et al. [15] developed a VM migration mechanism fully migrating the processes of a VM running on a mobile device to a remote server. The CloneCloud determines costly methods using Integer Linear Programming (ILP) at build time (off-line) and stores the pre-defined execution sets in a database to be checked at run time. In this method, first a mobile VM, then a thread containing offloadable methods are sent to a remote server. However, the VM cloning of application-layers and sending this clone with the application state to a remote server incur high communication costs. Therefore, sending only the components with high computation costs to the remote server can be more effective. Using this method, Android Dalvik VM [38] is also modified to implement dynamic profiling and state synchronization, but changing Dalvik VM is applicable only through cloud-side implementation in real scenarios.

In the Mobile Assistance Using Infrastructure (MAUI) system proposed by Cuervo et al. [11], remotable code parts are identified and marked by programmers. This method aims to reduce the energy consumption of smartphones through fine-grain (method level granularity) code offloading. A programmer needs to determine the offloadable methods in the development phase. Methods are implemented in Windows Mobile OS environment using the features of the Common Language Runtime (CLR), which allows using meta-data information with the methods. Compiled files contain this information to be used during offloading. In addition, other method information including parameters and static variables are serialized and then sent to the remote server in an XML file. However, sending the state of each method to a remote server may result in inconsistencies if the variables and states change during execution. In addition, in their research, Cuervo et al. did not illustrate how MAUI can handle the callback functionality. Sending an XML state file and marshalling/demarshalling this file is a time and resource consuming task.

Verbelen et al. [6], [39], [40] developed a framework based on the OSGi modular application development for Android. Since the Android platform does not inherently support the OSGi system, the researchers used a middleware (i.e. Apache Felix) to run the OSGi modules. In this method, Android application components were converted to suitable OSGi service interfaces to execute an Android application based on OSGi bundles. Therefore, the researchers introduced a plugin for Eclipse IDE for the development of an Android application. This plugin enables the developer to annotate possible offloadable methods. To publish the annotated methods as OSGi services at build time, the plugin produces suitable OSGi bundles. Moreover, the OSGi middleware needs to be embedded in all Android applications to run OSGi modules at runtime.

Kemp et al. [7] developed the Cuckoo framework using the Android service mechanism, which encapsulates a computation intensive task. This framework offloads Android services to a resourceful server and facilitates static partitioning at compile time. The programmer implements computation intensive tasks via the Android IDL as a local service. Android services use Inter-Process Communication (IPC) channel for remote procedure call. Then, the Cuckoo framework implements the same interface for a remote service. This study involves dummy method implementations that can be run on the remote server. While real methods can be similar to those used in the local service implementation, they can also be changed to implement a different algorithm.

Kovachev and Klamma [41] further improved the Cuckoo framework adding an offloading decision mechanism that implements ILP. It is too time consuming to provide a solution for each method call using ILP. Energy and execution time estimations for Android services are only based on the code size. Chen et al. [13] also utilized the Android IDL interface to offload resource intensive code parts at compile time using static partitioning.

Zhang et al. [42] offloaded code parts using the bytecode instrumentation at runtime in a mobile application. However, this way of code offloading requires de-compiling the signed mobile application and then adding the required functionality to the application, which may lead to inconsistency problems. In order to define code parts that can potentially be offloaded, Kristensen and Bouvin [12] used a method annotation. The authors proposed a cyber-foraging system based on Python methods using a history-based profiling that stores the parameter size and value of each method to estimate the remote execution time at runtime. However, this system does not support the callback functionality. Chen et al. [43] offloaded only specific computation intensive methods to a remote server using Aspect Oriented Programming (AOP). Since the development environments of mobile devices do not officially support AOP, the researchers modified the Android build process. This study was only based on the pure functions.

Ling et al. [44] presents architecture to achieve real-time mobile Augmented Reality (AR) application. Although mobile devices are necessarily resource–poor relative to the static client-server hardware, they are more suitable for outdoor AR applications. The main problem is integration of cloud resources and mobile devices. For collision detection and collision response in AR applications it should process the virtual objects with physical objects in real time. Mobile devices can achieve low-latency and high bandwidth wireless access by using a nearby resource-rich cloudlet instead of the remote public cloud. In the proposed architecture, smartphones, PDAs and other mobile devices can transmit image data via wireless network. Private AR cloudlets handle the data instead of the distance public cloud on internet. The architecture consists of three cloudlets which are pre-processing cloudlets, storage cloudlets and post-processing cloudlets and they are working respectively. The experiment results show that the proposed AR cloudlets are faster than traditional approach.

Although there has been a lot of research on offloading, most of the proposed methods encounter problems if they need to call back resources from the smartphone or shared memory. Therefore, the unity of the serial execution of programs cannot be guaranteed. This issue can even be completely ignored and therefore distribution transparency may not be completely achieved. Most of the studies only consider the pure functions (not depending on other components) for offloading.

2.3 Proxy-Based Method

A proxy class is used as a placeholder for an original application class to intercept communication between the client/calling class and server/called class [45]. Proxies can be injected to an existing application to support application adaptation through object mobility. It enhances access to the target class to provide extra capabilities before or after execution of the target class methods. By creating proxies for a software entity, frameworks have been also allowed to collect adaptation metrics such as execution times, CPU and memory usage.

There should be structural and semantic compatibility between a proxy class and the real class. Structural compatibility means the equivalence of the type compatibility that the proxy class and real class should implement the same proxy interface, method signature and method modifiers. On the other hand, semantic compatibility guarantees that the application behavior is not changed after proxy injection, in particular the polymorphic behavior of methods are preserved. Since the functionality of application middleware is commonly

supported via proxies such as distribution transparency, the proposed offloading framework proxies have important roles to provide an adaptive and seamless mobile application development.

2.3.1 Transparency of Proxy-based method

The transparency of developed frameworks and middleware is dependent on the transparency of proxies. The more transparency the proxy has, the less the software developer intervention to the application and the more capable software applications are achieved. The object-level proxy approaches [32], [46], [47] are discussed in Section 2.3.2.

2.3.2 Proxy Inheritance

A proxy is a class providing an interface to the real subject object that has the same interface as the proxy. The client code calls the method of the proxy; however, the proxy has the real subject object and forwards all calls to it [48]. The current studies have focused on many characteristics of proxy inheritance, which are the attributes of the parent classes and the external classes (system or library classes) extended by original classes. In order to access the methods of the proxy and the real subject objects in the same way, both classes (proxy and real subject) should implement an interface in the object oriented languages.

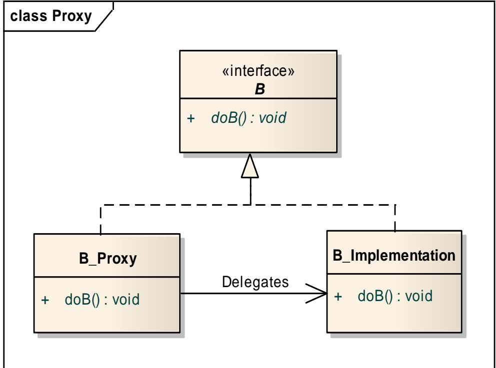


Figure 3 A class diagram of proxy

Figure 3 presents the class diagram of the proxy pattern [46]. An interface (B) declares public method of class B and class B_Implementation is transformed from the source code of the original class. Class B_Proxy delegates its behavior to the implementation class. This approach has some limitations due to not covering the inheritance relation of the original classes. This is so since object oriented languages such as Java and C# do not allow an interface to extend a class.

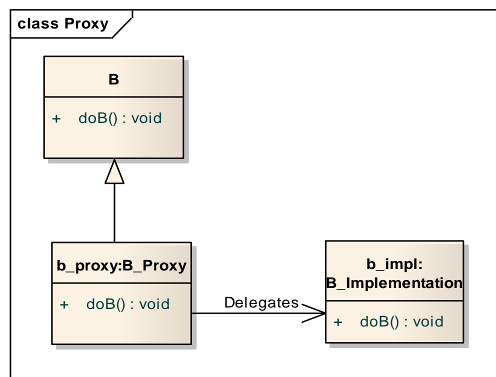


Figure 4 An object diagram of proxy [46]

An approach proposed by Eugster [46] is a flexible class structure (Figure 4). Since class B_Proxy is extended from the original class, it has an ability to execute all methods of the proxied class. However, the aim of the proxy is not only to execute the methods of the original class but also it generally forwards the execution to a separate class instance such as a remote object. In addition, if a proxy class only executes the methods of the inherited class, it will become a decorator design pattern [45]. Another case is to extend a proxy class and modify the original class to forward the method execution to another object. JavaParty [33] and J-Orchestra [49] implemented this approach but when an extending relation is used for external classes, some dependency problem might be caused in the external classes because of the modification.

In order to overcome these limitations; class members, member identifiers, super class and interfaces have been taken into consideration to reach a general solution. Gani and Ryan [32] proposed an approach to improve the proxy class structure of JavaParty and J-Orchestra frameworks in terms of extending a proxied class and a non-proxied class. The main characteristic of this approach is that a proxy class only extends classes that are extended from the original classes so that the class type compatibility can be assured. The drawback of this approach is that it does not support dynamic change of the proxy class and proxied class. Changing the proxy class and proxied class dynamically without modifying the client code is important for adaptive framework development.

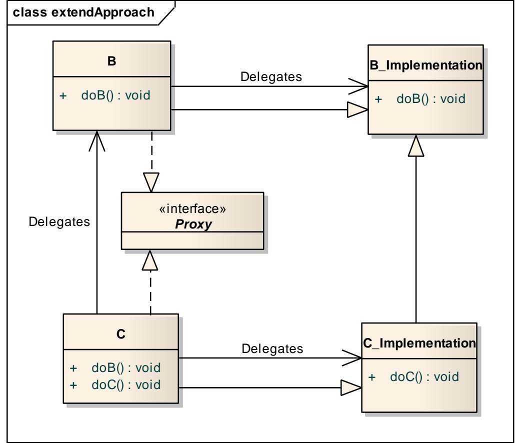


Figure 5 The extending approach for a proxied class [32]

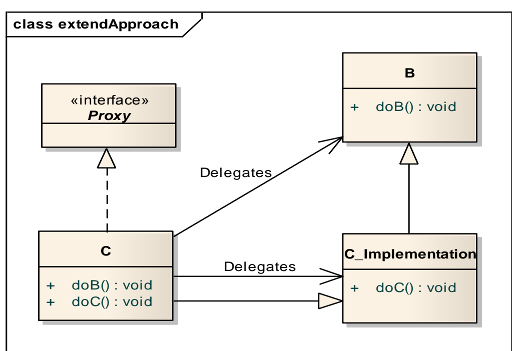


Figure 6 The extending approach for a non-proxied class [32]

In order to provide the dynamic swapping behavior between a proxy class and a proxied class, Ryan and Westhorpe [50] proposed the extending approach. In this approach, a proxy class (class C) also extends the implementation class (class C_implementation) to ensure type compatibility. As class C also extends the parent class which is class B, the type

compatibility among proxy of class C, class C implementation and class B implementation are preserved (Figure 5 and Figure 6). However, if the proxied class C also extends the proxied class B to delegate the method call of the parent proxied class, the proxied class C will have duplicate functionality. Therefore, to prevent this duplication problem the proxied class C has delegation relation with proxied class B rather than inheritance relation. The dynamic swapping between the proxy and proxied object is handled by using inheritance relation. For the proxy of classes which are not implemented an interface, the extending approach was implemented.

2.3.3 Proxy Instantiation

The proxy creation mechanism can explicitly be handled by programmers creating a proxy/stub in the software program. In Java RMI [3] and Dynamic Proxy API [47], the programmer should request proxy creation. In contrast, Enterprise JavaBean (EJB), which is an API to run software components, uses a dependency injection method by providing a setter method to create proxies. The EJB middleware in program initialization phase creates proxies and pass them to the setter method. On the other hand, JavaParty and J-Orchestra frameworks handle the proxy creation in a fully transparent way. When a programmer requests an object creation by using "*new*" keyword in Java program, the frameworks create the corresponding proxy instead of the real subject object. However, these frameworks are not explicitly targeted to constrained mobile devices. In particular, the development environments of mobile devices such as Android platform do not allow implementing these frameworks because Android virtual machine (Dalvik) structure is not same as the JVM.

In the framework presented in this thesis, the proxy creation is handled by the factory method. The programmer has to use the framework's factory to create all objects instead of using the "*new*" keyword. When a software developer requests the object creation from the factory method of the framework, the framework initiates a proxy in a fully transparent way. Instantiation of the original class is automatically handled by the proxy.

2.3.4 Field Access

In object oriented programming languages, the client object communicates with a target object through its fields and methods. The client object calls the desired method of the proxy and then the proxy forwards the requested method call to the original object. However, the field access is not achieved in the same manner since the field of an object allows only standard operations such as read and write. Therefore, the proxy object cannot delegate the field access to the original object. There are solutions to overcome this restriction. First, the programmer is forced to add a setter function to write a field value and a getter function to read a field value. The second solution is that frameworks modify the client code to call the proper field access method rather than directly accessing the field. In the framework presented in this thesis, the first solution was chosen to delegate the field access to the original object. Software developers have to write getter and setter methods of the fields.

2.3.5 Static Members and Private Methods

Static fields and methods are members of classes in object oriented languages. They do not belong to a specific instance of classes or objects so all instances of the classes access the same static fields and methods. These static fields and methods are shared in a single Java VM; therefore, in distributed applications, since each JVM loads separate copy of a class, the sharing mechanism does not work across JVMs. This issue can be handled by providing additional proxy class for static members (J-Orchestra). When a client needs to access static members, the call can be delegated to the proxy of the static members to synchronize the access.

Proxies are generally implemented to communicate with objects separated across the network. The other issue is about accessing private methods. Private methods, which are restricted to the code residing in the same class, cannot be accessed through proxies. To access a private method of the original class is only possible by swapping the private identifier for public. As expected, using public identifiers also change the semantic of the methods. As an example, assuming a super class and a child class is extended from the super class and if the identifier of a private method of the super class is changed to public, the child class can override the method of the super class. Therefore, modifying the identifier only can cause an inconsistency. In addition, private methods do not have polymorphic behavior because of invisibility from the outside of the class.

2.3.6 Dynamic Proxies

A dynamic proxy provided by Java Dynamic Proxy API [47] is a class that forwards method call from the proxy class to the original class at runtime. The dynamic proxy requires a list of interfaces that a proxied class needs to implement and an invocation handler that delegates the request to the original class. This API contains a proxy interface to be implemented by a proxy class to provide type compatibility.

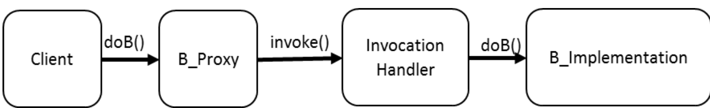


Figure 7 Dynamic proxy dispatch

```
public interface InvocationHandler {
    Object invoke(Object proxy, Method method, Object[] args)
        throws Throwable{  // dispatch to the original method  }
}
```

Listing 1 InvocationHandler interface

```
public class A_ProxyFactory {
    public static <T> getClassProxy(<T> classType) {
        return (classType) Proxy.newProxyInstance
          (classType.getClassLoader(),
                new Class[] { classType },
                new InvocationHandler() {
                    public Object invoke(Object proxy, Method method,
                      Object[] args) throws Throwable {
                       // collect the metrics before the method execution
                        return method.invoke(proxy, args);
                       // after method execution
                    }
                });
    }
}
```

Listing 2 A dynamic proxy factory that wraps a class

Figure 7 presents a dynamic proxy creation and calls the target method. An InvocationHandler interface is implemented by an invocation handler object which is associated to each proxy instance. The proxy method invocations are forwarded through its invocation handler instance’s invoke method. This invoke method takes the proxy instance, a

method object that contains all declared methods and an array of object that contains arguments. The invocation handler invokes the appropriate method and the result of the method will be returned on the method invocation on the proxy instance. This is detailed in Listing 1 and Listing 2 where:

proxy**:** The method calls are forwarded to the original object through an instance of the proxy.

method**:** The methods' signature of original objects is gathered from their interfaces and the proxy instance can implement the same interface to invoke the methods. A method class which contains declared methods is implemented by the proxy interface.

args: An array of objects which is the arguments of the proxied object is given to the invocation handler.

2.3.7 Inversion of Control

Inversion of Control (IoC) is a mechanism which separates an implementation from the application code. It is more specifically known as dependency injection. The implementation of a service can be wired at runtime to the application [16]. The main difference between frameworks and libraries is that frameworks are extendible. Libraries usually provide some specific functionalities and a client calls the methods to do some work. On the contrary, a framework presents some abstract design which allows customizing various behaviors either by sub-classing or plugging-in in specific places. This specific behavior is then called by the framework at runtime.

As a specific example, consider class C needs to work with different implementation of class B. If a concrete class B is provided to class C, the client class is needed to be changed for the different implementation of class B in Listing 3a. On the other hand, the different implementation of class B in Listing 3b can be provided through constructor injection. There are also different type of injections such as setter injection and interface injection.

(a)	(b)

Class C…
 B b;
 Public C() {
   b = new B();
   b.doB();
 }
	Class C…
	 B_Interface b
	 Public C(B_Interface b) {
	   this.b = b;
	   b.doB();
	 }

Listing 3 (a) Concrete implementation, (b) IoC constructor injection

The basic idea behind the IoC is to have a separate object, an assembler, that is responsible to provide the different implementation. In our framework, the object creation is requested from a factory class (an assembler). The factory class can provide either a local object or a proxy depending on the offloading decision.

2.3.8 Java RMI

The java virtual machine allows us to use local objects by providing their references. In Figure 8a, class B (caller) retrieves the local reference of class C (callee) from virtual machine (VM) and then invokes the methods of C. This structure is not suitable to offload any parts of the class C because the local reference of C that is held by B becomes invalid if C is offloaded to another VM. Therefore, in order to call a method of a remote object, it is

needed to use RMI or a special mechanism that implements transparent binding to the remote object references.

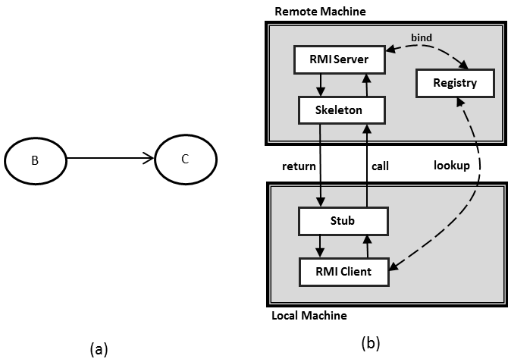


Figure 8 (a) Local invocation, (b) Remote method invocation (RMI) [3]

In Figure 8b, Remote Method Invocation (RMI) mechanism employed in java programs is illustrated. In RMI, the client objects can get the remote reference of the server object to invoke methods of the server object through stub-skeleton classes. The java remote communication service that is a *lookup* service and Android IDL mechanism are responsible to associate an object's reference with the real object which is located in the remote server. For example, in Android platform, an object B retrieves a reference of object C by Service Connection (Android IDL) class then invokes the remote method of C. However, this architecture suffers some performance problems if both classes are in the same local machine. If offloading frameworks decide not to offload class C, the interactions between B and C still go through the time consuming network stack because the local reference is not provided.

2.3.9 OSGi

OSGi, which is based on centralized service oriented architecture, enables java classes to be produced as a service. The other bundles can use this service. A service contains the instance of a class that is implemented, interface of the services and service properties. All published services are saved to a registry and they are monitored and handled through the OSGi framework [51]. Software modules are named as bundles which call each other via services. Since OSGi specification is based on local VM references, it is not specified for distributed application modules. Rellermeyer et al. [51] specifies and implements distributed OSGi called R-OSGi using a remote service discovery and registering remote services to allow consumers to find it. R-OSGi is implemented for managing interaction between bundles located in different devices. If a remote service is called through a bundle, R-OSGi will create a local proxy bundle which is responsible for forwarding the method call of the original bundle. Rellermeyer et al. [52] also proposed a framework that enables distributed application development and supports mobile devices to use modules of the applications based on R-OSGi. The framework, called AlfredO, enables most of the electronic devices in our vicinity to give software as a service instead of pre-installed drivers to use them. The devices which provide some functionalities are announces their capabilities as a service; thus, mobile devices can reach these services at runtime and use the devices.

Giurgiu et al. [53] constructs application resource consumption graph which is built using the OSGi module system. They use offline profiling to determine the resource consumption such as memory and data usage except CPU usage. They report that it is not easy to measure CPU usage for different mobile environments with precision. Verbelen et al. [6], [40] also implement OSGi and proxy-based method to offload the computation-rich tasks to the cloud by means of dynamic decision at runtime.

Since Android development model is different from OSGi modular system, it requires OSGi middleware such as Apache Felix to run OSGi modules. Therefore, in order to implement Android application as OSGi bundles, it is needed to convert Android application components to the proper OSGi service interfaces. Verbelen et al. [6] developed a framework that enables OSGi modular application development for Android. It presents a plugin for Eclipse IDE, where Android application is developed. The classes to be offloaded are annotated by the software developer via the provided plugin. Therefore, the suitable OSGi bundles are produced by converting the annotated classes to OSGi services at the build time. In order to run OSGi modules at runtime, it is also required to embed the OSGi middleware to each android application.

2.3.10 Android IDL

The most important components of an Android application are activities and services. An activity is a user interface (screen) of an application. A service is executed in the background for computation tasks. The android service mechanism separates user interface components from the application logic. Since it is also possible to run services in the background as a separate thread, it does not allow the application logic to intercept user interface components. When a user wants to start an android application, she/he first launches an activity which presents a graphical user interface, and allows binding to running services or initiate a new service. Android services communicate through inter process communication (IPC) mechanism [4], [54]. When an activity is bound to a running service, this service can also be shared by multiple activities. Android IPC mechanism uses a predefined interface by specifying an IDL file and a stub/proxy pair. Android pre-compiler produces the stub/proxy pair as illustrated in Figure 9.

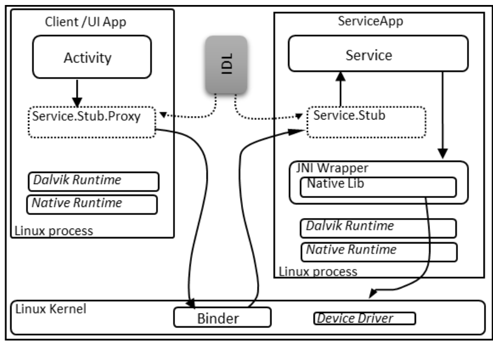


Figure 9 Android IDL [54]

Android interface description language called AIDL should be used to define interfaces in both the consumer and the provider side. The arguments of proxy methods which are used to invoke service methods can be primitive type and Parcelable [4]. The objects that implement Parcelable are serialized to Parcels. These objects are restored from Parcels like Java serialization mechanism that is used to serialize and deserialize objects from byte arrays. Services can also invoke a method on the activity by callback which is supported by Android IPC The Parcelable interface provides a protocol to write and read object from Parcel (write both the class type and its data to the Parcel that is responsible from writing and reading object states).

There are two main steps in Figure 9: Define a remote interface via AIDL, The AIDL compiler then generates a marshalling code via its stub methods and Service and Client-specific methods can be implemented. The Android AIDL build tool extracts a Java interface from each *.aidl files and places it into a *gen* directory. Android AIDL tool also creates a Stub inner class inside android service through an aidl file. Hence, developers first need to create an instance of their own ServiceConnection class in order to use an AIDL-defined service. In the ServiceConnection subclass, onServiceConnected() method, which is called

once activity is bound to the Service to obtain a proxy to the Binder implementation should be implemented[54].

Kemp et al. [7] proposed a framework called Cuckoo which benefits from Android service mechanism that encapsulates a computation intensive task. The framework offloads android services to resourceful server. It enables static partitioning at compile time. The programmer implements computation intensive tasks by an AIDL interface as a local service. For a remote service, the Cuckoo framework produces an implementation of the same interface. This implementation has the dummy methods to be executed on the server. The real methods can be the same with their local service implementation; however, since the developer may want to implement a different algorithm on the remote server, these methods have different implementation.

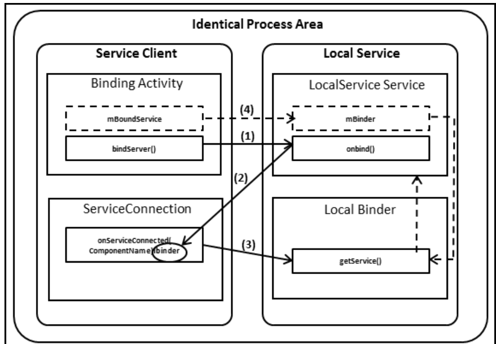


Figure 10 Android service binder [54]

A Service is the component of an application. Services generally performs heavy operations in the background and do not have a direct user interface. They are usually started from other application components and continue their execution in the background while the mobile client switches between application activities.

In Figure 10, an application component (client) calls bindService() to bind to a service. After that, onBind() method of the service is called by the Android system to provide an IBinder that handles interaction with service. To receive the IBinder, the client has to create an instance of ServiceConnection which is passed to bindService() method. When creating a Bound Service, it is necessary to provide an IBinder via an interface. Clients can interact with the Service via the following ways:

- Extending the Binder class: If the service runs in the same process as the client, it is possible extend the Binder class and return an instance from onBind().
- Using a Messenger: Create an interface for the service with a Messenger that allows the client to post commands to the service across processes via Message objects.
- Using Android Interface Definition Language: AIDL handles the works related with decomposing objects into primitives. The operating system then marshal these objects between processes to enable IPC.

2.3.11 DCOM and CORBA

Microsoft's Distributed Component Object Model (DCOM) and OMG's Common Object Request Broker Architectures (CORBA) are two main standard architectures to distribute objects across different machines in object oriented programming. The aim of these frameworks is to support software developer to concentrate on their application logic instead of complex network interactions. Thus, a client uses any object and their methods in the application without considering the location of objects that may reside in the local machine or the remote server. In case of a remote object, the method call will be forwarded to the remote server and this network issue is hidden from a software developer [8], [9]. By

implementing these frameworks, distribution transparency is achieved. Both frameworks are based on the communication type of client–server. A client calls a method of a remote object which encapsulates server services based on the method signature defined in the interfaces. By implementing an interface, an object guarantees to provide all functionalities presented in that interface. An extensive documentation on how to call a method of an object, and how to maintain object references are presented in the literature.

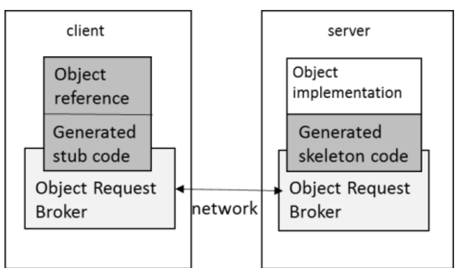


Figure 11 CORBA architecture [9]

Figure 11 presents CORBA architecture including client and server side. The Object Request Broker provides a central bus which is responsible for interacting objects transparently [9]. The stub in client side and the skeleton in the server side are automatically generated in each side to handle network communication on behalf of each object. In addition, IDL is used to define the public interfaces for objects. Since both is based on client-server communication, the callback functionality is not presented for objects.

2.3.12 Bytecode Instrumentation

An application can be analyzed and computation-heavy tasks can be extracted at runtime by using a bytecode instrumentation library. Zhang et al. [42] proposed a framework that achieves bytecode offloading of the desired application by using java instrumentation libraries. After partitioning the classes that are annotated, the framework creates proxy classes that stay on the mobile phone and are responsible for implementing remote function call. The framework also offloads the computation tasks to more than one surrogate. The basic idea of this framework is to transform the bytecode of the application to create proxies for original objects at runtime.

ASM, Javassist and Apache BCEL tools provide inspection, editing and creation of Java binary classes. The inspection aspect mainly copies what is available in Java through the Reflection API; however, if you have an alternative way to access this information, it will be useful when you are actually modifying classes rather than just executing them. This is because the JVM design does not provide any access to the raw class data after it has been loaded into the JVM. In addition, most of the libraries developed for Java platform are not compatible with Android Dalvik VM because of different VM structures.

2.4 Graph Partitioning

Graph based approaches for application partitioning are used to separate into parts that contain components to be offloaded. The graph consists of vertices and edges. The vertices stand for application components. The weight of vertices may consist of memory and CPU usage and execution time of each component. The weight of edges may reflect the data communication between vertices. A dynamic and static profiling is used to collect the application information. The static code profiling is based on inspecting Bytecode Instructions Count (BIC) of applications.

How a mobile application partitioning is achieved is the important subject to develop an adaptive and efficient offloading architecture. Following issues should be addressed:

- Application component classification: distinguish components to be offloaded and not offloaded.
- Application component weighing: according to resource usages such as execution time, assign the weights of each application component.
- Reducing communication overhead: transmission time between the mobile device and a cloud server.
- Decrease the algorithm complexity: algorithms running on the mobile device should be light-weighted.

The computation cost resulted from running components or the application on a mobile device and communication cost resulted from sending data between the mobile device and the server are gathered. The migration cost is taken into account when a component is migrated over wireless network. Then, these two classes of costs are added to the components and connectors; thereafter a general graph model that represents the software structure of an application is set up.

2.4.1 Multilevel Algorithm

A graph (G) can be partitioned through a multilevel algorithm which starts from coarsening a graph to have fewer vertexes. This coarse graph then can be partitioned into smaller graph. Lastly, the final partition is converted back to the original graph via several refinements [55].

Suppose a graph consists of vertices (V) and edges (E):

Coarsening phase: During the coarsening, the graph G is sequentially converted to smaller graphs G_1, G_2… G_n such that $|V|> |V_1|>|V_2|>\ldots>|V_n|$. |V| is the number of the vertices in the graph and $|V_i|$ is the number of the vertices in the subgraph G_i.

Partitioning phase: An initial two part or k part partition P_n of the graph $G_n(V_n, E_n)$ is calculated at the end of the coarsening phase when the coarsest level of the graph is produced.

Uncoarsening phase and refinement phase: The partition P_n of G_n is uncoarsened back to G by handling all the intermediate graphs.

During coarsening phase, A Graph G_{i+1}, which has fewer vertices, is computed from the finer graph G_i by removing incident edges, thereafter the vertices connected by those edges are joined. The weight of the combined vertices is equal to the sum of the weights of the vertices whose edges are removed. In the case where both vertices have an edge to a third vertex, these two edges are combined to one edge by summing the weight of the edges [20]. Thus, a multinode consisting of vertices that have matching is created. A matching of a graph is a set of edges without common vertices. A matching is maximal if any edge in the graph that is not marked as matching has at least one of its endpoints matched. Since the maximal matching is used to coarsen the graph. The number of vertices in the coarsened graph should not be less than half the number of vertices in a level finer graph. However, the size of the maximal matching can be lesser than the |Vi|/2 according to the connection of the edges of G_i. The coarsening algorithm can be stopped if the ratio of the number of vertices from G_i to G_{i+1} might be much smaller than two. The threshold value can be defined for comparing with this ratio.

A maximal matching can be computed by implementing various algorithms such as Random Matching (RM), Heavy Edge Matching (HEM) and Light Edge matching (LEM). RM is a effective method to calculate a maximal matching and decreases the number of coarsening level. However, the main aim of the graph partitioning in our context is to minimize the

edge-cut cost among separate parts. If the heavy hedge is chosen to combine the vertices, the weight of the matching is extracted from the total edge weight of the coarser graph. Figure 12 shows different ways to coarsen a graph. The weight of the vertex of the coarser graph is equal to sum of the sub vertices. In addition, if both vertices of an incident edge points to third vertex, the weight of the edges are summed.

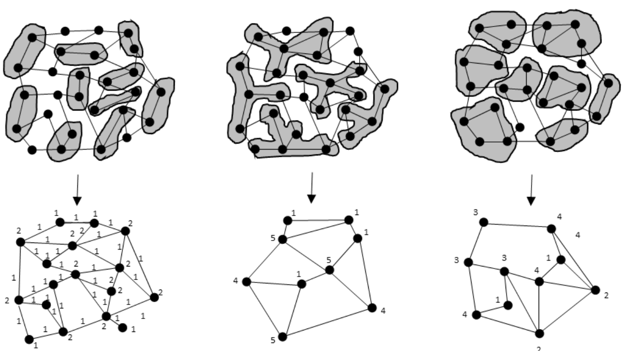


Figure 12 Different ways to coarsen a graph [20]

The second phase of a multilevel algorithm is a partitioning phase of the coarser graph. There are various partitioning algorithms such as spectral bisection and combinatorial methods. The spectral bisection algorithm is a time consuming algorithm; therefore, this study has focused on the combinatorial methods. KL algorithm [22] which is one of them begins with an preliminary partitioned graph. The algorithm tries to find a subset of the vertices that produce a smaller edge-cut cost from each part of the graph in each iteration. If it finds these subsets in the coarser graph, thereafter the swap is executed and this partition is used in the next iteration. The algorithm proceeds by reiterating the whole cycle. If such subsets cannot be found, the algorithm ends. At this point, the partition can be at a local minimum and the KL algorithm does not provide a further improvement. Fiduccia and Mattheyses (FM) [23] algorithm has improved the KL algorithm by reducing the run time by moving only one vertex. The gain of a vertex is defined as the decrease on the edge-cut if that vertex is moved from one partition to the other.

The second initial partitioning algorithm is the Graph Growing Partitioning (GGP) that starts from a random vertex and grow a region around it in a breath-first approach. The algorithms may stop when the vertices in the partition become a half of the coarser graph. The quality of this algorithm depends on the selected initial vertex to start the algorithm. In order to increase the quality, the algorithm may start by selecting ten different vertex and growing around them, then according to the lower edge-cut cost, the partition from this set can be chosen as an initial partition. The third one is Greedy Graph Growing Partitioning (GGGP). In this algorithm, the gain can be calculated in the edge-cut by inserting a vertex into the growing region. The vertices are ordered in increasing order in terms of their gain, then the vertex with the largest decrease in the edge-cut is added to the partition and the gain of frontier vertices are updated.

During uncoarsening phase, the partition of the coarser graph is uncoarsened back to the original graph. As a result of this process the graph becomes finer and it has more possibility to improve the partitioning and decrease the edge-cut. A partition refinement algorithm can be used after uncoarsening to the original graph. The main purpose of a partition refinement algorithm is to pick two subsets of vertices and then swaps those subset of vertices in different subpartitions that leads to the greatest potential edge-cut cost. At this phase, KL refinement algorithm can also be implemented to lead to a better partition within a few iteration. In addition to this, KL refinement algorithm can also be executed for only

boundary vertices in the each partition. The pseudocode of the algorithms are presented in Appendix A.

2.4.2 (k+1) Algorithm

The algorithm partitions an application represented as a call graph into k components to be offloaded and not offloaded so as to distribute partitions to k remote machines. An undirected graph, called dynamic multi-cost graph, presented as G(V, E). V stands for vertex that is application's classes and E presents edges between nodes and their corresponding communication costs. Each vertex has multiple costs such as CPU usage, memory utilization and bandwidth that are presented either as a vector or as a composite vertex weight that is weighted cumulative of these costs. Edges' cost stand for data accesses between nodes or classes. The categories of costs can be computational costs of the application including memory costs, processing costs, bandwidth costs and communication costs that are interaction between components and offloading cost itself.

The clustering of the components is based on finding the most related parts. In these types of partitioning algorithms, first collapse a graph by separating all nodes and then recursively reduce the size of the graph by partitioning the related parts, algorithm needs pre-defined constraints for finding match during coarsening the graph. In this step, selects k heavy weighted edges, then if edges satisfy the predefined constraints, the lower and upper bound for constraints should be defined, and nodes count is less then k, algorithm will finish; on the contrary, the nodes are needed to be merged in terms of heavy edge that means the incident vertices are tightly connected and light vertex that means more vertices will be merged under the predefined constraints in order to coarsen the graph. If a node matches the constraints it is marked in order to prevent to add in more vertices. If the cost constraints are fulfilled by the partitions but an unmatched vertex remains in the graph, the partition is an unsuccessful, and then follows this procedure again to produce the desired partitioned graph. Ou et al. [2] implemented this algorithm to distribute an application. Ou et al. [2] partition a given application into components to be offloaded iteratively. Some other approaches [2], [20] are finding heavy edge or light edge matching. In addition, some heuristics [20], [56] only consider the weights of edges and some [20] both weight of vertices and edges. Yang et al. [57] implements k+1 algorithm in terms of decreasing communication between classes and the memory and CPU request of the classes. The graph is constructed by a profiling phase which is a part of the offloading. Apart from the user interfaces and network communication classes, the graph is cut with respect to weights of the vertices and edges.

2.4.3 Min-Cut Algorithm

A min-cut heuristic was proposed by Stoer and Wagner [58] to dynamically partition a graph G (V, E). The algorithm separates a graph into two partitions so that the sum of the weights of edges separated to different parts is minimum. Figure 13 presents a sample graph with edge weights. The algorithm first selects a vertex that does not change through iterations and this vertex is one partition (the source partition). The other partition (the sink partition), which is a subset of the graph, is iteratively found. The second partition grows by selecting random vertices and adding them to the partition. When a random vertex is selected, it is added to the most tightly connected vertex and the edge weight is updated. The cut of the phase which is the sum of weights of last added edge is calculated and if the phase cut is less than the existing minimum cut, the phase cut is saved as the existing minimum cut. The minimum edge cut pattern will become the result. Figure 14 presents the addition of a random selected vertex to the subset after the first minimum cut phase which is 5.

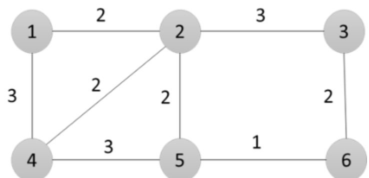


Figure 13 A graph with edge weights [58]

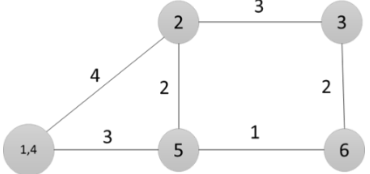


Figure 14 The graph after the first minimum cut phase [58]

Gu et al. [56] proposed an adaptive offloading framework that implements min-cut heuristic to partition an application at runtime. A min-cut [58] heuristic algorithm is adapted for this purpose. The algorithm separates a graph into possible partition plans in line with the edge-weight, thereafter it picks the best partition plan that produces the profitable cut. Since the most of the resource usage are associated to vertex-weights such as execution times and memory consumption rather than edge-weights such the interaction level between components, in the min-cut heuristic of Gu et al. [56], there is a possibility to not find some better partitioning solutions.

2.4.4 Runtime vs Compile Time Partitioning

Application partitioning can be handled either at pre-compile time or at runtime. After application development, software developers can use proper plugins in order to create parts to be offloaded of their applications, which is called as pre-processing. Offloading frameworks' plugins benefit from method annotations, pre-defined classes or services to convert the computation heavy part of the application to the components to be offloaded. These components to be offloaded then can be sent to resourceful servers at runtime. On the other hand, application partitioning can also be handled at runtime according to online profiling and runtime partitioning algorithms; however, this method incurs some extra overhead to the mobile devices. Most of the runtime algorithms use the bytecode instrumentation mechanism to extract the computation intensive partition from the application.

2.5 Mobile Cloud Computing

Bridging the gap between constrained mobile devices and cloud infrastructure, there has been an arising interest to develop the middleware that controls and coordinates communication between the resourceful cloud machines and smartphones. The aim of mobile cloud computing frameworks is to augment constrained mobile devices in terms of CPU, memory and storage capabilities via utility computing vision of computational clouds. There have been a lot of mobile cloud applications because of being assisted by cloud resources. Mobile applications can utilize cloud resources on demand and at different levels (SaaS, IaaS, PaaS). In order to bring cloud resources to the nearby of the mobile devices, there are two main ways that most of the middleware implement, which are offloading and delegation [59].

Most of the current cloud services providing a lot of functionalities to mobile devices work on a delegation model. Mobile devices implement the cloud services which are based on service oriented architecture. Mobile applications gathers RESTful-based [60] services at runtime to send mobile tasks. These pre-defined services aim to fulfill the specific functionality of mobile applications. Offloading provides more flexibility than a delegation model. Mobile application can be partitioned at different granularity levels such as methods, classes and services, and analyzed at either compile time or runtime to determine

computation-intensive components sent to a resourceful server to increase the performance (responsiveness of an application) and decrease the energy consumption.

Currently, cloud providers present Web APIs to develop and deploy web services to be used by mobile applications. A mobile task can be distributed to a cloud service which can be from platform level or infrastructure level and is located on different clouds such as public, private. However, these Web APIs causes several problems such as compiler limitations, additional dependencies and code incompatibility. In addition, Web APIs require a specialized knowledge to develop and deploy each mobile task to specific smartphone OSs platform. Kaya et al. [61] developed RESTful-based services to be consumed by mobile applications. Mashup services combining Google Map API, flickr API to provide location-based social campus application was developed.

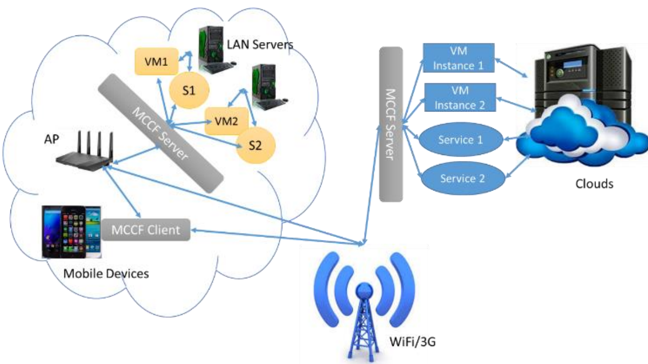


Figure 15 Mobile cloud computing structure [36]

Figure 15 presents the communication structure of Mobile Cloud Computing Frameworks (MCCF). MCCFs are deployed to servers located in LAN and the cloud to provide the computation capability to smartphones.

2.5.1 Cloud Computing

Cloud Computing has recently appear as a key technology for sharing resources. The concept of cloud computing depends on making many computers together to get a super computer in order to deliver computing-on request. This computing concept works like other public services such as electricity and gas [62]. However, cloud computing is not a new concept. Grid computing, utility computing and on-demand computing precede cloud computing by trying to solve the problem of organizing computational power to easily accesses and publicly available resources [63], [64]. According to Malathi [65] Cloud Computing has six key characteristics: “on-demand self-service, broad network access, resource pooling, location independence, measured service, rapid elasticity”.

In the On-demand computing service, without any user intervention cloud resources are provided to clients’ devices when additional resource is required. In addition, in order to benefit the resource pool of cloud high-bandwidth network communication is necessary. These resources are used by various client devices such as mobile phones, laptops and tablets. Resources of cloud are available to be used by many consumers according to the multi-tenancy or virtualization model. Cloud resources are allocated at runtime based on customer demand. Consumptions of the computing resources are automatically monitored and billed to the consumer.

Cluster computing, Grid computing and currently cloud computing aim to deliver computing as a utility vision. Cloud computing has improved this utility computing vision further step by allowing users to reach provided services at anytime and from anywhere. Developing

software as a service to be used by many users instead of running on their individual computers is becoming inevitable development aspects [66].

Cloud computing has achieved its characteristics by virtualization. Virtualization has recently enabled the abstraction of computing researches by means of multiple logical VMs on a physical machine. VMs enables hosting of many operating systems which are independent from each other but actually working on the same machine, which also provides security and privacy. In addition to this, resource allocations such as CPU and memory usage are also varying in terms of the need of the user. According to the changing demand of resources by user, VMs can be dynamically stopped and started.

Cloud Computing differs from Grid Computing in terms of resource utilization and deployment model. In Grid computing, the resources are collaboratively used to construct virtual structures or corporations while cloud is generally conducted by private corporations except some open source organizations.[67]. Grid computing tries to achieve the maximum capacity by dividing a huge task into a lot of independent and no related sub task, and then allow every node to do the jobs.

Scalability and elasticity are important aspects in cloud computing in terms of a hardware view. The cloud computing user is not worry about further plan for provisioning due to the fact that resources is available as though they are unlimited. Cloud computing also eliminates traditional corporations' trade-off whether to satisfy customer needs also in the peak by establish as many servers as required or to sacrifice some profit by establish servers on steady usage. Thus, corporations can allocate small hardware resources at the beginning and if an increase occurs in the demand, the cloud resources can be easily increased [68]. Since the payment of resource usage is based on short term basis such as hour and day, the resources can be released when they are no longer needed. There are many well-known cloud computing providers which are Amazon, Microsoft and Google. Table 3 presents the comparison of the well-known cloud providers.

Amazon Elastic Compute Cloud (EC2) [66], [68], [69] offers a virtual computing environment in which user can run Linux and MS Windows based applications. It is a web service and scales up and down the capacity in the cloud. An Amazon Machine Image (AMI) contains an operating system, application software and other settings related with configurations is available. Multiple virtualized instances can be provisioned by using these AMIs. In addition, it is adjustable by the web interface through web service calls to change the capacity according to requirements. The user can start, monitor and stop instances of either created or selected AMIs after S/he uploaded the desired AMIs to Amazon Simple Storage Service (S3). Pricing is varying according to which service is instantiated if Amazon EC2 is used, the price is depends on the time spent to run the instance; otherwise, Amazon S3 price depends on any data either upload or download transferred.

Google App Engine [70] permits user to build web based applications on the same resizable systems that support Google applications. Developers can use Python programming language to write their applications and once applications are deployed on Google App Engine, all maintenance and scaling up-down works will be handled by Google App Engine. Users can also web applications which is based on Java technologies. After development, these web applications will be available on the infrastructure provided by Google. The data store, Google accounts, URL get, image manipulation and e –mail services have been supported by Application Programming Interfaces (APIs). Users can use web-based administrator control so as to manage and monitor their running web applications. It provides 1 GB of storage and about 5 million page requests each month without any bill. When applications are enabled, it is only billing usage above the free limit.

Since Microsoft Windows Azure [71] offers an unified development, hosting and environment in which user can control their application. User can create, upload, and control web and other applications via Microsoft datacenters. It consists of three parts which are Windows Azure and SQL Azure. Compute and storage services are supplied by Windows Azure. A relational database based on cloud services is supplied by SQL Azure. Windows Azure aims to provide general computing services instead of serving to a specific application. The system do not allow users to control the provided operating system but users can choose the language. Although the network configuration, fault tolerance and scalability are automatically handled by the libraries in the system, the developer should explicitly configure some properties of the application. As a result, Windows Azure works as a middle structure that has whole application framework functioning as Google AppEngine and has virtual machines functioning as Amazon EC2

Table 3 Comparison of the well-known cloud providers [68]

	Amazon Elastic Compute Cloud(EC2)	**Microsoft Windows Azure**	**Google AppEngine**
Computation Model	Infrastructure	Platform	Platform
Service Type	Compute, Storage	Web and other applications	Web applications
Virtualization	Operating System level on a Xen hypervisor	Operating System level on Fabric Controller	Application container
User Access Interface	Administrator control provided through web interface	Microsoft Windows Azure Portal	Administrator control provided through web interface
Programming Framework	Adjustable Linux, Windows Server	Microsoft .NET	Python and Java

2.5.1.1 Service Model

Software as a Service (SaaS): cloud clients or software developers deploy their applications on an environment providing any hosting structure. Thus, application clients can reach these services via internet by different devices such as web browsers, smartphones and tablets. Cloud providers do not allow cloud clients to control the cloud infrastructure. Applications uploaded by various cloud clients is structured as an isolated logical environment. This scenario is called Software as a Service (SaaS). Examples are SalesForce.com, Google Docs and Mail.

Platform as a Service (PaaS): An abstraction level providing the software platform rather than a virtual machine providing infrastructure is offered by cloud providers. An additional resource requirement is transparently fulfilled when services executed on the software platform need extra hardware resources. This is named as Platform as a Service (PaaS). Google Apps Engine and Microsoft Windows Azure are famous examples.

Infrastructure as a Service (IaaS): Cloud providers are allocate isolated hardware infrastructures based on virtualized environment which allocates various computing resources. The underlying hardware resources can be changed at runtime with the help of virtualization. Cloud clients can choose an operating system and then deploy their software

structures which is providing different services. This scenario is named the Infrastructure as a Service (IaaS). Amazon EC2 is a famous example.

Data Storage as a Service (DaaS): A virtualized storage is supplied by cloud providers as a distinct cloud service which is named as data storage service. DaaS can be considered as a specialized version of IaaS. Google BigTable, Amazon S3, Apache HBase and so forth are well-known examples of the DaaS.

2.5.1.2 Deployment Model

Private cloud: Private corporations conduct the cloud infrastructure which can also be provided by a third party and located in the corporation facility. Cloud providers can provide specialized private cloud within their organizations.

Community cloud: For different purposes, various organizations collaboratively build and make available the same cloud infrastructure and policies.

Public cloud: Cloud clients can generally benefit from the public cloud. Cloud providers has a complete possession of the public cloud with a large set of properties such as policies and charging model. Google AppEngine, Microsoft Windows Azure, Force.com and Amazon EC2, S3 are well-known examples of public clouds.

2.5.2 Cloudlet Approach

Cloudlets are not centralized like cloud and broadly-distributed and networked infrastructure. Various powerful computers (or a cluster of multicore computers) located in near vicinity provide computation and storage resources. A cloudlet is considered as a “**data center in a box**” [15]. They are only need internet connectivity and access control with respect to self-managing. They provide a simple management and are deployable to specific locations such as coffee shop and restaurants. Actually, they should be one hop away from mobile clients and accessible through high-bandwidth wireless LAN. Table 4 presents key differences between cloudlet and cloud.

In the cloudlet approach, mobile clients which work as a thin client delegate their computation intensive tasks to a cloudlet in the same LAN. The most important feature of it is that it is located in near vicinity such that the transmission time of tasks have to be in a few milliseconds in order to overcome high and variable WAN latency of the cloud. However, if there is not an available cloudlet which is attached in the same LAN environment with mobile client, the computation requests can be sent to a cloud.

Table 4 Key differences: Cloudlet vs. cloud [15]

	Cloudlet	**Cloud**
State	Only soft state	Hard and soft state
Management	Self-managed	Professionally administered
Environment	Datacenter in a box at business premises	Machine room with power conditioning and cooling
Ownership	Decentralized ownership by local business	Centralized ownership by Amazon, Google etc.
Network	LAN latency/bandwidth	Internet latency/bandwidth
Sharing	Few users at a time	100s-1000s of users at a time

Satyanarayanan et al. [15] proposes the temporary modification of cloudlet infrastructure via VM technology. They try to simplify cloudlet management. In the cloudlet approach, a large range of mobile clients can benefit with the least restrictions on their software. A VM overlay of the mobile client is migrated to the cloudlet infrastructure in the dynamic VM synthesis. The small VM overlay is derived from cloudlet base VM and The cloudlet combine the base VM and the overlay VM in order to create launch VM. This method is independent of specific language such Java, C#. On the contrary the other methods, processes migration or software virtualization are language dependent. (Capture speech from mobile device and then apply speech recognition and language translation). They develop a prototype called as Kimberley. The method was tested in the Linux applications and the synthesis time is measured. The author also indicated that the method requires optimizations.

Clinch et al. [72] examines the impacts of execution location on user experience. They conduct an experimental study in line with deploying cloudlets in different location and test the user experience. The public displays on which users play a game connected to the three different cloudlets. Each of them is separated to the different locations. The user experience varied according to the distance of cloudlets because of network latency. However it is accepted that some applications are latency tolerant.

2.6 Discovery of Local Machines and Services

Service discovery is a mechanism in which networked devices and services can notify each other about their availability to consume services they provided. Service discovery protocols are constructed so as to decrease administrative overworks and increase usability [24]. Although UPnP and Jini which were the most widely used service discovery protocols provide machine-to-machine communication architectures for home networking and enterprise automation applications, since the multicast DNS and DNS based service discovery mechanism such as Bonjour and Zeroconf are almost eliminated the administrative overhead, they are currently implemented protocols by most of the networked devices.

2.6.1 Jini Service Discovery

The Sun Company developed the Jini to support a distributed environment for devices to communicate with each other [73]. The main concept of Jini is to enable devices work together. Both hardware devices and software devices can be represented as Jini services. When a new Jini-enabled device is plugged into a network, it broadcasts a message to any lookup service on the network. Then the lookup service registers the new machine. Thus, client searches proper services and sends the job. Jini consists of a set of APIs and network protocol. The service is the resource which is made available in the distributed environment.

2.6.2 UPnP and DPWS

UPnP was promoted by the UPnP Forum to make devices to communicate discarding any installation steps [74]. UPnP is based on internet protocols such as HTTP, IP Multicasting, TCP/UDP, DHCP, SOAP and XML. Web Services Discovery and Web Services Devices Profile (DPWS) is first completely web services based home networking protocol which was developed by OASIS [75]. Because of disadvantages of Jini and UPnP, the need for a fully compact web services based protocol was emerged. In addition, standard WS-based protocols require too many resources such as computing power, memory and energy.

2.6.3 DNS-SD

DNS-SD is a naming structure to facilitate and classify network resources such as services and machines to become aware of availability and capability of networked devices. This protocol is based on two specific DNS resource records: DNS SRV and DNS TXT. Type and

domain specification based service records are grouped and along with name of service and key-value pairs are inserted as DNS SRV records. An entity on a network such as client searches for domain specific services via sending standard DNS messaging query. In response to a query, an appropriate service instance is returned to the client [76], [77].

Multicast DNS and DNS based service discovery (also known as Bonjour) are combined to improve the current service discovery protocol [26]. The Zeroconf and its successor, Bonjour announced by Apple, are service discovery mechanisms in use today interchangeably. The format of a service type which is specified as "<Service>.<Domain>" make all service instances available in that domain. For example, a DNS query including a name format which ends with ".local." is sent to local network to be responded by local devices with their address. DNS-SD protocol is heavily based on a set of naming formats presenting services as DNS records. To find a desired service, clients need to indicate the service types using the form "_http._tcp.example.domain". After returning a specific service instance name, a client can use this service instance to gather service's host and port number.

2.7 Security

2.7.1 Secure Sockets Layer (SSL)

The Secure Sockets Layer (SSL) and its advanced form, Transport Layer Security (TLS), are computer networking protocols that used to provide secured communication between servers and clients, and prevent third-party access (i.e. eavesdropping) [28]. To provide the security, SSL regulates authentication of the client and server, and encodes the communication between them. SSL utilizes the public and symmetric key encryption to establish a secure connection.

To initialize a secure communication, the handshaking procedure is followed by clients and servers. To authenticate the identity of the server, a client uses the digital certificate that is the public key of the server. An X.509 certificate is created according to the Public-Key Cryptography Standards (PKCS) and a Certificate Authority (CA) signed the certificate [78]. Servers acquire their certificates, and once a client connects the public key is forwarded to the client by the server. Then, the digital certificate is validated by the client, and the client assured that a server is indeed the server it asserts to be. SSL recommends some validation checks but practice of them are left to developer to decide. The key validation checks are listed below:

- Check the CA whether it is trusted or not.
- Check the signature whether it is correct or not.
- Check expiry time of the certificate whether it is valid or not.
- Check the subject of the certificate whether it is equal to the destination selected by the client.

After server authentication, the client and server assign a shared key. In order to obtain data confidentiality and integrity, this key is used to encrypt the data that is exchanged throughout the session.

Moreover, the handshaking procedure also permits client authentication. After server authentication, the client authenticates itself to the server via forwarding its certificate to the server. Then, the encrypted SSL session is established.

2.7.2 Open Authorization (OAuth)

OAuth is a widely accepted authorization standard that was presented in 2009. OAuth facilitates users to share their resources with third party applications without revealing their credentials (i.e. password). OAuth has two versions, and OAuth 2.0, new version, is not backward compatible with its antecedent OAuth 1.0. [30].

OAuth describes four roles and these roles are briefly defined below.

- **a) Client** is an application that gains authorization from resource owner and requests a protected resource on behalf of it.
- **b) Resource Owner** is an entity that is able to give access permission to its protected resources.
- **c) Authorization Server** is a server that supplies tokens to the client once successfully authenticating the resource owner and gaining authorization.
- **d) Resource Server** is a server that stores the protected resource of the resource owner and able to cater to access request via access tokens.

In OAuth, the client makes an access request to protected resources that are managed by a resource owner and hosted by a resource server. During authorization, the credentials of the resource owner are not used. The client acquires an access token that indicates various attributes such as scope and life time. The token is provided by an authorization server [79].

Figure 16 shows the abstract protocol flow in OAuth. The communications among the four roles and the steps followed are described below.

- **Step1:** The client requests authorization from the resource owner for the usage of its protected resource. This request usually is sent through the authorization server as an intermediary.
- **Step2:** An authorization grant is sent to the client to notify about the authorization of the resource owner.
- **Step3:** The client request an access token from the authorization server via providing the client credentials and authorization grant.
- **Step4:** The validity of client credentials and the authorization grant are approved by the authorization server and an access token is send to the client.
- **Step5:** The client presents the access token and requests the protected resource from the resource server.
- **Step 6:** The access token is validated by the resource server and if valid, the requested resource is serviced.

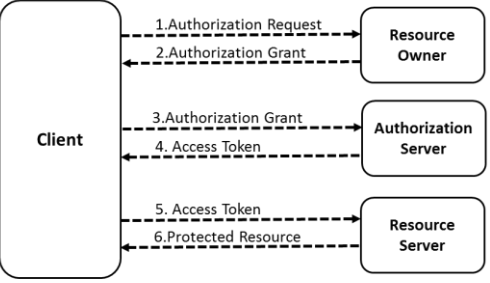


Figure 16 Abstract protocol flow [79]

CHAPTER 3

CODE OFFLOADING

The focus of this study is providing an offloading programming model that embodies distribution transparency and remote method execution. In this chapter, the proxy-based (IoC) offloading technique is explained.

3.1 Offloading Approach

Mobile devices can transparently utilize cloud resources by migrating some or all of the components of applications such as classes, objects, services or methods to resourceful servers that are in the near vicinity (surrogate) or the cloud. This approach is known as code offloading. If the execution time and/or energy consumption costs of a component are larger when it is run on the smartphone than its cloud execution, then this component is a good candidate for offloading (components to be offloaded). On the other hand, components depending on the smartphone OS such as user interface, sensors, and network classes are considered non-offloadable. The granularity level of the component to be offloaded is also important. For example, object-level granularity increases the memory cost because of larger number of components that create complex interaction patterns. Method-level granularity not only increases the number of the components to be offloaded but also forces to consider the dependency on object attributes and other methods in object oriented systems. Therefore, in this thesis, class-level granularity is chosen and instances of classes are offloaded to reduce the cost and complexity of offloading.

In this thesis, an approach that is independent of the underlying OS is proposed. By creating proxies of objects to be offloaded on both the server and the smartphone sides, distribution transparency is fully achieved. In addition, coordinating object access through a unique object identification (id) on both sides and passing this id rather than the object as the method parameter (passing by reference) overcomes the argument inconsistency problem and achieves complete distribution transparency.

3.2 Offloading Programming Model

In this section, the offloading programming model is explained in detail. An IoC technique at constructor stage is proposed. In this technique, the offloading factory is given the responsibility of creating objects at runtime. Mobile software developers request the creation of each object from the offloading factory.

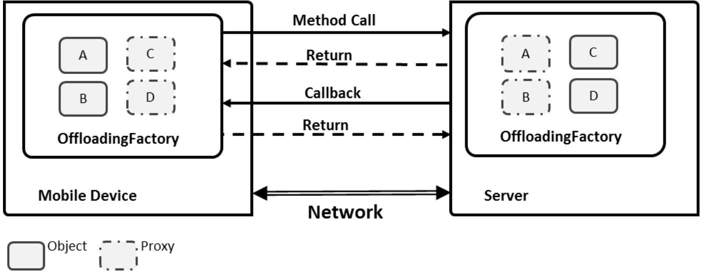


Figure 17 An overview of the offloading programming model [1]

The overview of the offloading programming model is shown in Figure 17 where the runtime snapshot of an example application consisting of classes A, B, C and D is illustrated. Classes A and B are marked as local classes, namely non-offloadable classes. Classes C and D are marked as classes to be offloaded. All class definitions are also located in the remote server at the initiation phase. When an instance of these classes (C, D) is requested on the smartphone, a proxy instance of the class is created in the smartphone, and the instance of class is created on the server side, during which a unique id is associated to this remote object to handle the method calls. The offloading factory on the server side finds the requested object using this unique object id. On the server side, assuming object C needs to call a method of object B on the smartphone, proxy B should be provided. Such reverse proxies from the server side to the smartphone side provide flexibility in software development and prevent marshalling inconsistencies. Thus, distribution transparency is completely achieved.

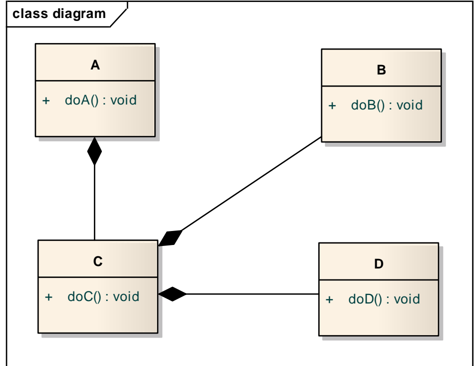


Figure 18 The class diagram of a sample application

3.2.1 *Proxy and Object Creation*

The offloading mechanism is initiated when the application requests an object creation. A class diagram belonging to an example application is presented in Figure 18. This application consists of four classes. Assuming classes A and B are marked as local classes and the other classes are marked as classes to be offloaded. Class A calls a method of class C (doC) and class C calls methods of class B (doB) and class D (doD). In this application class B can be considered as a sensor manager class of the smartphone which can provide a sensor data upon each request. Since classes C and D are computation intensive classes, they are offloaded to the server and executed there.

[1] The offloading factory creates proxies for the objects to be offloaded (on Classes C and D on the smartphone and Classes A and B on the cloud). These proxies can delegate method calls to the server and monitor all method calls.

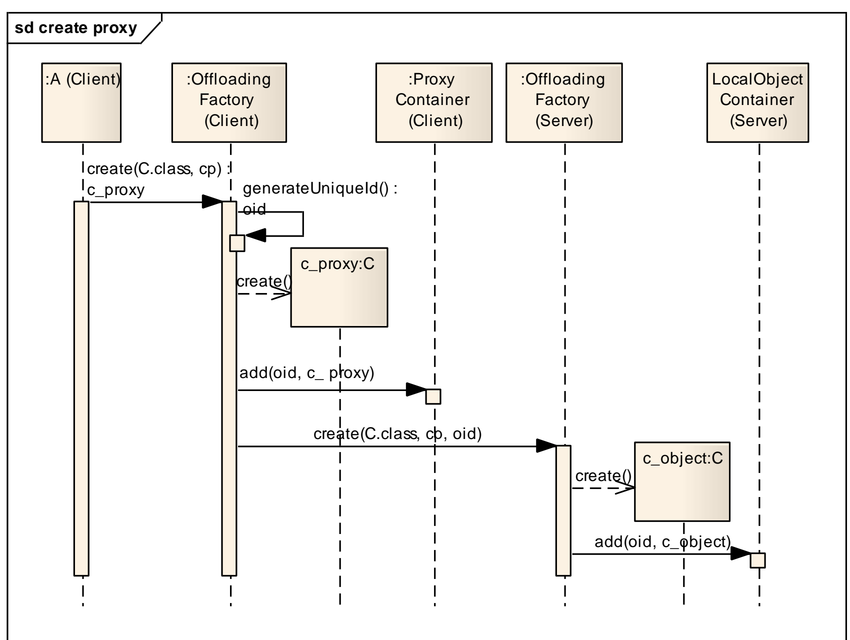


Figure 19 The sequence diagram of object creation using a proxy on the smartphone

```
//Offloading factory create-method on the smartphone and the server side
1 FUNCTION static <T> T create(Class<T> Type, final Context context,
2           final ConstructorParam cp  ){
3   IF (DecisionManager.isClassTypeOffloadable(Type)) THEN
4       //creates the proxy object dynamically and add this object to the proxy map container
5           Object proxyObj := createsProxyObject(Type, context, cp)
6           oid := generateUniqueID()
7           proxyContainer.add(oid, proxyObj);     // PMAP container
8           RETURN (T) proxyObj
9    ELSE
10        //create the local object dynamically  and add this object to the local map container
11          Object localObj  := createsLocalObject(Type,cp)
12          oid := generateUniqueID()
13          localObjectContainer.add(oid, localObj)
14          RETURN (T) localObj
15    END IF
16 END FUNCTION
```

Listing 4 The pseudo code of the offloading factory create-method

Listing 4 presents the pseudo code of object creation by the offloading factory and Figure 19 presents the proxy and local object creation mechanism. The factory first checks the class in Listing 4 (Line 3) to determine whether it is a component to be offloaded. If the requested class is a component to be offloaded, then the factory creates a proxy and associates a unique id to this proxy (Lines 5-7). This indicates that the object that is responsible for doing job is created on the server side with specified object id through the offloading factory and added to a local object container that is responsible for providing the same object for later method calls of the specified object. After creating the proxy, the method call of the proxy is sent to the remote server via an Invocation handler to be delegated to the object residing on the server.

3.2.2 Method Call

After proxy and object creation, the method calls of the proxies are delegated to the real object residing on the remote server. Proxy method call goes through an invocation handler

which takes a proxy, the object itself, the method and the arguments as the parameters of invoke method.

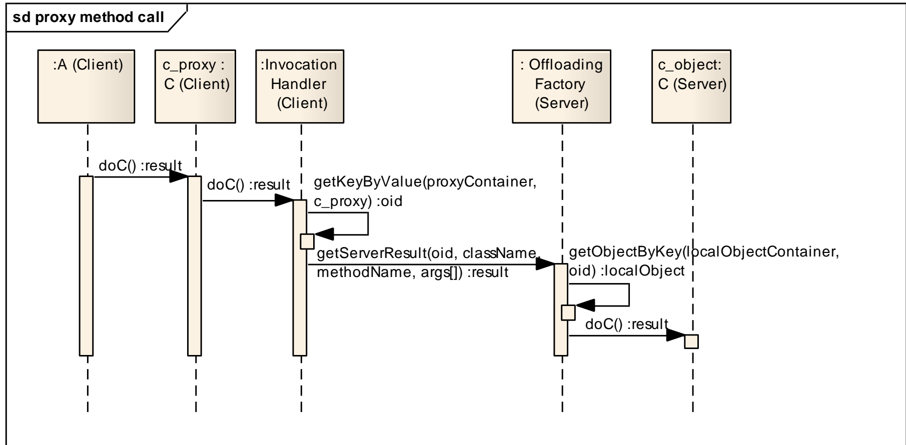


Figure 20 The sequence diagram of a proxy method call

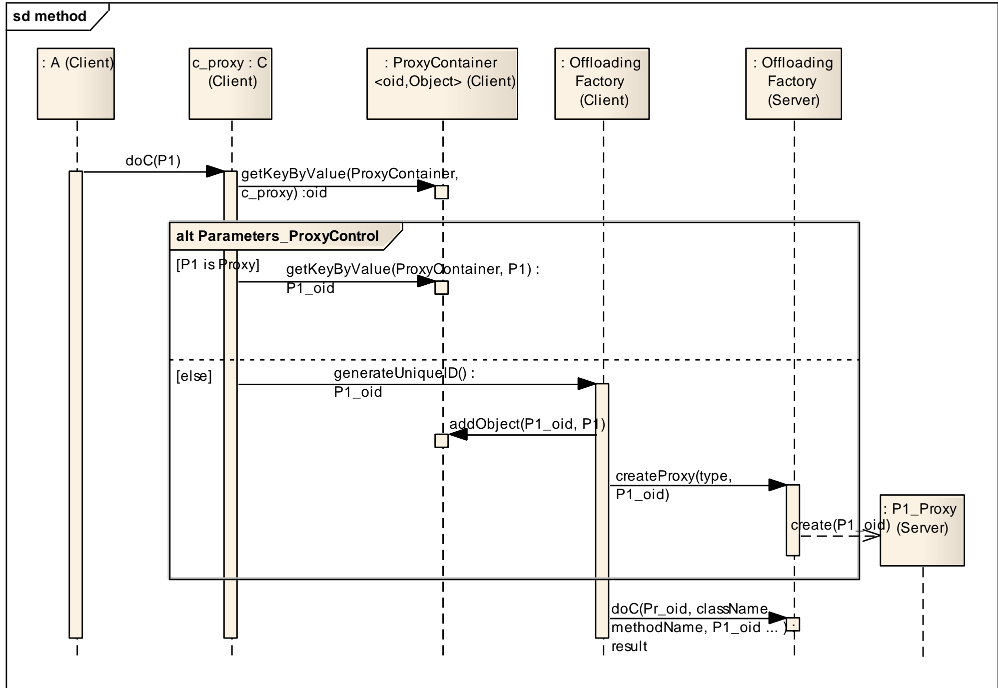


Figure 21 The sequence diagram of a proxy method call with object parameter

```
 // Proxy method handler
1     InvocationHandler handler := new InvocationHandler() {
2  FUNCTION Object invoke(Object proxy, Method method, Object[] args) throws Throwable
3    IF(applicationOffloadingChoiceSelected) THEN
4          oid := getKeyByValue(proxyObjectContainer, proxy)
5         Object[] changedMethodArgs := prepareMethodParameters(args)
6          IF(ConstructorParam is NOT NULL)
7              ConstructorParam cp := prepareConstructorParameters (cp)
8          END IF
9          RequestMessage requestMessage := new RequestMessage (oid, className,
10                    method.getName(), changedArgs, cp)
11           ReceivedMessage receivedMessage :=
12                       CommunicationManager.sendRequestMessageToCloud(requestMessage) ;
13           RETURN receivedMessage.getMethodResult()
14     ELSE
15           methodResult := ProxyBuilder.callSuper(proxy, method, args)
16           argsSize := calculateArgumentDataSize(args, methodResult)
17           throwable := new Throwable()
18           StackTraceElement[] elements := throwable.getStackTrace()
19           ProfileManager.profile(elements, method, elapsedTime, argsSize, packageName)
20           RETURN methodResult
21    END IF
22  END FUNCTION
```

Listing 5 The pseudo code of the proxy method handler

Figure 20 presents the method call of a proxy in which method call is handled through the invocation handler. As the proxy has to delegate the method call to the server, the identification number of the proxy needs to be gathered from the proxy container. The request is sent to the offloading factory of the server with the object identification number, the class name, the method name and the method arguments. Figure 21 presents the method call of the proxy with parameter. Upon a method call, the offloading factory checks whether the method parameters are of primitive types or instances of classes, or both. If a parameter is of a primitive type, it is converted to a wrapper type; however, if an object is passed as a parameter, to check whether it is a proxy or a local object is needed. Parameters can be an instance of the proxy interface, so the id of this proxy object is obtained from the map container and passed to the server. In all other cases, the offloading factory creates a proxy of this object in the remote server (if no such proxy has been created before) and gives the unique id of this object as the method parameters (Figure 21 , Listing 5, Lines 9-10). When a proxy method is called, a RequestMessage is created to set the method call information. Then the RequestMessage is sent to the remote server via TCP/IP protocol (Listing 5, Lines 9-13). In addition, for the profiling mode, proxies of all objects are created to gather profiling information Listing 5, Lines 15-20).

3.2.3 Callback Mechanism

In order to achieve complete distribution transparency in the programming model, a transparent callback mechanism needs to be provided. The callback mechanism is important because objects residing on the remote server may need to instantly access the resources of the smartphone such as sensor data.

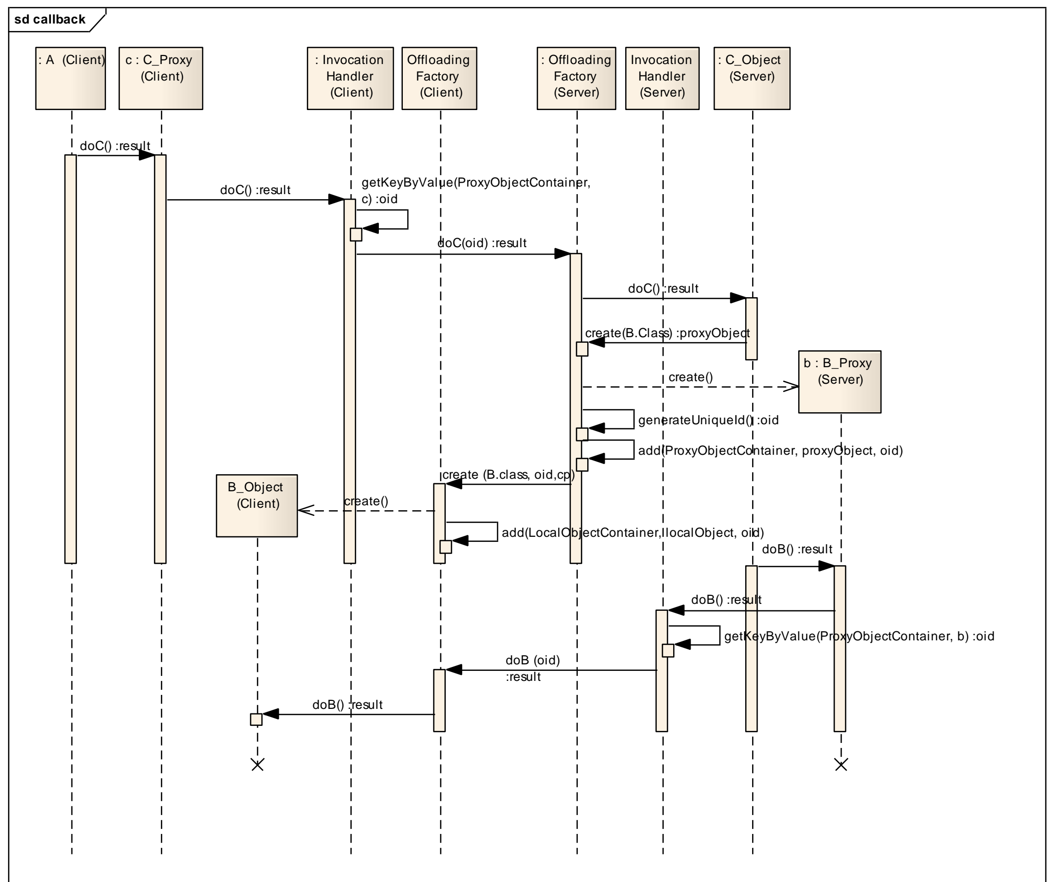


Figure 22 The sequence diagram of a callback

Figure 22 presents the callback mechanism including the proxy and object creation. This mechanism is the same with calling the remote method on the server side. The reverse proxies on the server side are used to intercept method calls on the smartphone side. In Figure 22 Object C needs to call a method of the Object B but Object B belongs to non-offloadable classes; therefore, when object C needs to create an Object B, the offloading factory returns a proxy of B, at the same time an Object B is created on the smartphone side. The reverse proxy on the server side delegates the method call to the object residing in the smartphone.

3.2.4 Processing Requests

Method call requests are sent to the server side through socket connections. In both side (smartphones and servers) a network connection manager is responsible for delivering requests.

```
//Offloading Factory processes the RequestMessage (this functionality is the same both on the
smartphone and the server end)
1 FUNCTION Object processRequestMessage(RequestMessage rm)
2   object := null;
3   processMethodArguments(rm.getMethodArgs())
4   IF(proxyObjectContainer.contain(rm.getObjectID) ) THEN
5       object := proxyObjectContainer.get(rm.getObjectID())
6   ELSE IF (proxyObjectContainer.containsKey(rm.getObjectID())) THEN
7       object := realObjectContainer.get(rm.getObjectID())
8   ELSE
9       IF (rm.getConstructorParameter is NULL) THEN
10          object := createLocalObject(rm)
11        ELSE
12          object := createLocalObjectWithConstructor(rm);
13        END IF
14    END IF
15      class := Class.forName(rm.getClassName());
16      method := class.getMethod(rm.getMethodName(),partypes)
17    RETURN  method.invoke(object, rm.getMethodArgs())
18 END FUNCTION
19 FUNCTION processMethodArguments(Object[] args)
20      Class<?> paramtypes[] := new Class[rm.getArgs().length]
21      Object[] args := sr.getMethodArgs()
22       FOREACH Object args[i] in args THEN
23          IF(args[i]  instanceof ParameterObjectType) THEN
24              ParameterObjectType pot := (ParameterObjectType) args[i]
25              IF(proxyObjectContainer.containsKey(pot.getId()) THEN
26                  args[i] := proxyObjectContainer.get(pot.getId())
27              ELSE IF (localObjectContainer.containsKey(pot.getId())
28                  args[i] := localObjectContainer.get(pot.getId())
29              ELSE
30                  IF(pot.isOffloadable) THEN
31                      args[i] := createProxyObject(pot)
32                      Paramtypes[i] := args[i].getClass()
33                      proxyObjectContainer.put(pot.getId(),obj)
34                  ELSE
35                      args[i] := createLocalObject(pot)
36                      Paramtypes[i] := args[i].getClass()
37                      localObjectContainer.put(pot.getId(),obj)
38                  END IF
39                  IF (isWrapperType(pot)) THEN
40                     Paramtypes[i] := getPrimitiveType(pot)
41                END IF
42              END IF
43            END IF
44     END FOR
45 END FUNCTION
```

Listing 6 The pseudo code of the offloading factory processing the request message

Listing 6 presents the pseudo code of processing the request both on the server and the smartphone side. On the server side, the offloading factory first checks whether an object has already been created in the remote server, and if not, creates the local object and associates the unique id of the proxy to this object (Listing 6, Lines 4-14). After the creation of the local object, this unique id is used to handle all remote method calls via proxies. On the server side, the method call continues with processing the method arguments (Listing 6, Lines 22-44). The dynamic method invocation is handled on the server (Listing 6, Lines 15-17).

3.3 Discussion on Programming Model

Proxy-based approaches that provide distribution transparency are Microsoft's DCOM [8], [9], OMG's CORBA [8], [9], Java RMI [3], Android IDL [4] and OSGi [51]. Although the current well-known distributed frameworks (CORBA, DCOM) are not explicitly targeted for mobile environments, they provide a valuable starting point to design an offloading framework. Microsoft's DCOM hosted on Windows OS computers and OMG's CORBA that is independent of the underlying OS are viable architectures for distributed application development. Both architectures specify how calls are made across a network and how references to objects are represented and maintained. However, both require IDL to define the distributed objects. Furthermore, these frameworks require the method arguments be marshalled and unmarshalled (serialized and deserialized) to be sent to the remote server (being passed by value). Passing by value may lead to argument inconsistency when the marshalled and demarshalled objects are modified by the remote call. In these frameworks, the callback functionality should be separately designed by the software developer.

The OSGi-dependent approach requires the implementation of an OSGi middleware for each application. To transform the programing code parts to suitable OSGi and IDL services, certain rules should be strictly followed and a pre-compiling phase should be completed to develop a mobile application. In addition, both in OSGi and AIDL services, developers need to allocate a great amount of time to statically designing a callback functionality for specific services. In the proxy-based offloading approach proposed in this thesis, all these limitations have been overcome.

The VM migration based offloading techniques not only incur high data communication but also require the modification of the native VM of the smartphone' OS. The proposed offloading technique does not require the modification of a VM or any pre-compiling stage to run the system. In addition, rather than the whole VM, only the computation intensive classes are offloaded to the remote server

As an alternative to coordinating object access through an object id, the object can be serialized and sent to the remote server. In this case, the offloading factory of the cloud uses this object and calls its related method. However, this may cause argument inconsistency problems if the remote object modifies the parameter object. Therefore, the serialization of parameter objects is avoided. Moreover, if a method parameter is of an object array type, the array object elements were replaced with their unique object ids.

The software developer may not want to offload a certain part of the application or specific classes where passwords are stored. In such cases, the developer can create objects without using the offloading factory or annotate them as local.

CHAPTER 4

OFFLOADING DECISION MODEL

This chapter presents an offloading decision model to determine productive application partitioning schema. Mobile applications are monitored through the offloading factory and the invocation handler of each object at runtime. The execution time of each class and dependencies between classes are gathered via this monitoring process. Section 4.1 presents application partitioning model. Section 4.1.1 explains how the weights of vertices and edges are calculated. Verification of graph partitioning model is presented in Section 4.1.2. How an application call graph is constructed by using the method call stack is given in Section 4.1.3. Lastly, Section 4.2 presents the application partitioning heuristic.

4.1 Offloading Decision Making

We created a call graph based model to store the profiling information. The graph G (V, E) consists of the vertices (V) representing classes and the edges (E) representing the method call between the dependent objects. The vertex weight is the cumulative execution time of the methods that belong to the instances of the same class. The edge weight is the cumulative time it takes to send the method arguments to the cloud, receive the results and execute the called method on the cloud. In this study, execution time optimization also contributes to the reduction of energy consumption as will be explained in Section 6.5.2.

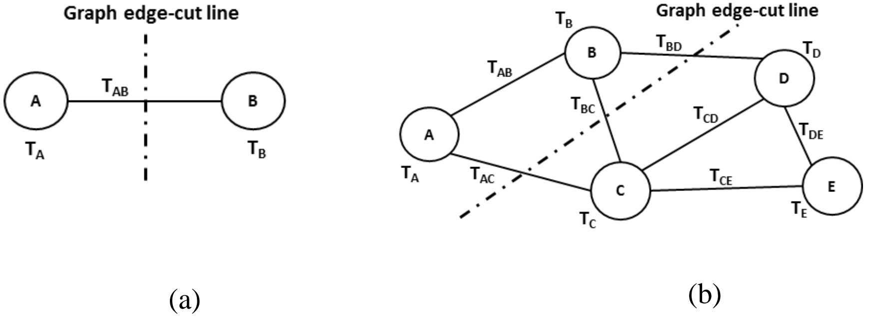


Figure 23 Application call graphs

In Figure 23, the vertices, the instance of classes, carry out the computation intensive tasks. The execution time to perform these tasks on the smartphone is to our benefit if we decide to offload these classes. On the other hand, if we offload these classes to the remote server, the time required to send and receive the data (the method arguments and return value), and the

time spent on executing the same tasks on the cloud will be the offloading cost. The aim of offloading is to increase the performance of an application. In the proposed model, the offloading gain is the reduction in overall execution time. In addition, in the scope of the thesis, gain means execution time difference (execution time difference: the local execution time to perform the task on the smartphone and the remote execution time spent on executing the same tasks on the cloud). Section 4.1.1 presents how to define the vertex and edge execution time. We propose a method for the calculation of the offloading gain to define the productive offloading decision.

Table 5 Offloading gain calculation

Figure 23a	Offloaded classes : B	Offloading gain := $T_B - T_{AB}$
Figure 23b	Offloaded classes : C, D, E	Offloading gain := $T_C + T_D + T_E - T_{AC} - T_{BC} - T_{BD}$

$$G = \sum_{i}^{N} b_i T_i \qquad C = \sum_{i,j}^{N} b_{ij} T_{ij} \qquad \text{Offloading gain} := G - C$$

Equation 1 Offloading gain calculation

Where:

T_i: The local execution time to perform the task on the smartphone

T_{ij}: The transmission time and the time spent on executing the same tasks on the cloud

Table 5 presents the calculation of the offloading gain in terms of the offloaded classes. The graph edge-cut line separates the classes to be offloaded and not offloaded (Figure 23). In Equation 1, b_i is equal to 1 if the class is marked as a component to be offloaded, and 0 otherwise. If there is an edge between vertices i and j, T_{ij} is equal to the weight of this edge; otherwise, it is 0. b_{ij} is equal to 0 if both classes are marked as components to be offloaded or local, and 1 otherwise. Following the profiling phase, the edge weight needs to be recalculated according to the local execution time of the method call (t_{local}), size of method arguments (p), and size of return values (r):

$$T_{edge_remote} = S(t_{local}) + C(p + r) \;\; ; \quad t_{local} = \sum_{i \in method} \left(T_{edge(CPU,local)}\right)$$

Equation 2 Edge cost estimation

Assuming an edge between a caller and a callee object, the callee object is offloaded to the remote server. We now need to estimate the edge weight in terms of the time that is spent on the network and on the cloud. To estimate the edge weight, we need a speed-up function S(t) of the processing time of the remote calls, since the CPU of the server is likely to be faster than the processor of the mobile device. In Equation 2, function C(p+r) presents the network time (round trip time). The sizes of the method argument data (p) and return value (r) are used to estimate the round trip time. We provide the regression analysis formulas S(t) and C(p+r) to estimate the edge cost (Section 6.3). These functions are also updated based on the history-based profiles. The model is based on the assumption that the speed up (S) and network cost (C) functions are linear mappings. The offloading gain model has similarities with the work of Niu et al. [80] with respect to the cost of vertices and edge. However, the proposed offloading model dynamically adapts itself at runtime.

After gathering the execution time of each node and data communication between nodes, the process can also be implemented to optimize energy consumption. We can detect the amount of the energy consumed to fulfill the task of a node by a constant of proportionality of the execution time of the node and a constant of proportionality of transmitted data size (KB).

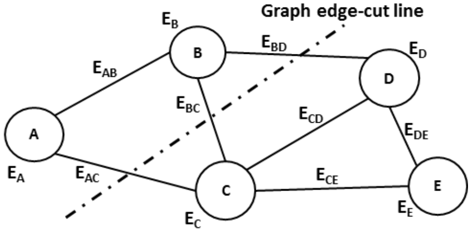


Figure 24 Energy model

Figure 24 presents the energy model of the framework. Energy consumption of the vertices and edges can be estimated. The energy consumption of the vertex i running locally on the smartphone is $E_i = \alpha * T_i$, where T_i is the local execution time and α is constant of proportionality of the execution time (Joule/ms). The energy consumption of the edge is $E_{ij} = \beta * (p + r)$. Where (p+r) is transmitted data size and β is constant of proportionality of the transmitted data size (Joule/KB). α and β constant proportionalities are provided in Section 7.1. Offloading gain: = E_i - E_{ij} and in Figure 24, offloading gain will become: E_D + E_C + E_E – (E_{AC} + E_{BC} + E_{BD}). The gain means energy consumption difference in the thesis.

4.1.1 Defining the weights of vertices and edges

To construct the call graph, we need to trace all instances of classes of the application. By creating proxies of each object, we can trace and gather all the information on the method call. Assuming we have an application containing classes A, B, C, D, the method call dependency of this application and the method life (execution time) would be as presented in Figure 25 (sequence diagram). To simplify, each method first carries out certain tasks, then calls a method of the other object, and lastly completes the method execution by performing other tasks. We converted the sequence diagram to the application call graph (Figure 26). For instance, the weight of the object D is t_{10} (t_{10} is the method execution time, C is the caller and D is the callee) and the edge weight between C and D is $S(t_{10})$ + $C(p_3+r_3)$. $S(t_i)$ is the speed-up function which is used to estimate the server execution time. $C(p_i+r_i)$ is the network cost function which is used to estimate the network time using p_3+r_3, which is the method argument and returned data size. The cumulative weight of object C is $t_9 + t_{11} + t_4$.

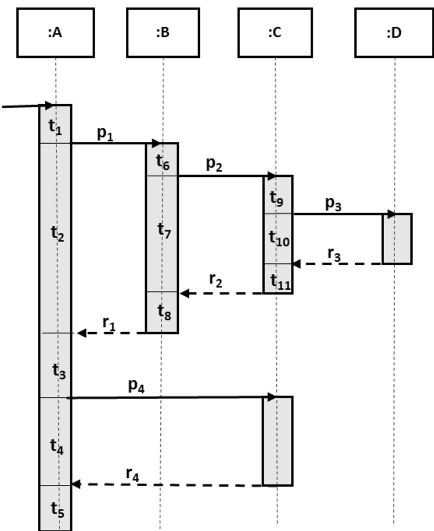


Figure 25 Calculating execution times of the vertices and edges

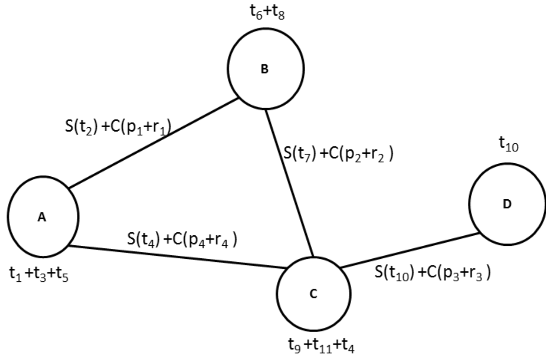


Figure 26 Graph representation of the execution times

Execution times of each object and called methods are explicitly presented in Figure 26. We backtrack from the last method call to assign the execution times to the objects and corresponding method calls. Figure 26 shows the graph representation of the classes and their method calls. This graph is constructed from the method call stack (Section 4.1.3). In Figure 26, t_i stands for local execution times and $(p_i + r_i)$ represents the data size to estimate the time spent on sending and receiving data.

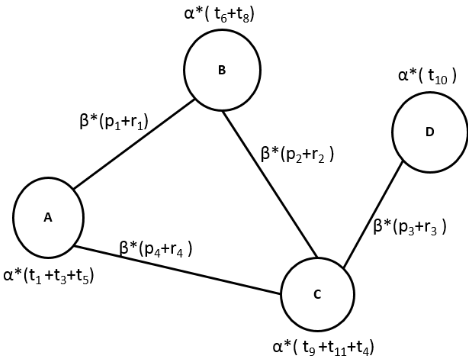


Figure 27 Graph representation of the energy model

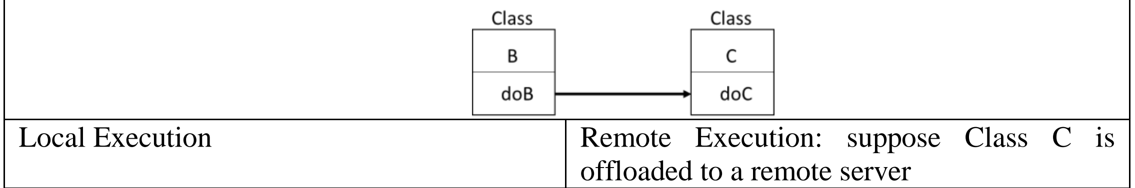
Figure 27 shows the graph representation of the energy model, and execution times of nodes and the transmitted data on the edges are converted to the energy consumption.

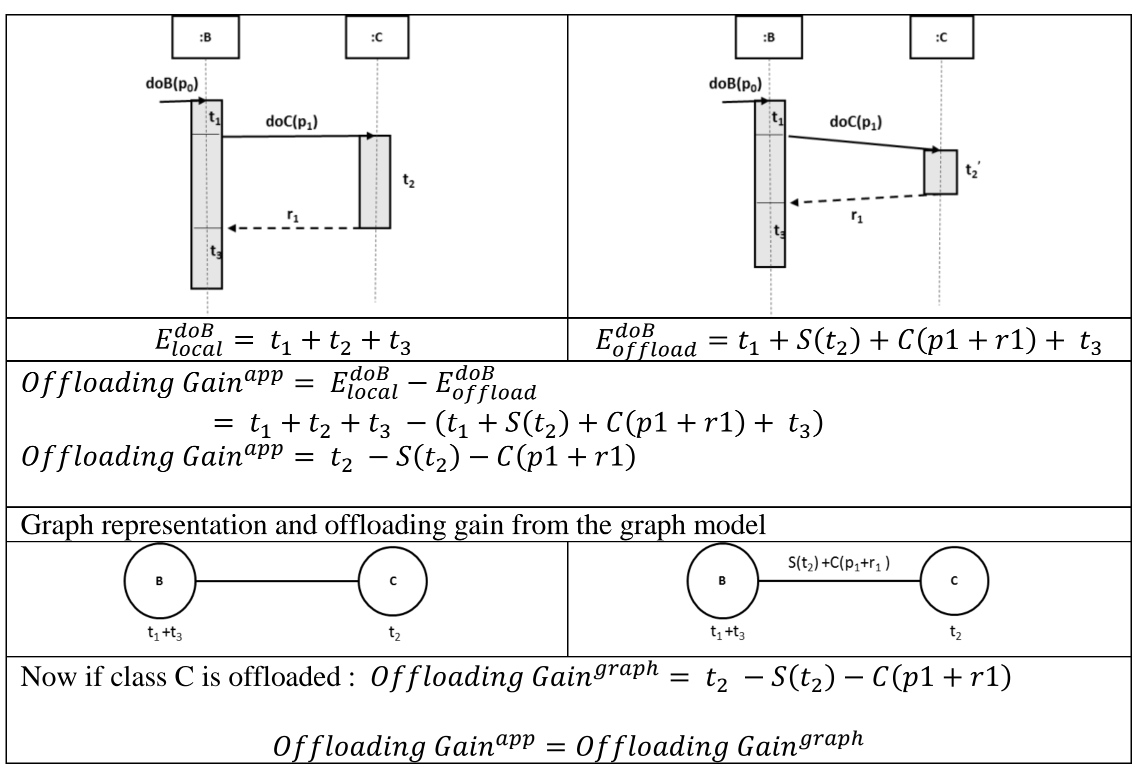
4.1.2 Verification of the Graph Model

The local and remote execution times of application components are presented in this subsection in order to verify the graph model. An application consisting of Class B, C and D is assumed and the offloading gain is calculated from both application execution times and the graph model. The offloading gain of an application is equivalent to the offloading gain of the graph model.

Case 1: An instance of Class B calls a method of an instance of Class C. The graph model and offloading gains are presented in Table 6.

Table 6 Graph model verification (two classes)

	
Local Execution	Remote Execution: suppose Class C is offloaded to a remote server



$E^{doB}_{local} = t_1 + t_2 + t_3$	$E^{doB}_{offload} = t_1 + S(t_2) + C(p1 + r1) + t_3$

$Offloading\ Gain^{app} = E^{doB}_{local} - E^{doB}_{offload}$

$= t_1 + t_2 + t_3 - (t_1 + S(t_2) + C(p1 + r1) + t_3)$

$Offloading\ Gain^{app} = t_2 - S(t_2) - C(p1 + r1)$

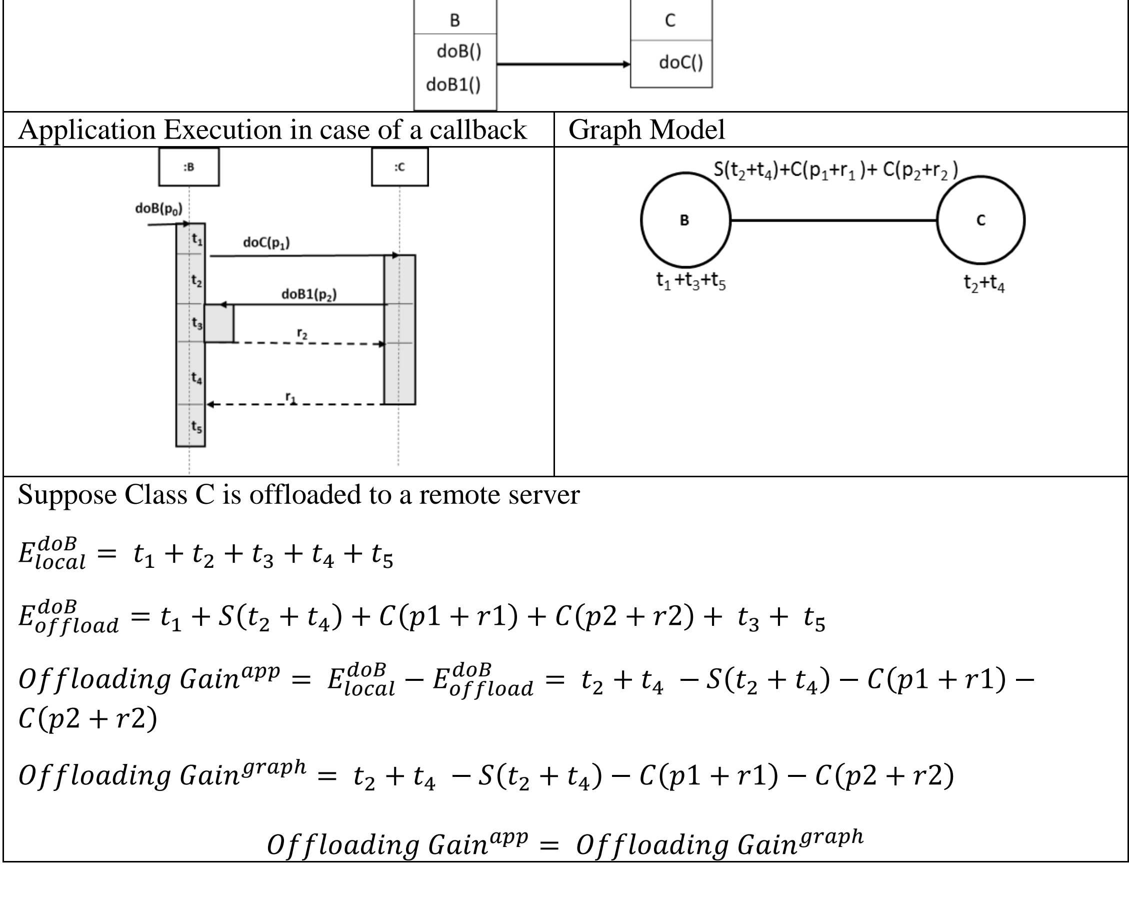
Graph representation and offloading gain from the graph model

Now if class C is offloaded : $Offloading\ Gain^{graph} = t_2 - S(t_2) - C(p1 + r1)$

$$Offloading\ Gain^{app} = Offloading\ Gain^{graph}$$

Case 2: when an offloaded component needs a callback is presented in Table 7.

Table 7 Graph model verification on a callback



Application Execution in case of a callback	Graph Model

Suppose Class C is offloaded to a remote server

$E^{doB}_{local} = t_1 + t_2 + t_3 + t_4 + t_5$

$E^{doB}_{offload} = t_1 + S(t_2 + t_4) + C(p1 + r1) + C(p2 + r2) + t_3 + t_5$

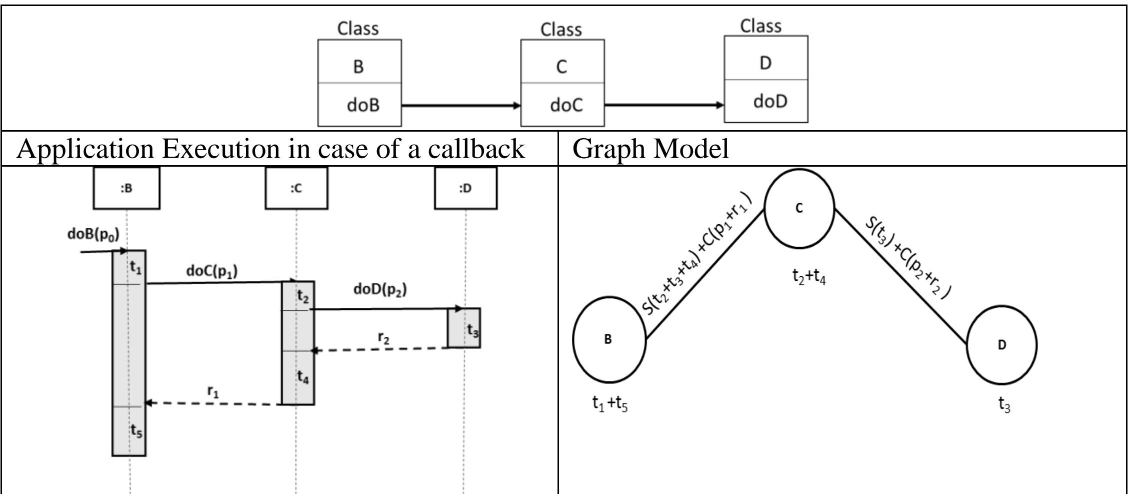
$Offloading\ Gain^{app} = E^{doB}_{local} - E^{doB}_{offload} = t_2 + t_4 - S(t_2 + t_4) - C(p1 + r1) - C(p2 + r2)$

$Offloading\ Gain^{graph} = t_2 + t_4 - S(t_2 + t_4) - C(p1 + r1) - C(p2 + r2)$

$$Offloading\ Gain^{app} = Offloading\ Gain^{graph}$$

Case 3: An application consisting of Class B, C and D is presented in Table 8.

Table 8 Graph model verification (three classes)



Suppose Class C is offloaded to a remote server

$$E_{local}^{doB} = t_1 + t_2 + t_3 + t_4 + t_5$$

$$E_{offload}^{doB} = t_1 + S(t_2 + t_3 + t_4) + C(p1 + r1) + C(p2 + r2) + t_3 + t_5$$

$$Offloading\ Gain^{app} = E_{local}^{doB} - E_{offload}^{doB} = t_2 + t_4 - S(t_2 + t_3 + t_4) - C(p1 + r1) - C(p2 + r2)$$

$$Offloading\ Gain^{graph} = t_2 + t_4 - S(t_2 + t_3 + t_4) - C(p1 + r1) - S(t_3) - C(p2 + r2)$$

The speed up (S) and network cost (C) functions are linear mappings (demonstrated in Section 6.3). Hence: $S(qx + vy) = qS(x) + vS(y)$, q and v are scaler.

$$S(t_2 + t_3 + t_4) = S(t_2 + t_4) + S(t_3)$$
$$2S(t_3) = S(2t_3)$$

$$\begin{aligned} Offloading\ Gain^{graph} &= t_2 + t_4 - S(t_2 + t_4) - S(t_3) - C(p1 + r1) - S(t_3) - C(p2 + r2) \\ &= t_2 + t_4 - S(t_2 + t_4) - 2S(t_3) - C(p1 + r1) - C(p2 + r2) \\ &= t_2 + t_4 - S(t_2 + t_4) - S(2t_3) - C(p1 + r1) - C(p2 + r2) \\ &= t_2 + t_4 - S(t_2 + 2t_3 + t_4) - C(p1 + r1) - C(p2 + r2) \end{aligned}$$

$$Offloading\ Gain^{app} = Offloading\ Gain^{graph}$$

Theorem: The minimal application execution time equals the minimum edge-cut of the graph model.

Proof: According to the results presented in these three cases, Theorem presented above is proven via the equality of offloading gains:
($Offloading\ Gain^{app} = Offloading\ Gain^{graph}$).

4.1.3 Graph Construction Algorithm

The profiling manager is responsible for constructing the call graph. A proxy object allows us to gather method call information. Thus, we can trace the current thread of the method call stack that presents all caller and callee objects and their methods. An example of the call stack is shown below. We need to trace the application-specific objects and their methods so that we define a filter function to eliminate objects and their methods depending on smartphone OS (Listing 7).

com.myproxy.OffloadingFactory$1.invoke B_Proxy.doY A_Proxy.superdoXvoid java.lang.reflect.Method.invokeNative java.lang.reflect.Method.invoke com.google.dexmaker.stock.ProxyBuilder.callSuper com.myproxy.OffloadingFactory$1.invoke A_Proxy.doX com.myproxy.MainActivity$1.onClick android.view.View.performClick android.view.View$PerformClick.run android.os.Handler.handleCallback android.os.Handler.dispatchMessage android.os.Looper.loop android.app.ActivityThread.main java.lang.reflect.Method.invokeNative java.lang.reflect.Method.invoke com.android.internal.os.ZygoteInit$MethodAndArgsCaller.run com.android.internal.os.ZygoteInit.main dalvik.system.NativeStart.main	B_Proxy.doY A_Proxy.doX com.myproxy.MainActivity
(a)	(b)

Listing 7 (a) Raw and (b) filtered method call stack

```
1 FUNCTION getCallGraph(){
2    ApplicationGraph graph  := new ApplicationGraph();
3    return graph
4 END FUNCTION
5 FUNCTION addVertexToGraph(Vertex callee, elapsedTime)
6       prevElapsedTime :=0
7       IF( graph.contains(vcallee) ) THEN
8            Set<Edge> elist := graph.edgesOf(vcallee)
9            IF(elist is not NULL )   THEN
10                FOR EACH edge e in elist :
11                     IF (e.inputClassandMethodName.equals(callee.OutputclassandMethodName) &&
12                             e.getCalleeMethodPath.contains(calleeMethodPath)) THEN
13                             loopCount := calculateLoopCount(e,vcaller,vcalle)
14                             prevElapsedTime += e.getExecutionTime()*loopCount
15                      END IF
16                 END FOR
17             END IF
18             vTime := elapsedTime- prevElapsedTime
19             Vertex callee = graph.getVertex(vcallee)
20             callee.setExtime(callee.getExtime+ vTime)
21             OffloadingFactory.checkLocal(vcallee)) // if vcallee is local, setLocal true
22        ELSE
23              vcalle.setExtime(vTime)     // if vcallee is local, setLocal true
24              graph.addVertex(vcallee)
25        END IF
26 END FUNCTION
27 FUNCTION buildCallGraph(StackTraceElement[] elements, method, elapsedTime, argumentSize,
28                                  packageName)
29    ArrayList<StackTraceElement>  stackElements :=  filterStackTraceElements (StackTraceElement []
elements)
30    IF(stackElements.size()>1)  THEN
31      // top two element of the stack reperesents the current call stack of caller and callee methods
32       StackTraceElement callee := stackElements.get(0)
33       StackTraceElement caller := stackElements.get(1)
34       IF (NOT caller.className.equals(caller.className)) THEN
```

```
35         String calleeMethodPath := getCalleeMethodPath(ArrayList<StackTraceElement>  stackElements)
36         Vertex vcallee := new Vertex(callee.getClassName)
37         Vertex vcaller :=  new Vertex(caller.getClassName)
38         // if  method is called through other methods of the vcaller
39         vcaller := backTrackCallerMethod(stackElements, vcaller)
40         addVertexToGraph(vcallee, elapsedTime)
41         IF (NOT graph.contains(vcaller)) THEN
42            OffloadingFactory.checkLocal(vcaller)) // if vcaller is local, setLocal true
43             graph.addVertex(vcaller)
44          END IF
45          edgeName := stackElements.get(1).classAndMethodName ->
46                               stackElements.get(0).classAndMethodName
47          Edge edge := graph.getEdge(vcaller,vcallee)
48          IF(edge is NULL) THEN
49              Edge edge := new Edge(edgeName, elapsedTime, argumentSize, calleeMethodPath);
50              graph.addEdge(vcaller,vcalle,edge)
51          ELSE
52               IF(edge.getExtime < elapsedTime)
53                     Edge.setxtime(elapsedTime)   // consider worst case
54                     edge.increaseEdgeFrequency()
55                END IF
56         END IF
57    END IF
58  ELSE
59     addVertexToGraph(new Vertex(stackElements.get(0).getClassName), elapsedTime)
60  END IF
61 END FUNCTION
```

Listing 8 Application Call Graph Construction Algorithm

In Listing 8, the profiling manager gathers the call stack information from the offloading factory and backtracks the last method call in the stack to construct the call graph with vertex and edge costs. The offloading factory provides the call stack, the method name, the method execution time and data size of the arguments and return value. In order to assign execution times to vertices and edges, the algorithm iterates over the stack elements of the called method and retrieves top two stack elements. Then, it creates caller and callee vertices associated with classes (Lines 32-39). First, the callee vertex needs to be added to the graph and the weight of the callee vertex is calculated by obtaining all of its edge set. The costs of edges belonging to the same method path (successor calls) are extracted from the current elapsed time (Lines 6-18). If the graph contains the same vertex, the vertex weight is aggregated (Lines 19-20). The caller vertex is added to the graph with an empty weight and updated later. The edge is created with the path name of the caller-callee method and the elapsed time is assigned as the edge weight (Lines 45-56). If the graph contains the same edge, the edge frequency is increased and the longer elapsed time is assigned to the edge weight. The algorithm also checks whether the vertices depend on the native resources of the mobile device. If so, these vertices are marked as local. Before the decision manager retrieves the graph, execution time and data size of the edge are converted to the edge costs using Equation 2.

4.2 Decision heuristic for offloading classes

By constructing the call graph, we converted the offloading decision problem to a graph partitioning problem. Graph partitioning is a major problem in many areas of computer science, such as the Very Large Scale Integration (VLSI) design, parallel processing and task scheduling [20]. Graph partitioning mainly involves dividing a graph in k number of equal sets while at the same time minimizing the edge costs of connecting vertices in different parts. If k equals two, the partition becomes a min-cut bipartitioning problem. Finding an optimal solution for graph partition is shown to be NP-Hard [20], [22]. Heuristic approaches to solving this problem include move-based algorithms, which try to iteratively improve the partition by moving a vertex or swapping vertices between parts. In this study, we

implemented the heuristic by FM [21], [23] to partition the call graph and then send the computation intensive classes of an application to remote servers.

The graph min-cut algorithm is presented in Listing 9. We implemented the FM heuristic to compute the best offloading decision based on the weight of vertices and edges. The FM heuristic only uses the edge weight to estimate the gain, so we adapted our cost model to the FM heuristic. In this process, in each pass, the vertex producing the best offloading gain is identified. After a pass, this vertex is moved to another partition. In this algorithm, we first find the candidate classes to be offloaded. For instance, smartphone OS dependent classes are marked as non-offloadable (local) classes. The candidate classes to be offloaded are moved to the local side one by one to check whether there is an increase in the total offloading gain from the graph edge-cut.

```
FM Partitioning Heuristic
G(V,E)
LocalList : = find local vertices (GUI-Activity Classes, DataBase Classes, SensorManager
Classes etc.)
MovedList :=  G(V,E) -  LocalList     // initial bipartition
Until No better partition is found
        Gain := Find Gain of offloadable vertices
                Until All offloadable vertices
                     NewGain := Move one vertex to the local side and find new gain
                     If (NewGain > Gain)   Gain := NewGain; Vtemp = vertex (i) ;
                End Until
        If Vtemp is not null   add Vtemp to the LocalList and remove from The MovedList
End Until
```

Listing 9 FM heuristic for the graph partition

```
KL based Partitioning Heuristic
Find local object (vertices)
Extract from the graph and find candidate remotable objects
Sort vertices
Best partition := Current partition
MovedList := EmptyList
repeat
start from the vertex that has the highest weight
create a bucket and add this vertex to the bucket
repeat
Select the adjacent vertices (Heavy edge and Heavy vertex)
if vertex in MovedlList then
break
compute the gain
If gain > bestgain  then
     If  Currentpartition > Bestpartition  then
     add the adjacent vertices to the bucket
     Bestpartition := Currentpartition
     Mark the vertex
     add vertex to the MovedList
     Update the adjacent vertices' gain
     bestgain := gain
     Else break
Else break
Until No more adjacent vertices
Until No more vertices
```

Listing 10 KL based partitioning heuristic

For algorithm complexity of the FM heuristic, the cost of outer loop becomes $c_1*|V| + c_2 * \sum_{i=1}^{|V|-1} t_i$, |V| is the number of vertices, the cost of the inner loop becomes $c_3 * \sum_{j=1}^{|V|-1} t_j$.

$T_{(V)} = c_1 * |V| + c_2 * |V| *(|V|-1) /2 + c_3 * |V| *(|V|-1) /2$. According to the Big-O Notation (the upper asymptotic bound of the function) , $T_{(V)} = O(|V|^2)$.

A KL (Kernighan and Lin) [22] based partitioning heuristic is developed at first. Initial partitions consisting of a part that contains vertices to be offloaded and an empty part are created. Each vertex and with their neighbor vertex are moved separately, and then the offloading gain is calculated. If the gain of the moved vertex is positive, then making that move will reduce the total cost of the edge cut in the partition. A KL based partitioning algorithm is presented in Listing 10. It firstly finds the local classes and marked these as non-offloadable classes, then extract these classes from the graph. After removing non-offloadable classes, the vertex weight is sorted in increasingly and the weightiest vertex is found to initiate the partition. It also searches the adjacent vertices that produce the maximum gain as breadth-first approach. The vertex is added to the partition if its gain is higher than the best gain. If not, the inner loop ends and the outer loop for the second vertex starts. The main idea behind KL based algorithms is the concept of the gain related with moving a vertex from a set to a different set.

CHAPTER 5

OFFLOADING FRAMEWORK ARCHITECTURE

In this chapter, the offloading framework is explained. Section 5.1 presents a high level architecture of the offloading framework. The runtime behavior of the framework is discussed in Section 5.2. The fault tolerance mechanism of the framework is explained in Section 5.3. The comparison with other offloading approaches is discussed in Section 5.4. The extension-point of the framework is presented in Section 5.5.

5.1 High Level Architecture of the Offloading Framework

The offloading framework consists of six modules on the smartphone side which are offloading factory, profiling manager, deployment manager, decision manager, discovery and network communication manager and eight modules on the server side which are offloading factory, deployment manager, library store, Android OS shadow classes, discovery, decision manager, recovery manager and network communication manager. Each of them is presented in Figure 28.

The proposed framework is composed of the following modules:

1. The **Offloading Factory** is responsible for creating and managing proxies of the requested classes. It handles access to resources between the mobile device and the cloud at runtime. By creating proxies of each object to be offloaded, the offloading framework delegates a method call to the objects located in the remote server. For instance, if a proxy is created in the mobile device that needs to access an object created in the remote server, the offloading factory associates a unique id to the object on the remote server to coordinate the access to the object through the method call. This unique id as well as the object type (class name), method name, the id of method parameters if they are object type and primitive values are then sent to the remote server to execute the method on the server side. If the object on the remote server needs certain resources of the mobile device such as sensors, it is essential to create proxies that will point to those resources located in the mobile device. The offloading factory handles the coordination of access to resources by creating proxies of all remote resources. This means that if a resource is created in the mobile device, the offloading factory automatically creates a proxy of that resource on the cloud service. This mechanism is also used for resources created in the cloud, by automatically and seamlessly creating their proxies on the mobile device side. Any errors resulting from the network communication is handled fault tolerance mechanism in Section 5.3.

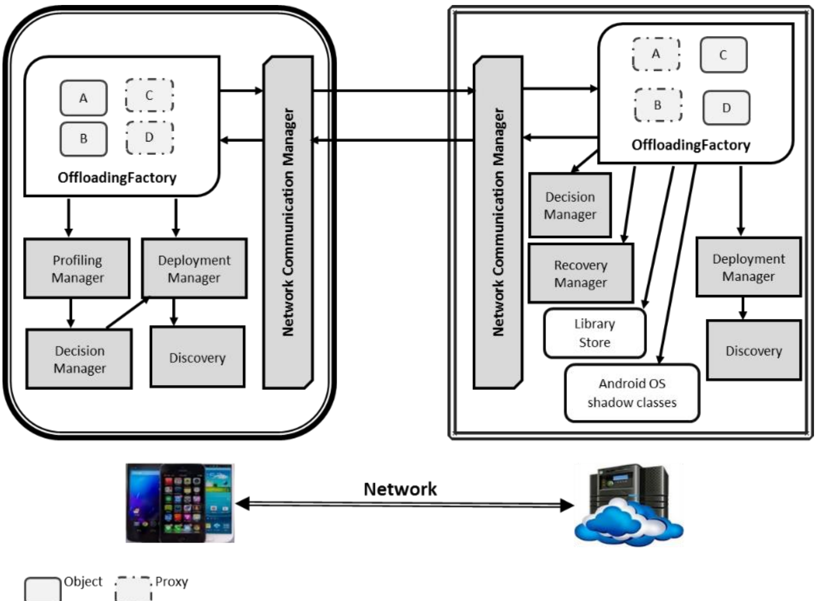


Figure 28 An overview of the offloading framework [2]

2. The **Profiling Manager** collects the monitoring information of each method call and inspects the method call stack to construct the call graph containing information from all methods required during execution, such as the execution time of the method, method call frequencies, amount of data passed as method parameters, and the size of the return value. This information is then used by the decision manager to determine the candidate classes for remote execution.

3. The **Decision Manager** decides on the instance of the classes to be executed on the remote server considering the execution time of the methods (or energy consumption), the dependencies of the classes, and available network bandwidth. The profiling and decision process are explained in Section 5.2.

4. The **Deployment Manager** is responsible for sending the server side application to a repository server. After profiling phase, it is also responsible to send the graph object and statistics to the repository server. When a mobile client wants to initiate offloading, the deployment manager sends a request to a cloud server to prepare a server process that runs the server side application and make it ready for listening in a specified port (Section 5.2). All connections and communication tasks between a mobile device and a server are handled by the network and communication manager.

5. The **Discovery module** is used to find an available cloud server or cloud service to initiate offloading. Cloud discovery is initiated when an application is started. The DNS-SD protocol [25] was implemented to discover the local servers. If there is a suitable server, a connection is established and then a specific port is allocated by the discovery module to prepare the offloading service. The network communication manager communicates through these ports.

[2] The offloading factory creates proxies for the objects to be offloaded. Profiles of all objects can then be used in the decision manager to make an offloading decision. The deployment manager is responsible for building up classes to be offloaded on the remote server.

Secure Sockets Layer (SSL) [27] for secure connection was implemented. The software developer can choose an SSL connection for application-specific classes by indicating this at object creation. In addition, a single sign-on authentication was implemented through Twitter's OAuth mechanism [29]. Once users log into Twitter, they can access all offloading services located either in nearby servers or the cloud. The remote offloading framework contains other important parts. The first is the Library Store module that is responsible for downloading and loading required libraries that are requested during method execution. The second is shadow classes of the smartphone OS. These classes were created for Android OS in order for their proxies to point to the resources in the mobile device.

5.2 Flowchart of a mobile application development with the offloading framework

In this section, the flowchart of a mobile application development by the framework is given and runtime behavior is explained from the start to the end. Responsibilities of the software developer and framework are described.

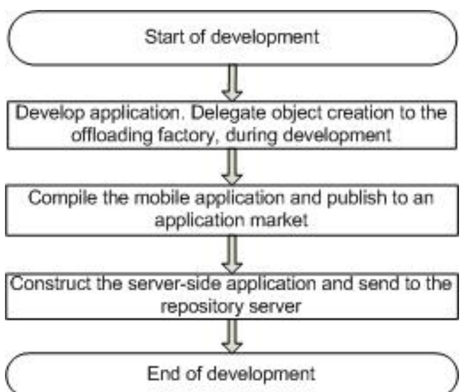


Figure 29 Flowchart of a mobile application development

Figure 29 presents step by step mobile application development by a software developer utilizing the framework. The software developer imports the framework to the desired mobile application development environment. When developing the mobile application, the creation of the objects is delegated to the offloading factory. Then, java class definitions of the mobile application are added to the server-side application template provided in the framework. The mobile and server-side template applications are built. The developer publishes the mobile application to the application market and sends the server-side application to the repository server.

Figure 30 presents runtime behavior of the mobile application. The software developer or a user downloads the mobile application from the application market and starts it. The software developer firstly enables the offloading and profiling mode and run all possible use cases of the application at least once. If software developer doesn't want to profile the application, he/she can employ static code analysis to construct call graph. Niu et al. [80] implemented static code analysis by inspecting Bytecode Instructions Count (BIC) of pure java applications. On the other hand, the framework utilizes the data collected during runtime which reflects the actual execution profile better. The framework gives different

application ids to different versions of a mobile application. If the application completes the execution in profiling mode, the framework constructs the application call graph and sends it to the repository server with a specified unique application-id. When the repository server receives the “save call-graph” request with an application-id, first it checks whether a graph with same id exists or not. If a graph with same id exists, the repository server merges the graphs and saves with the application-id. If the user or developer desires offloading, the framework checks whether a server instance is ready or not by asking to the cloud server; if a server instance is not ready, the framework requests a server process by sending the application id. Then, the cloud server checks server-side application file through the application id. If it does not exist in the cloud server, it is downloaded from the repository server and is run on the cloud server. The server instance accepts offloading requests through a TCP server socket. The server socket is bound to an unused port number. After the server instance becomes ready, the cloud server sends the server socket’s port number to the application running on the smartphone. The cloud server can also work as a load balancer; if the server instance processes reach a certain threshold value, the cloud server can run the server-side application on a different server and sends the server’s IP address and port number to the application running on the smartphone.

In order to start offloading, the smartphone application sends parameters which contain the application id, connection type, and location information to the server process which is listening on a specified port. The server process downloads the call graph from the repository server (if it does not exist in the server-side application process) and solves it according to the parameters and returns the list of classes to be offloaded. The mobile application creates objects or proxies according to the list of classes to be offloaded. In order to delegate graph solution task to the server side process, the heuristic solution class of the decision manager is marked as a component to be offloaded. When connection type or location change, the graph is dynamically re-solved in the server side process and the new list of classes to be offloaded are determined. In order to adapt to bandwidth changes, the network cost function coefficient is updated (Section 5.2.1) and the graph is also re-solved in the server side process according to changed network cost function coefficient and updated list of classes to be offloaded is determined.

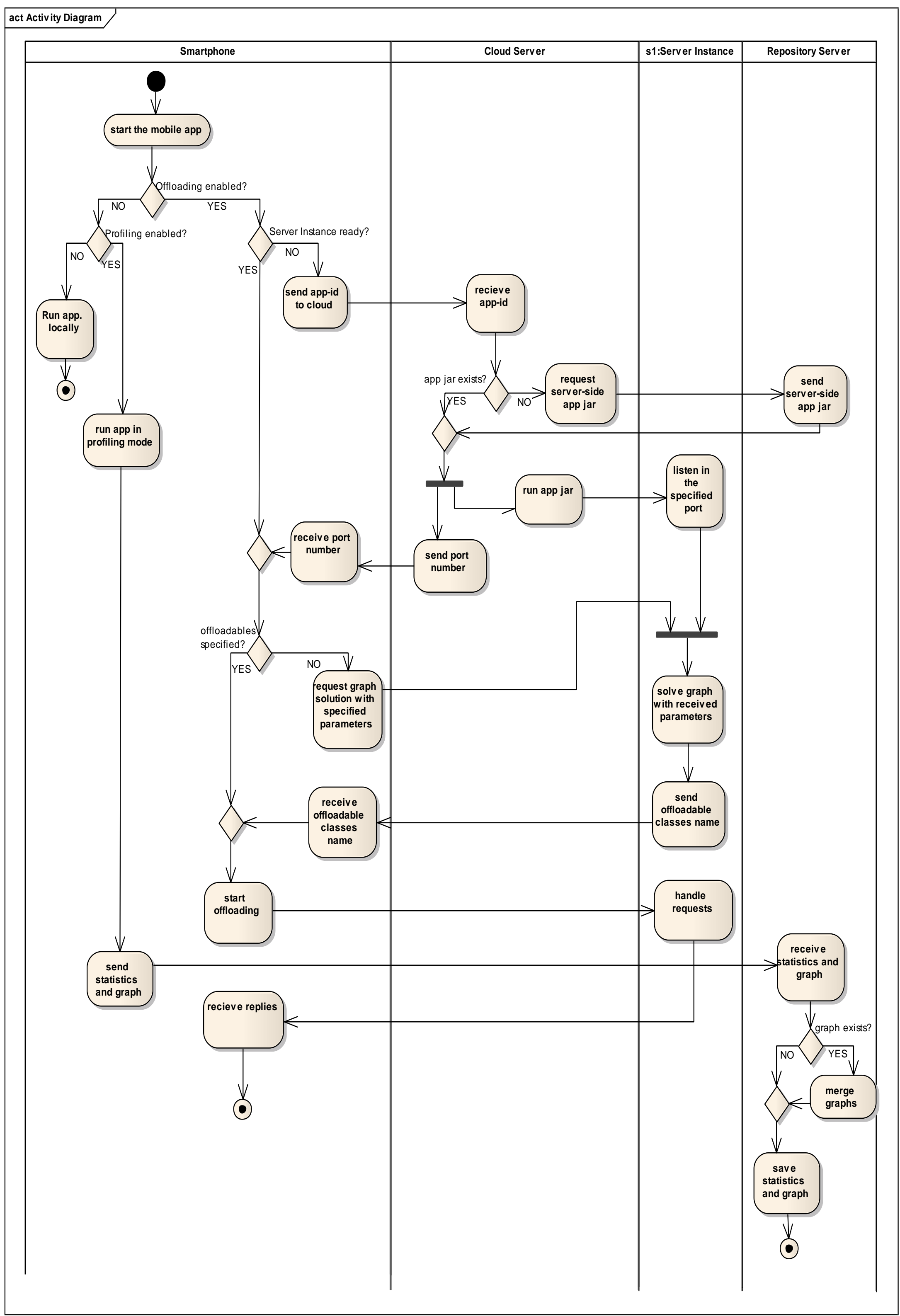


Figure 30 The activity diagram of the runtime behavior of the mobile application

5.2.1 *Determining the network cost function coefficient on-the-fly*

The framework continues to collect the network profiling data during offloading in order to adapt to bandwidth changes. The network cost function gives the network time according to the data sent and received between the smartphone and the server. In this thesis, y=c*(p+r) is used as the network cost function where p is the size of function arguments sent to the server and r stands for the size of data returned from the function. During offloading, the network cost function coefficient is updated and if any change occurs, the framework starts to re-solve the graph in server side process and finds new set of classes to be offloaded. Figure 31 illustrates the transmission time measurement. In this example,

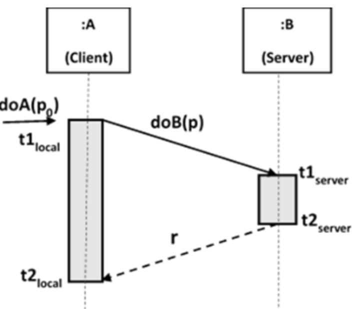


Figure 31 Transmission time

$t_{local} = t2_{local} - t1_{local}$, t_{local} stands for execution time on the smartphone.

$t_{server} = t2_{server} - t1_{server}$, t_{server} stands for server execution time

Time spent in the network is (t_{local}- t_{server}) and the network cost function is (y = c*(p+r) = (t_{local}- t_{server})) so the coefficient of the network cost function is calculated as (c = (t_{local}- t_{server}) / (p+r)) for each function call. In order to adapt the application, the solver, which defines classes to be offloaded, can be called using the updated network cost function. The measured parameters will vary in practical settings. Therefore, an exponentially weighted moving average (EWMA) is used to smooth the estimated averages and to make the application more sensitive to the recent measurements. The averaging function used is as follows:

$$C_{avg}^{t} = \alpha * C_{new}^{t} + (1-\alpha)\ C_{avg}^{t-1}$$

Where :

$C_{avg}^{t} = The\ calculated\ average\ coefficient\ value$

$C_{new}^{t} = The\ last\ measured\ value$

$\alpha = A\ constant\ smoothing\ factor\ between\ 0\ and\ 1$

When compared to maintaining the history of metrics, the overhead of using EWMA is very low. The degree of decreasing the older value is determined by a smoothing factor α, the higher value of α fastly decreases the effects of older values. The value of α should be determined by the software developer depending on dynamic responses of the application.

The frequency of sending the request of *re-solve the model* to the remote server is depending on change of the network cost coefficient. After each method invocation the network cost coefficient is recalculated and the ratio measurement scale is also calculated based on the value of the last recorded network cost coefficient used to re-solve the model. If absolute value of the ratio measurement scale of the network cost coefficient is over 30% , the graph model of the application is re-solved according to new network cost coefficient and the set of the new classes to be offloaded is adapted at runtime. Furthermore, change of the network

connection type (WiFi or 3G) and the server location (LAN server or cloud server) are also initiates the re-solve request of the model at runtime.

5.3 Fault Tolerance Mechanism of the Framework

The fault tolerance mechanism is important for distributed applications to continue requested executions even after some fault occurs during execution. The OMG's FT-CORBA [81], [82] defines specifications for distributed applications to handle failures. FT-CORBA specifications describe entity redundancy (replication of objects), fault detection, and fault recovery. Stateless, active, and passive replication styles are mostly defined by FT-CORBA. In stateless replication style, the context information is independent of invocation of objects. For objects that access a database as read-only can use stateless replication style. If at least some context information is maintained between invocations, the passive and active replication style can be used. In passive replication style, a single replication server is used as a primary and other replications are used as backups. A request from client for replicated object is forwarded by server's Object Request Broker (ORB) to the primary replication server and then replication server logs the request and dynamically calls the target object. The reply message is also logged and returned to the server's ORB. The primary replication server processes all invocations and succeeds consistency with other backups by logging and recovery mechanism. In active replication style, all replication servers gather the requests and process them. The same reply messages from the replication servers should be distinguished by the server's ORB.

In this thesis, we implemented logging and recovery mechanism presented in Figure 32 according to the OMG's FT-CORBA specifications. When a request comes to the server implementation, it is forwarded to the recovery manager. The recovery manager saves the request to the repository server. After processing the request, the reply and current objects' state in the local object container are also saved to the repository server. After that, if the server implementation does not respond to the request in case of any fault occurrence, the mobile client sends FT_REQUEST to the cloud server. The cloud server firstly checks the server process to determine whether it is alive or not. If the server process crashes, the cloud server prepares a new server process and sends the port number to the mobile client. The mobile client sends the request to the new server process. When the new server process receives the FT_REQUEST, it gathers the objects' state from the repository server and processes the invocation. The server process then returns the reply message to the mobile client.

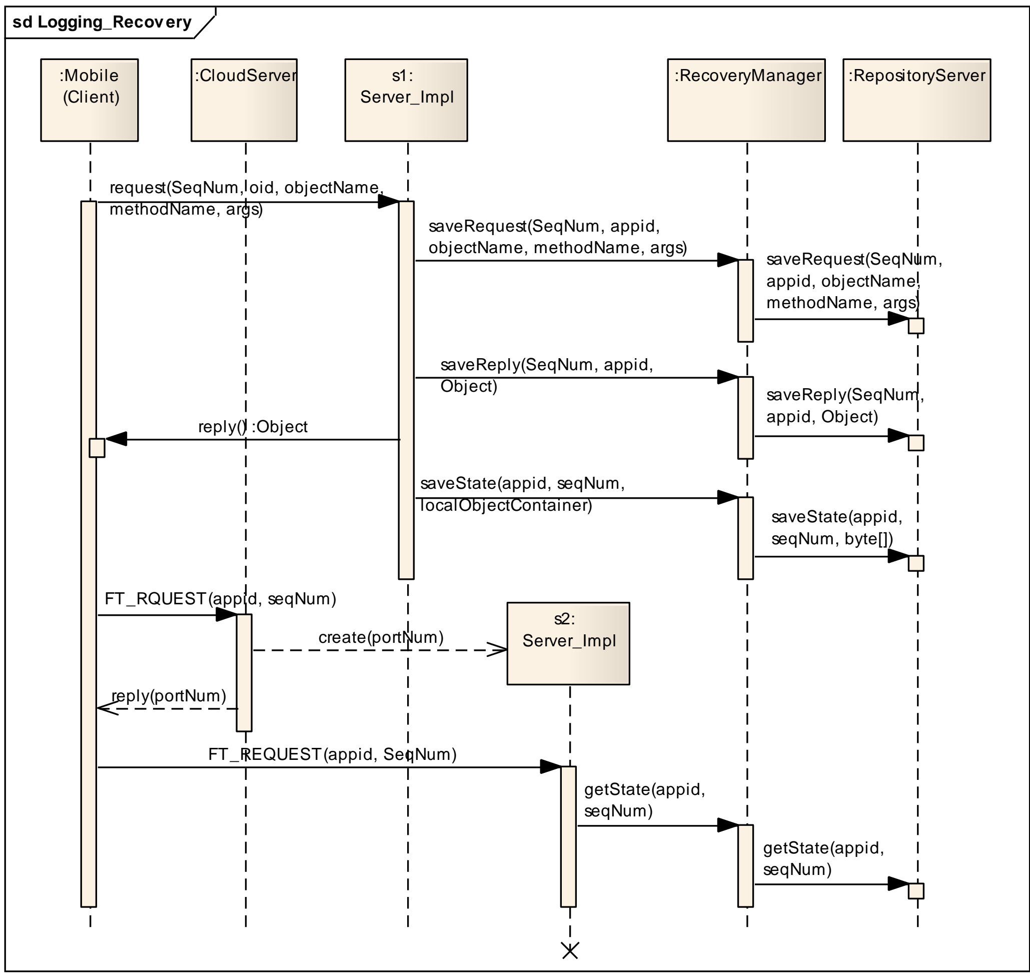


Figure 32 Logging and recovery mechanism of the framework

5.4 Comparison with Other Framework Approaches

In order to support development of mobile applications that can offload their computation intensive components to a resourceful server, many mobile cloud computing approaches [2], [6], [7], [11], [13], [37], [41], [83], [84] have been developed. Due to the complexity of adapting a mobile application at runtime, many limitations have been presented to be overcome. The complexity begins with deciding what, when, where, and how to offload with precision in order for the mobile device to gain a benefit. The studies [6], [7], [11], [13], [14], [41], [83], [84] are dependent on annotations determined by software developers. Software developer explicitly adds annotations to components such as classes, methods, services; then these frameworks at compile time convert the components to be offloaded. The decision process of these frameworks is only based on whether the pre-defined components to be offloaded are to be sent or not to the server. As expected, this decision process only checks the computation complexity of the component. In addition, history-based profiles are generally used to make such a decision. For example, in case of the method offloading, the execution time of the method is estimated based on method arguments. A comparison can be made whether to offload a specific method or not. However, the global optimal solution and distribution transparency are not completely achieved in existing works. They only offload the pre-defined (at compile time) components which are not dependent on any resource or component residing in the smartphone side.

Table 9 Comparison of the frameworks

Frameworks	Method	Partitioning	Platform	Granularity	Offloading Decision	Callback Functionality
Our Framework	Proxy - based	runtime, Graph model	Android & Java	Classes	Graph based partitioning model	Reverse Proxies, (dynamic)
AIOLOS [6]	OSGi - service	compile time, annotations	Android & Java	Classes	History based profile	Reverse IDL implementation at compile time (static)
Cuckoo [7]	Android IDL	compile time, annotations	Android & Java	Methods	Annotated methods	None
Flores and Srirama [83]	Android Service	compile time, annotations	Android & Java	Service	History based profile	None
MAUI [11]	Proxy - based	compile time, annotations	MS Windows Mobile	Methods	ILP at runtime	None
Ou et al. [2]	Proxy-based, Bytecode Instrumentation	runtime, graph model	Java	Classes	Graph partitioning	None
Abebe and Ryan [37]	Proxy – based, Bytecode Instrumentation	runtime, graph model	Android & Java	Classes	Graph partitioning	None
Kosta et al. [84]	Proxy – based, Bytecode Instrumentation	compile time, annotations	Android	Methods	History based profile	None

Table 9 presents comparison of frameworks in terms of offloading method, partitioning model, platform, offloading granularity, decision model, and callback. All existing approaches only consider the delegation model of the offloading approach. They do not take callback from server side into account. In addition, the proposed framework dynamically adapts to changes, and finds an updated list of classes to be offloaded, thereby continuing the offloading with the updated list. This is one of the strong features of the framework. Moreover, offloading performance measurements for the sample application with Flores and Srirama [83] and Cuckoo [7] frameworks are presented and compared in section 6.6.

Adaptive software systems have the ability to adjust their behaviors to provide context specific optimization [85]. In an adaptive offloading framework, this is achieved by distributing the components to remote servers to reduce the processing and memory cost of applications when necessary. If an available network bandwidth deteriorates, the network consumption of applications should be reduced. In addition, to make productive offloading decisions, the application must update its offloading decision parameters on the fly.

5.5 Extensibility of the Framework

Proposed framework is structured with a modular approach that allows implementation of new components and different functionalities. An extension-point is a reference to different implementations of a task in the framework. In the framework, we defined an extension-

point for the heuristic that solves the graph and finds classes to be offloaded. For different implementations of the graph solving heuristic, software developers should implement Heuristic interface and override solveGraph method to plug the new heuristic to the framework. Listing 11 presents extension-point for solving graph and finding list of classes to be offloaded.

```
public interface Heuristic {
            public List<Vertex> solveGraph(Graph<Vertex,Edge> graph);
}
public class NewHeuristicSolution implements Heuristic {
            @Override
            public List<Class> solveGraph(Graph<Vertex, Edge> graph) {
                        // implement new heuristic , return the list of offloadable vertices
                        return result;
            }
}
public class DecisionManager{
   Heuristic heuristicSolution;
   setHeuristic(Heuristic heuristicSolution){
            this.heuristicSolution = heuristicSolution;
   }
}
```

Listing 11 Heuristic solution extension point of the framework

5.6 Sample Application

In this section, we present an implementation of the offloading framework in Android OS and J2SE using the OR application [31] as an example scenario. The OR application takes a bitmap image as an input and computes the feature vector and then compare it with the stored feature vectors and returns the most related object information. We provide detailed explanation on how the offloading framework creates proxies and local objects, and how the mobile software developer can implement the offloading framework. Furthermore, the importance of the enabling callbacks to the smartphone side as well as the efficiency and efficacy of our offloading framework are presented.

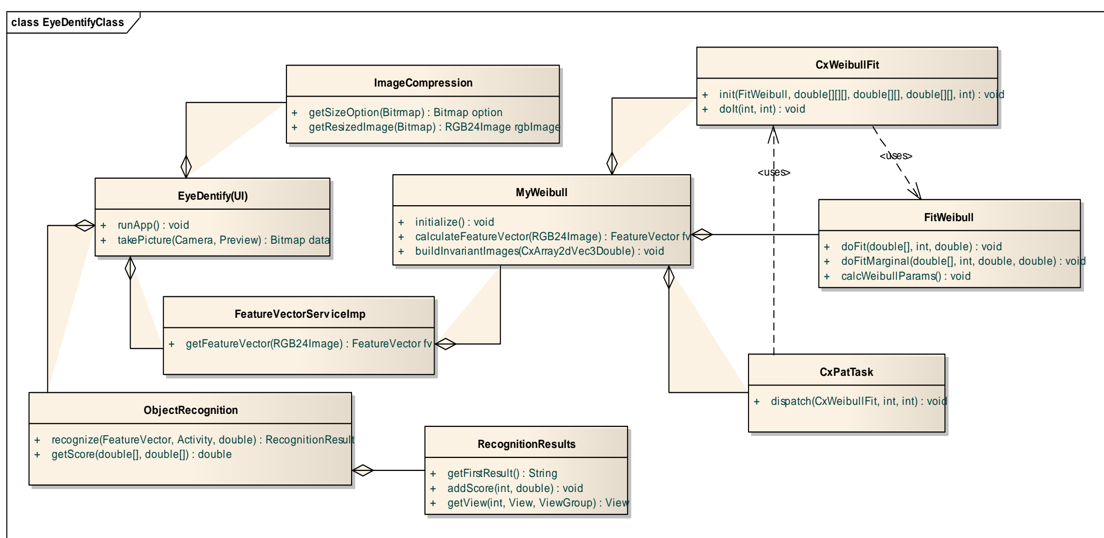


Figure 33 The class diagram of the OR application

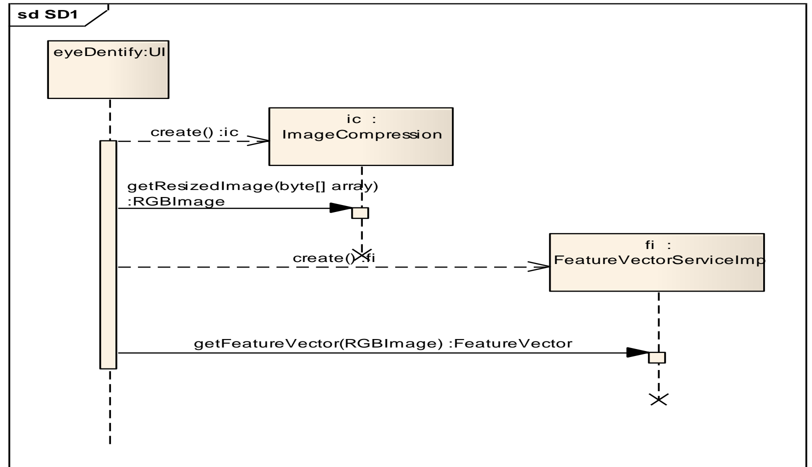


Figure 34 The sequence diagram of the OR application 1

Figure 33 presents the class diagram of the application. The EyeDentify class is a user interface class. The ImageCompression class is responsible to compress a bitmap image as an RGB image. The FeatureVector class is responsible to initialize the MyWeibull class and request the feature vector by providing a RGB image. MyWeibull, CxWeibullFit and FitWeibull classes are responsible to produce the feature vectors of an image. CxPatTask is responsible for dispatching the jobs to the CxWeibullFit. ObjectRecognition class compares the feature vectors in the recognize mode and finds the most related object according to a specified threshold value. RecognitionResult class gathers the object information from the database.

```
Class EyeDentify {
   public void onPictureTaken(final byte[] data, Camera camera) {
      public void run() {
        ImageCompression ic = OffloadingFactory.create(ImageCompression.class,
EyeDentify.this, null);
        RGB24Image rgb24Image =  ic.getResizedRGB24from(takePhotos());
        FeatureVectorServiceImpl fi = OffloadingFactory.create(FeatureVectorServiceImpl.class,
        EyeDentify.this,null)
        FeatureVector mFeatureVector = fi.getFeatureVector(rgb24Image,EyeDentify.this);

      }
   }
}
```

Listing 12 A code snippet from the OR application 1

```
Class FeatureVectorServiceImp {
public FeatureVector getFeatureVector(RGB24Image rgb24Image, Context context) throws
RemoteException {
 if (mWeibull == null) {
   ConstructorParam cp = new ConstructorParam();
   cp.setConstructorArgTypes(Integer.TYPE,Integer.TYPE,Integer.TYPE,
Integer.TYPE,Integer.TYPE);
   cp.setConstructorArgValues(COLOR_MODELS, RECEPTIVE_FIELDS,
     HISTOGRAM_BINS, COMPUTATION_WIDTH, COMPUTATION_HEIGHT);
   mWeibull = OffloadingFactory.create(MyWeibull.class,context,cp);
   mWeibull.initialize();
   FeatureVector result = mWeibull.calculateFeatureVector(rgb24Image,context);
 }
}
```

```
Class MyWeibull {
public FeatureVector calculateFeatureVector(RGB24Image image,Context context) {
  CxWeibullFit fit =
  OffloadingFactory.create(CxWeibullFit.class,context,null);
  FitWeibull fw = OffloadingFactory.create(FitWeibull.class,context,null);
  fit.init(fw, histos, betas, gammas, numberBins);
  CxPatTask cpt = OffloadingFactory.create(CxPatTask.class, context, null);
  cpt.dispatch(fit, numberReceptiveFields, numberPartialColorModels);
 return getFeatureVector();
 }
}
```

Listing 13 A code snippet from the OR application 2

Code "snippets" from the sample application are given in Listing 12 and Listing 13 to exemplify some of the noteworthy aspects of the use of the framework such as creating proxies of the classes on the smartphone side, creating proxies of the classes on the server for callbacks.

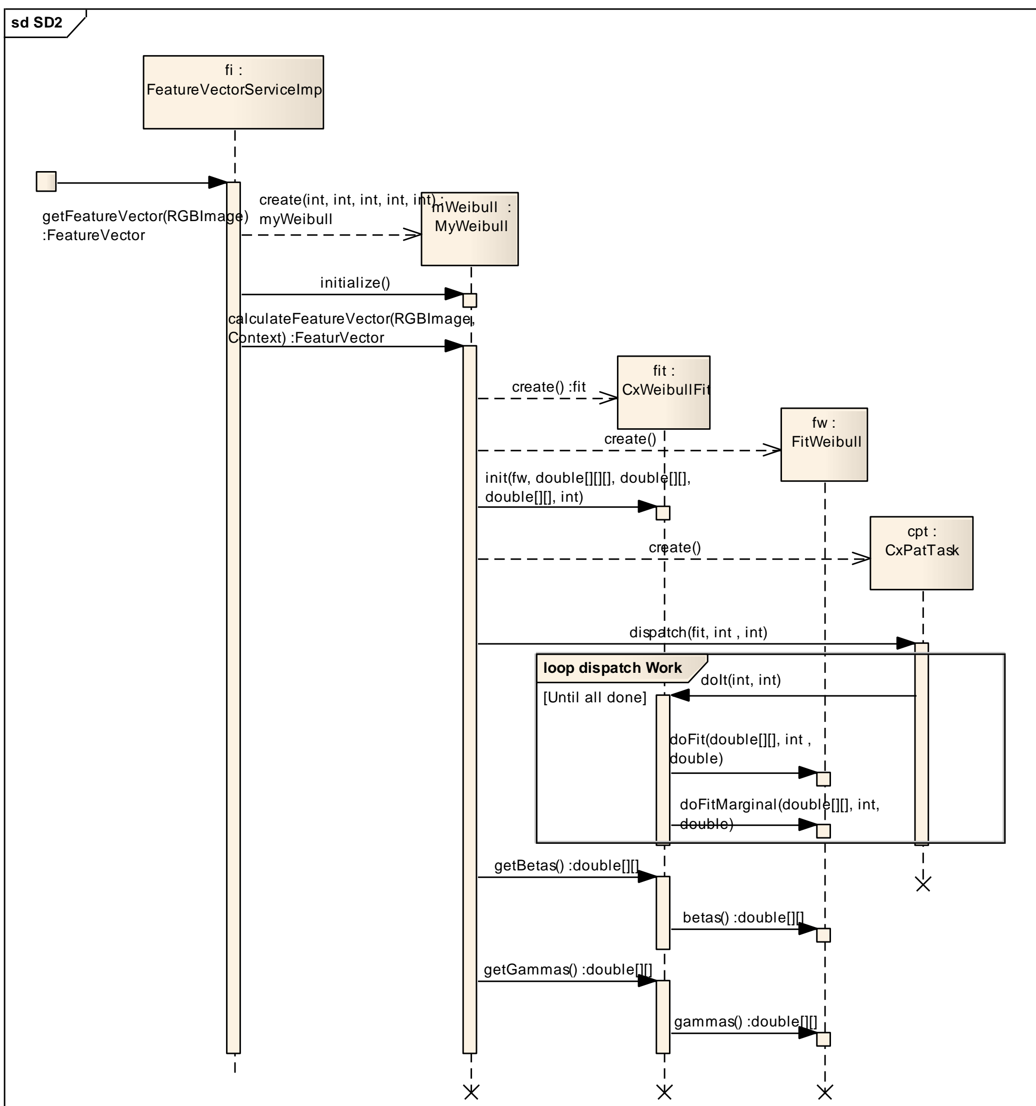


Figure 35 The sequence diagram of the OR application 2

The call graph of this application is presented in Section 6.5. Figure 34 and Figure 35 present the sequence diagrams of this application. Suppose that the FeatureVectorServiceImp and MyWeibull objects are decided to be offloaded to the server side (see Case 5 in Section 6.5). On the smartphone side, whenever the application requests the creation of the FeatureVectorServiceImp object (Figure 34, Listing 12) from the offloading factory, the factory first checks the classes to be offloaded from the decision manager, and since, in this example, the FeatureVectorServiceImp class is decided to be a component to be offloaded, the factory creates the proxy of this object. When the getFeatureVector method of the FeatureVectorServiceImp is called (Listing 12), the request is sent to the server side by the offloading factory. On the server side the offloading factory processes the request message (Listing 6) and creates the local object of FeatureVectorServiceImp and calls the requested method. In the getFeatureVector method, the offloading factory creates the local object of MyWeibull since this class is also decided to be a component to be offloaded. Here, the calculateFeatureVector method of the MyWeibull object is important. In this method, when an object creation is requested using CxWeibullFit, FitWeibull and CxPatTask (Figure 35, Listing 13), the offloading factory on the server side knows that these objects are non-offloadable and creates the proxy of these objects to send their method calls to the smartphone side. Similarly, on the smartphone side, the offloading factory processes the request message and creates the local objects of CxWeibullFit, FitWeibull and CxPatTask. After completing the method execution of these objects on the smartphone side, their return values are sent back to the server. At this point, the server completes the getFeatureVector method and returns the result to the smartphone. Figure 35 shows the method call trace of different offloading combinations in a sequence diagram. Other combinations of classes for offloading are presented in the results section. Listing 12 and Listing 13 present a code snippet from the offloading technique of the OR application. Here, the mobile software developer requests the creation of a desired object from the offloading factory only providing the class type and context, and if needed, the constructor parameters. The remaining is transparently handled by the framework.

Other offloading techniques have limitations when the offloaded code parts require smartphone resources. As described above, our offloading technique easily handles this situation by creating a proxy of the desired object, which then calls back smartphone resources. Another important characteristic of our technique is that except for the use of the factory method to create objects, offloading details such as remote object creation, remote method call, parameter passing and communication between devices are not shown to the programmer. This means that we propose a seamless technique to develop and use software modules that can be offloaded when necessary.

5.7 Using Mobile GPU for General-Purpose Computing

The GPU (Graphics Processing Unit) is a specialized circuit for accelerating the image output in a frame buffer to display which also allows GPU-accelerated computing. Together with a CPU, GPUs can be used to accelerate scientific, engineering and enterprise applications [86]. GPUs are useful at manipulating computer graphics and they are suitable for algorithms in which large blocks of data are processed in parallel. Mobile application developers can implement image processing algorithms and algorithms where processing of large blocks of data can be executed in parallel by using the GPU while the rest of the application can be run on the CPU. Nvidia Tegra, Qualcomm snapdragon and Samsung Exynos are new processors with multicore architectures and GPUs for mobile devices. The NVIDIA's Compute Unified Device Architecture (CUDA) framework, OpenCL parallel computing framework and OpenGL ES API can be used to implement applications utilizing GPU-accelerated computing for mobile devices.

There are two approaches to implement computation intensive applications in terms of GPU usage. First, if the remote server allocated for processing offloading requests is not a GPU instance, the GPU related classes should be run locally on the mobile device and the rest of

the code parts requiring heavy-computation can be migrated to the remote server. Hauswald et al. [87] statically implement this approach for an image classification application where feature extraction is handled through mobile GPU and machine learning based prediction is migrated to remote server. Second, if the remote server is a GPU instance, the computation intensive components can be migrated to remote server. For instance, Amazon EC2 also provides a GPU instance with access to NVIDIA GPUs (up to 1536 cores and 4 GB of video memory). Ayad et al. [86] statically implement a face detection application by using a GPU server instance.

In this thesis, the proposed framework can handle both situations for applications using GPU. For the first approach, the proposed framework can dynamically detect the classes related to GPU programming (OpenCL, openGL ES and NVIDIA's CUDA) and mark them as local components that are not offloaded. For solution to second approach, if a GPU server instance is available for offloading, a new speed-up factor for a GPU server instance is needed to convert the edge costs depending on the components of GPU programming of the graph model and the framework sends the components to be offloaded to the remote server after finding a productive offloading solution.

CHAPTER 6

EXPERIMENTAL EVALUATION

In this thesis, the performance of offloading was evaluated with respect to specific metrics, namely the execution time and energy consumption. The experiments which have been conducted using the offloading framework aimed to 1) investigate whether offloading might bring performance benefit or not 2) evaluate the proposed framework's effectiveness in terms of improving performance and achieving complete distribution transparency 3) validate the behavior of the proposed call graph based decision model.

Section 6.1 presents the first experiment that an OCR application is implemented to observe whether offloading reduces the execution time of the application or not. Section 6.2 presents Synthetic applications that are implemented to observe whether profiling and the graph construction algorithm work correctly or not. Section 6.3 presents the speed-up and network cost functions. Section 6.4 presents an image filtering application that is implemented to evaluate the decision process in terms of server location. An object recognition application is presented in Section 6.5 to evaluate different offloading combinations of the application's components and callback mechanism.

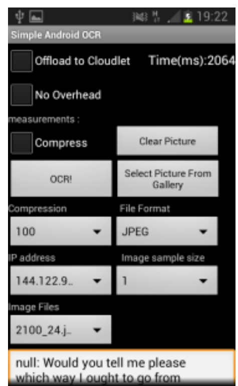
All measurement results were obtained from the following real hardware platforms: Samsung Galaxy S3 as the mobile device (1.4 GHz Quad-core, Android 4.3-operating system) and the wireless laboratory computer with a 2.0 GHz i7 263QM CPU and 8 GB RAM as the nearby server. The operating system of the server was 64-bit Windows 7. In the experimental setup, the server was accessible through a Local Area Network (LAN). The mobile device was connected to the LAN through a Wi-Fi access point with a 54 Mbps capacity. The LAN (nearby) server was connected to Internet with a 100BaseTX Ethernet connection with a 100 Mbps capacity. We also deployed our framework to Amazon EC2 [35] (m3.large instance). The operating system of Amazon cloud server instance is Windows Server 2012 r2. The experiments have been carried out in the campus LAN at the same time period (at 20:00-22:00). Smartphone and servers is only allocated for experiments.

6.1 Experiment 1

As discussed in Section 2.2, mobile application developers adapt different libraries by optimizing complex algorithms in order to avoid performance problems in smartphones. The goal of this experiment is to present whether offloading is beneficial or not via implementing a computation intensive library which is OCR library. In addition, performance and energy consumption are evaluated. The cases where the server located in LAN and in the cloud for offloading are also assessed in this experiment.

In the OCR application (Figure 36), a bitmap image of a text was taken using the phone camera. Using the training dataset, the text was computed and presented to the user. The smartphone version of the Tesseract OCR library [88] was used on the smartphone

side and a PC version of the same library was used on the server side. The proposed framework is used. In this application, a software developer indicated the dependency to Tesseract OCR library in a XML configuration file consisting of the wrapper class name, library version and the names of the methods called. Using the information from the configuration file, the Library Store module prepared the requested library with a wrapper class on the server side. The Library Store enabled the application developers to implement complex libraries without much effort for optimization.



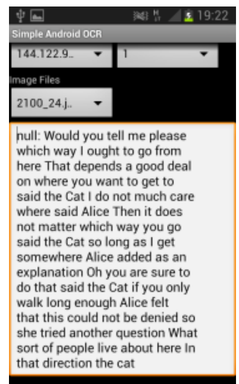


Figure 36 An OCR application

The example image inputs that have 100 to 500 words are shown in Figure 37. For each text, the OCR application was run 10 times. The average execution time results are shown in the bar graphs. The text images were also classified according to their resolutions, 480x800 and 1232x2048 pixels. The text images in Figure 37 were taken by smartphone cameras. Since these images were affected by distortion due to the amount of light, a control image input set in Figure 38 using an image-processing tool is created and copied to the smartphone. In addition, in order to ensure that the data transferred to the server was the same size as the original, the original font and image canvas size was kept the same.

6.1.1 Execution time results

Figure 39 present the execution times of the OCR application for 480x800 resolution images given in Figure 37. The images were uploaded to the remote server via a Wi-Fi (LAN) connection. Offloading reduced the execution times by 76% to 81% depending on the number of the words given in Figure 39. The offloading execution time consists of the server execution time, network transmission time, and time taken to display the results on the smartphone. As shown in Figure 40, the server execution time was higher than the network transmission time since when the image resolution was decreased to 480x800 pixels, it took longer time for the algorithm to extract a text. Furthermore, all the experiments were also conducted without using the offloading framework to see whether the offloading framework would increase the execution time. The results in Figure 39 (blue and orange bars) show that the overhead incurred by the framework was not significant.

Figure 37 The OCR image set taken by the smartphone camera

Figure 38 The OCR control image set taken by an image-processing tool

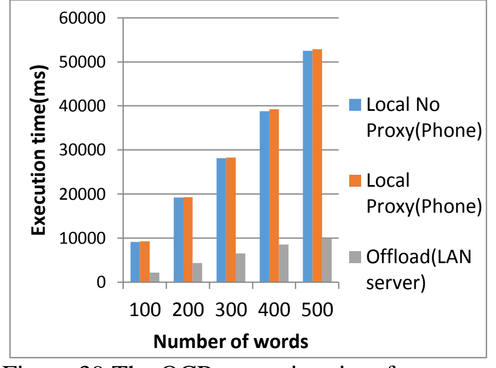


Figure 39 The OCR execution time for 400x800 resolution images

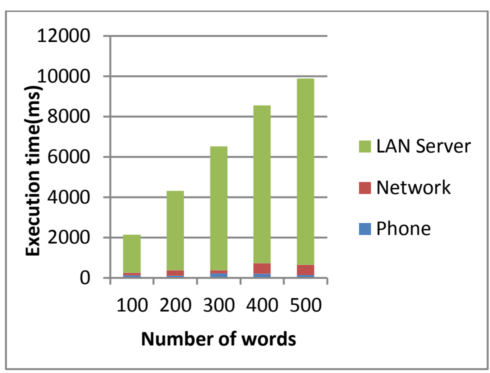


Figure 40 The offloading execution time for 480x800 resolution images

Figure 41 and Figure 42 present the execution times for 1232x2048 resolution images. The images were uploaded to the remote server via a Wi-Fi (LAN) connection. The execution times were reduced by 1% to 52% since the increase in the image size also increased the time taken for the images to be transmitted between the smartphone and the server. Figure 41 shows that the text extraction from the image with a resolution of 1232x2048 pixels gives better results both on the smartphone and on the server side. The execution times were also reduced by 77% to 84% for 1232x2048 resolution images when compared with 480x800 (Figure 39). The Tesseract OCR library was found to handle higher resolution images with considerable improvements in terms of time, even when it is run on the smartphone. Therefore, this is not related to the proposed offloading technique, but rather due to the functioning of the library.

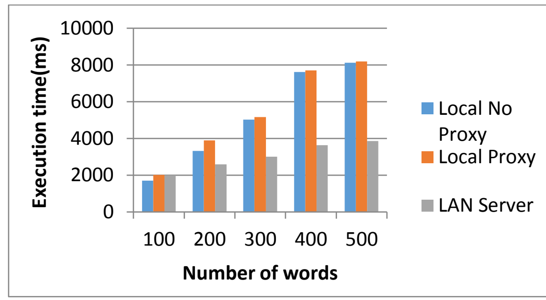


Figure 41 The OCR execution time for 1232x2048 resolution images

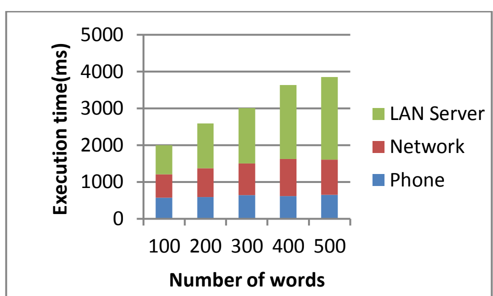


Figure 42 The offloading execution time for 1232x2048 resolution images

The effect of network connection between the smartphone and the server were also tested. A Wi-Fi Local Area Network (LAN) connection established in the university campus and a Wi-Fi Wide Area Network (WAN) connection established outside the campus were used to access the remote server.

Figure 43 shows the execution time results for these connections. The Wi-Fi (LAN) connection gave better results than other. On the other hand, the Wi-Fi (WAN) connection leads to higher network latencies. Furthermore, as shown in Figure 44, network bandwidth and latency have important effects on the response time (execution time) for Wi-Fi (WAN) connection since this has a higher communication time when compared to that of the local area network connection. In addition, the traceroute network diagnostic tool was used for Android OS to ensure that the same route was kept in WAN connection in various experiments. As a result in this experiment, offloading becomes counterproductive in cases where high data communication was required and the server was accessed through a WAN connection.

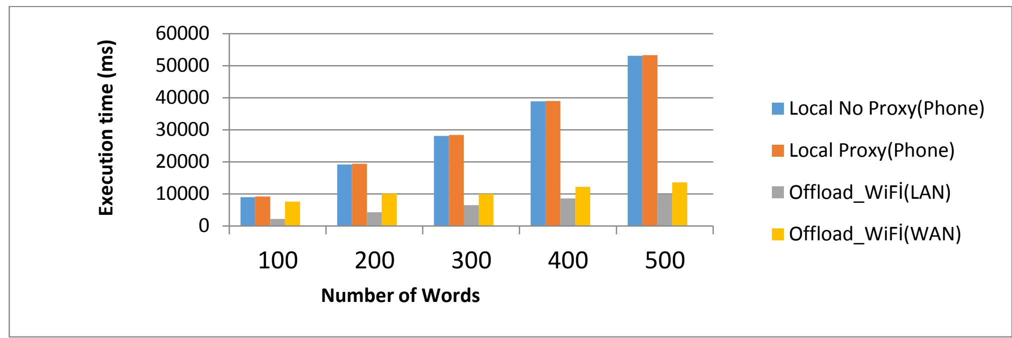


Figure 43 The OCR execution time for different network connection (400x800 pixels)

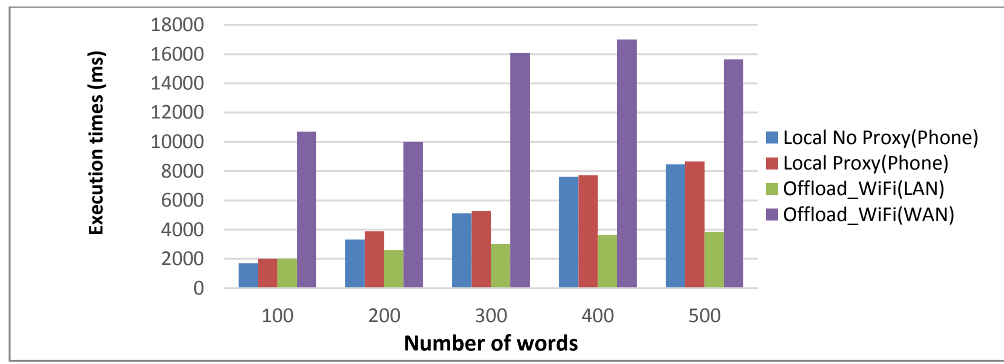


Figure 44 The OCR execution time for different network connections (1232x2048 pixels)

To eliminate any distortion on the images, the experiments with the control image sets extracted using an image-processing tool were conducted. The execution times for these images were also reduced by 55% to 74% for the 1232x2048 resolution (Figure 45 and Figure 46).

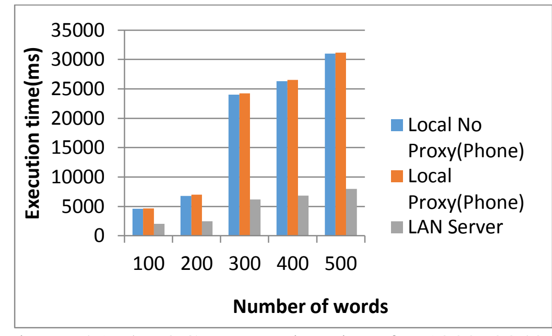


Figure 45 The OCR execution time for 1232x2048 resolution control images

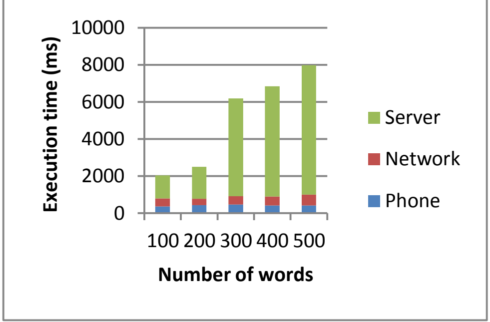


Figure 46 The offloading execution time for 1232x2048 resolution control images

Table 10 shows the precision values of the text extracted from the images. Although the Tesseract OCR library was used both in the smartphone and in the server, the precision value of the server was found slightly higher than that of the local mobile device. These higher

precision values in the server were attributed to the library not having to decrease the algorithm complexity to optimize memory in the server contrary to the situation in the phone.

Table 10 The OCR precision values

The OCR precision values (%)		
Number of words/location	*Local*	*Offloaded to the LAN server*
100	83	91
200	89	95
300	97	98
400	86	89
500	77	79

6.1.2 Energy consumption results

The application was run 10 times on both smartphones to collect data on the power consumption. The image input sizes were 0.4 MP (480x800) and 2.5 MP (2048x1232). Since the HTC Evo smartphone threw an out-of-memory exception when the number of words was more than 300, the power consumption experiments were only conducted for text images containing 100, 200 and 300 words (Figure 47). Offloading was found to save energy by 66% to 81%. Figure 48 shows the CPU and Wi-Fi energy consumptions for offloading.

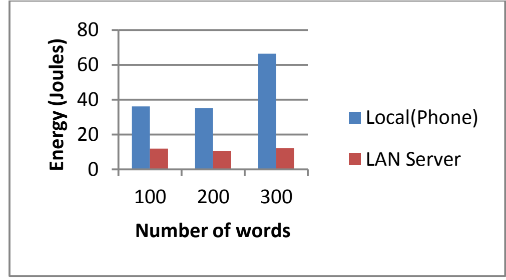


Figure 47 The OCR power consumption

Figure 48 The offloading power consumption

Figure 49 shows the energy consumption for different network connections. Depending on the available network bandwidth and latency, the WAN connection resulted in higher power consumption.

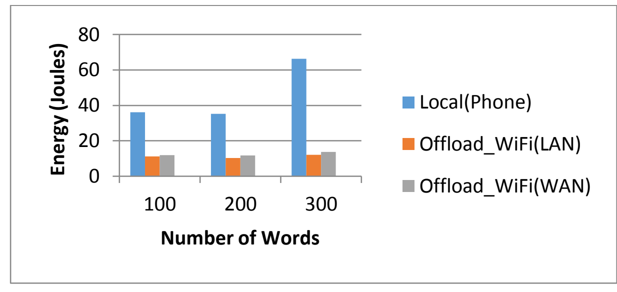


Figure 49 The OCR power consumption for different network connections

According to results of this experiment, Research Question 1 was answered, that is, offloading improves the performance and energy consumption of mobile applications.

6.2 Experiment 2 (Synthetic Applications)

The experiments in this section aimed to verify the correctness of the graph construction and the decision making heuristic of the offloading framework. Synthetic test applications which have different method calls were implemented. In these applications, the execution times of vertices were statically assigned and the execution times for edges were randomly converted. The method call stack was monitored. The method call relations, graph construction and metrics assignment for each case were compared in the test applications. After the call graph construction, the decision-making algorithm, which is based on the graph model, was applied to find the optimal offloading solution. In addition, each test application was executed several times to demonstrate different offloading decisions.

6.2.1 Synthetic Application 1

The class diagram of the Synthetic Application 1 is presented in Figure 50. MainActivity is user interface class of the application. Figure 51 presents the sequence diagram of the application. Delay for each method is statically added and presented in Table 11. The edge cost conversion function for speedup and network cost is (Math.random()*extime + Math.random()*200). The first random part is considered for speedup (extime stands for edge execution time spent for method call) and the second random part is considered for transmission time.

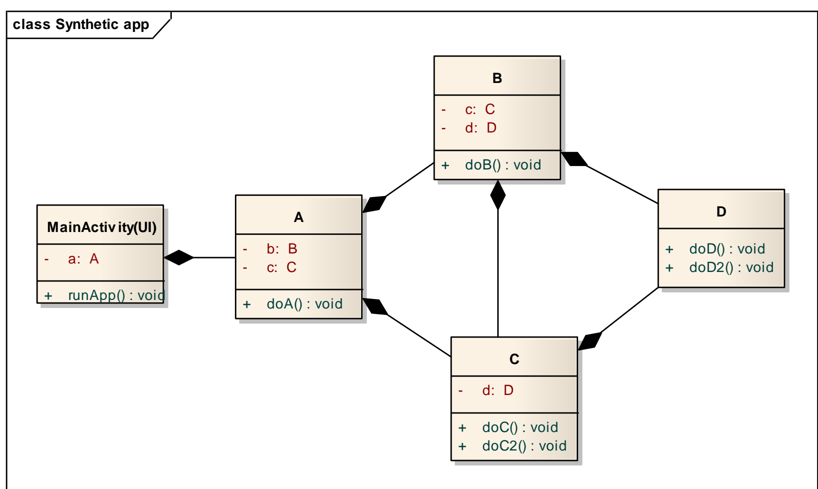


Figure 50 The class diagram of Synthetic Application 1

Table 11 Delay for each method of Synthetic Application 1

Method name	Delay (added statically - ms)
doA()	200
doB()	100
doC()	200
doC2()	300
doD()	100
doD2()	200

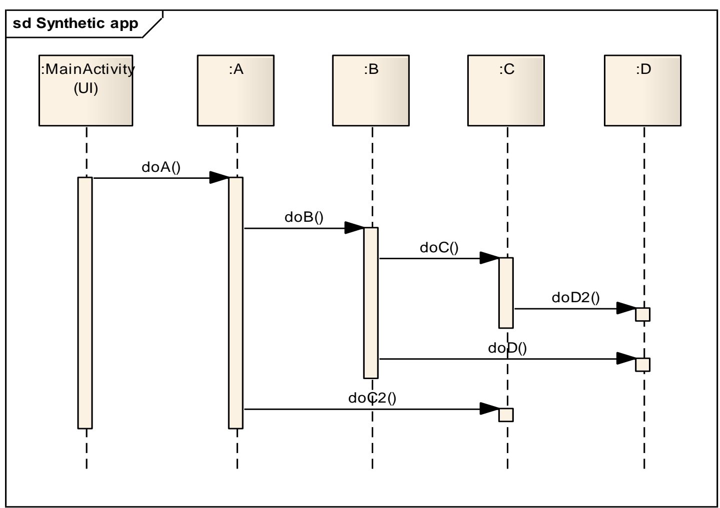


Figure 51 The sequence diagram of Synthetic Application 1

Table 12 Application call graph of Synthetic Application 1 and offloading results 1

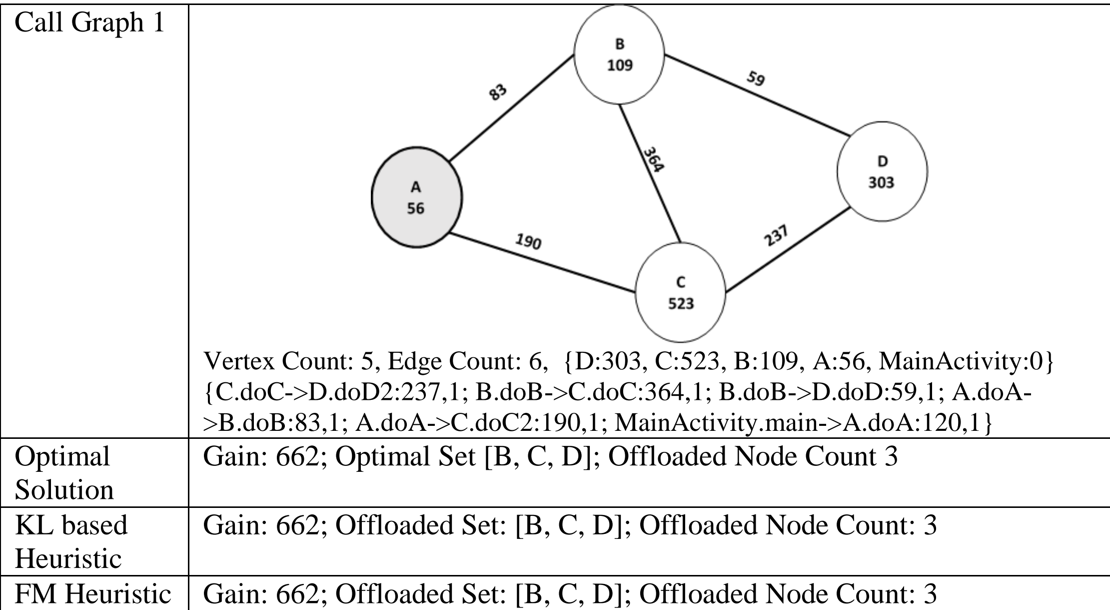
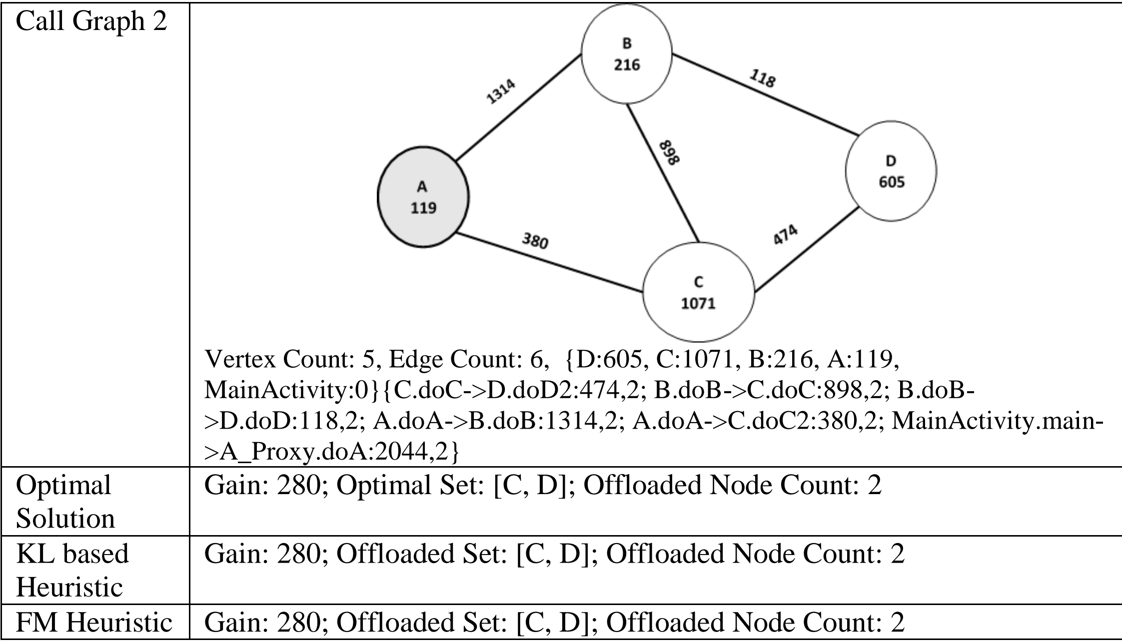
Call Graph 1	 Vertex Count: 5, Edge Count: 6, {D:303, C:523, B:109, A:56, MainActivity:0} {C.doC->D.doD2:237,1; B.doB->C.doC:364,1; B.doB->D.doD:59,1; A.doA->B.doB:83,1; A.doA->C.doC2:190,1; MainActivity.main->A.doA:120,1}
Optimal Solution	Gain: 662; Optimal Set [B, C, D]; Offloaded Node Count 3
KL based Heuristic	Gain: 662; Offloaded Set: [B, C, D]; Offloaded Node Count: 3
FM Heuristic	Gain: 662; Offloaded Set: [B, C, D]; Offloaded Node Count: 3

Table 12 presents the application call graph and the offloading decision result. The application consists of 5 classes. MainActivity is a user interface class and marked as local class which is non-offloadable. In addition, Class A was marked as a local class. Once, the application was executed, the decision model of the framework took the call graph constructed at runtime as an input from the profiling manager and applied the optimal algorithm and two offloading decision making heuristics. The results were logged. Three algorithms provided the same result for this application. Instances of Class B, C and D were eligible for offloading.

Table 13 Application call graph of Synthetic Application 1 and offloading results 2

Call Graph 2	 Vertex Count: 5, Edge Count: 6, {D:605, C:1071, B:216, A:119, MainActivity:0}{C.doC->D.doD2:474,2; B.doB->C.doC:898,2; B.doB->D.doD:118,2; A.doA->B.doB:1314,2; A.doA->C.doC2:380,2; MainActivity.main->A_Proxy.doA:2044,2}
Optimal Solution	Gain: 280; Optimal Set: [C, D]; Offloaded Node Count: 2
KL based Heuristic	Gain: 280; Offloaded Set: [C, D]; Offloaded Node Count: 2
FM Heuristic	Gain: 280; Offloaded Set: [C, D]; Offloaded Node Count: 2

The same application was executed in a for-loop to present the method call frequency. Since the edge costs were randomly assigned, for this application graph, the decision heuristics and optimal algorithm decided to offload two classes' instances which are Class C and D (Table 13)

Table 14 Application call graph of Synthetic Application 1 and offloading results 3

Call Graph 3	Vertex Count: 5, Edge Count: 6, {D:302, C:533, B:121, A:65, MainActivity:0}{C.doC->D.doD2:88,1; B.doB->C.doC:146,1; B.doB->D.doD:13,1; A_Proxy.doA->B.doB:432,1; A.doA->C.doC2:249,1; MainActivity.main->A.doA:272,1}
Optimal Solution	Gain: 427; Optimal Set: [C, D]; Offloaded Node Count: 2
KL based Heuristic	Gain: 427; Offloaded Set: [C, D]; Offloaded Node Count: 2
FM Heuristic	Gain: 427; Offloaded Set: [C, D]; Offloaded Node Count: 2

Table 15 Application call graph of Synthetic Application 1 and offloading results 4

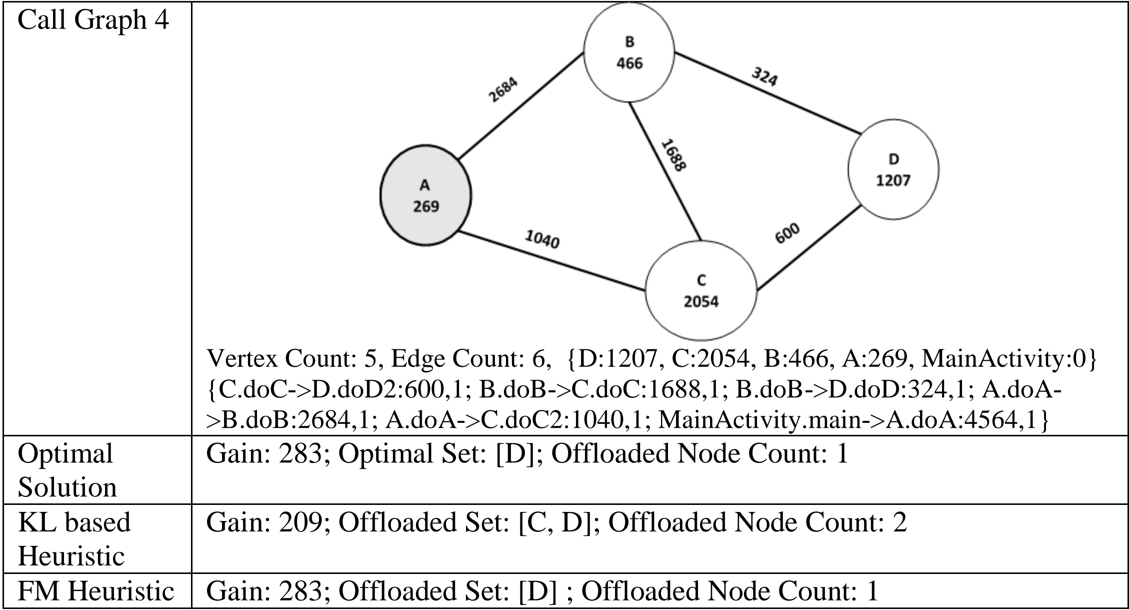
Call Graph 4	 Vertex Count: 5, Edge Count: 6, {D:1207, C:2054, B:466, A:269, MainActivity:0} {C.doC->D.doD2:600,1; B.doB->C.doC:1688,1; B.doB->D.doD:324,1; A.doA->B.doB:2684,1; A.doA->C.doC2:1040,1; MainActivity.main->A.doA:4564,1}
Optimal Solution	Gain: 283; Optimal Set: [D]; Offloaded Node Count: 1
KL based Heuristic	Gain: 209; Offloaded Set: [C, D]; Offloaded Node Count: 2
FM Heuristic	Gain: 283; Offloaded Set: [D] ; Offloaded Node Count: 1

Table 14 and Table 15 present the different execution of the same application. In Table 15, the optimal algorithm and FM heuristic decided only to offload Class D but KL based heuristic, which was the first heuristic developed, decided to offload Classes D and C. It did not find the optimal solution.

6.2.2 Synthetic Application 2

The class diagram of the Synthetic Application 2 is presented in Figure 52. MainActivity is user interface class of the application. Figure 53 presents the sequence diagram of the application. Delay for each method is statically added and presented in Table 16. The edge cost conversion function for speedup and network cost is (Math.random()*extime + Math.random()*200). The first random part is considered for speedup (extime stands for edge execution time spent for method call) and the second random part is considered for transmission time.

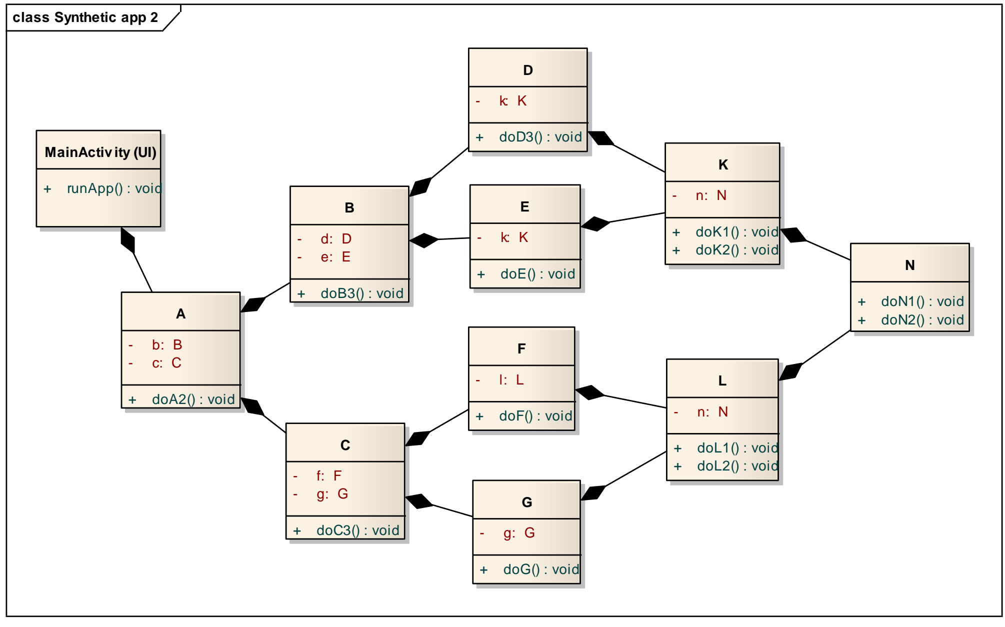


Figure 52 The class diagram of Synthetic Application 2

Table 16 Delay for each method of Synthetic Application 2

Method name	**Delay (added statically - ms)**
doA2()	50
doB3()	75
doC3()	300
doD3()	200
doE()	350
doF()	550
doG()	450
doK1()	250
doK2()	150
doL1()	450
doL2()	350
doN1()	100
doN2()	200

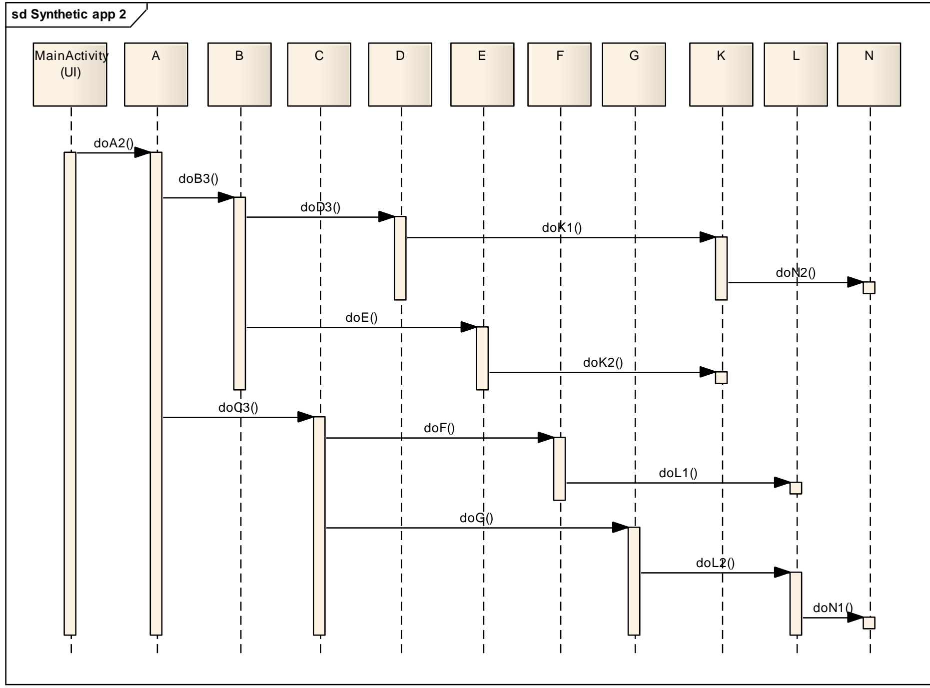


Figure 53 The sequence diagram of Synthetic Application 2

Table 17 Application call graph of Synthetic Application 2 and offloading results 1

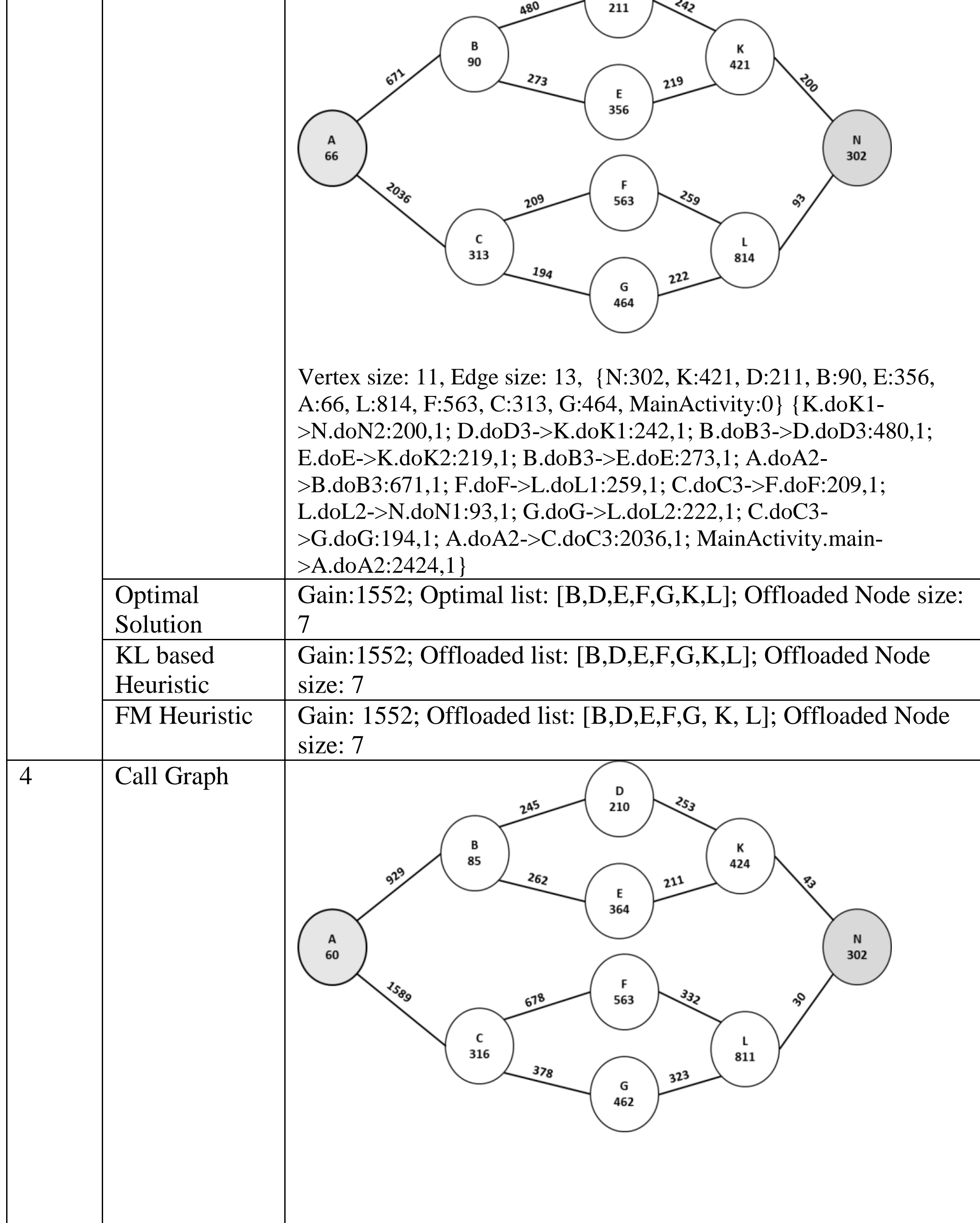
1	Call Graph	Vertex Count: 11, Edge Count: 13, {N:301, K:423, D:203, B:87, E:361, A:64, L:814, F:564,C:320, G:460, MainActivity:0} {K.doK1->N.doN2:154,1; D.doD3->K.doK1:639,1; B.doB3->D.doD3:167,1; E.doE->K.doK2:68,1; B.doB3->E.doE:598,1; A.doA2->B.doB3:1194,1; F.doF->L.doL1:476,1; C.doC3->F.doF:273,1; L.doL2->N.doN1:134,1; G.doG->L.doL2:61,1; C.doC3->G.doG:708,1; A.doA2->C.doC3:777,1; MainActivity.main->A.doA2:1761,1}
	Optimal Solution	Gain: 2073; Optimal Set: [C,D,F,G,K,L,N]; Offloaded Node Count: 7
	KL based Heuristic	Gain: 2073; Offloaded Set: [C,D,F,G,K,L,N]; Offloaded Node Count: 7
	FM Heuristic	Gain: 2073; Offloaded Set: [C,D,F,G,K,L,N]; Offloaded Node Count: 7
2	Call Graph	Vertex size: 11, Edge size: 13, {N:301, K:436, D:209, B:89, E:366, A:60, L:808, F:562, C:313, G:457, MainActivity:0} {K.doK1->N.doN2:174,1; D.doD3->K.doK1:271,1; B.doB3->D.doD3:598,1; E.doE->K.doK2:229,1; B.doB3->E.doE:407,1; A.doA2->B.doB3:612,1; F.doF->L.doL1:456,1; C.doC3->F.doF:506,1; L.doL2->N.doN1:205,1; G.doG->L.doL2:251,1; C.doC3->G.doG:561,1; A.doA2->C.doC3:2064,1; MainActivity.main->A.doA2:738,1,}
	Optimal Solution	Gain: 1549; Optimal Set: [B,D,E ,F,G,K,L,N]; Offloaded Node Count: 8
	KL based Heuristic	Gain: 1549; Offloaded Set: [B,D,E ,F,G,K,L,N]; Offloaded Node Count: 8

Table 17 (Cont.)

	FM Heuristic	Gain:1549; Offloaded Set:[B,D,E ,F,G,K,L,N]; Offloaded Node Count: 8

Table 17 presents the call graph and the decision results of the second synthetic application consisting of 11 classes. Class A was marked as a local class. In Table 17, two different execution of the same application led to two different results. In the first one, offloading decision heuristic offloaded 7 classes' instances of the application but in the second execution 8 classes were offloaded.

Table 18 Application call graph of Synthetic Application 2 and offloading results 2



3	Call Graph	Vertex size: 11, Edge size: 13, {N:302, K:421, D:211, B:90, E:356, A:66, L:814, F:563, C:313, G:464, MainActivity:0} {K.doK1->N.doN2:200,1; D.doD3->K.doK1:242,1; B.doB3->D.doD3:480,1; E.doE->K.doK2:219,1; B.doB3->E.doE:273,1; A.doA2->B.doB3:671,1; F.doF->L.doL1:259,1; C.doC3->F.doF:209,1; L.doL2->N.doN1:93,1; G.doG->L.doL2:222,1; C.doC3->G.doG:194,1; A.doA2->C.doC3:2036,1; MainActivity.main->A.doA2:2424,1}
	Optimal Solution	Gain:1552; Optimal list: [B,D,E,F,G,K,L]; Offloaded Node size: 7
	KL based Heuristic	Gain:1552; Offloaded list: [B,D,E,F,G,K,L]; Offloaded Node size: 7
	FM Heuristic	Gain: 1552; Offloaded list: [B,D,E,F,G, K, L]; Offloaded Node size: 7
4	Call Graph	

Table 18 (Cont.)

		Vertex size: 11, Edge size: 13, {N:302, K:424, D:210, B:85, E:364, A:60, L:811, F:563, C:316, G:462, MainActivity:0} {K.doK1->N.doN2:43,1; D.doD3->K.doK1:253,1; B.doB3->D.doD3:245,1; E.doE->K.doK2:211,1; B.doB3->E.doE:262,1; A.doA2->B.doB3:929,1; F.doF->L.doL1:332,1; C.doC3->F.doF:678,1; L.doL2->N.doN1:30,1; G.doG->L.doL2:323,1; C.doC3->G.doG:378,1; A.doA2->C.doC3:1589,1; MainActivity.main->A.doA2:1377,1}
	Optimal Solution	Gain: 1198; Optimal list: [D,E,F,G,K,L]; Offloaded Node size: 6
	KL based Heuristic	Gain: 1198; Offloaded list: [D,E,F,G,K,L]; Offloaded Node size: 6
	FM Heuristic	Gain: 1198; Offloaded list: [D,E,F,G,K,L]; Offloaded Node size: 6

In Table 18, the decision results were monitored. When both Classes A and N are marked as local, the different execution of the same application results in different optimal solutions for offloading. The other application topologies tested are presented in Appendix B. In Synthetic applications, the graph construction algorithm and the heuristic solution that finds optimal solutions were tested and the results show that the decision model found the optimal solution successfully.

According to results of this experiment, Research Question 2 was answered, that is, a mobile application was profiled at runtime and a call graph of the application was constructed. After constructing the call graph, the optimal graph partitioning was achieved at runtime.

6.3 Speed up and network cost functions using history-based profiles

In this section, the experiments carried out to derive the speed-up and network cost functions for the computation of edge costs are presented. For the speed-up function, a computation intensive application that finds the prime numbers from 1 to 300 was executed. This function was also executed 1 to 225 times. The execution times of the smartphone and server are collected and a regression analysis was carried out to find the estimation formula. Figure 54 presents the results of the regression analysis on the smartphone and Figure 55 on the LAN server. Combining Figure 54 and Figure 55, the speed-up function in the LAN server based on the method execution time on the smartphone (Figure 56) was found. In addition, the regression analysis on the cloud server is presented in Figure 57 and Figure 58. This speed-up function is used in Equation 2 to estimate the edge cost.

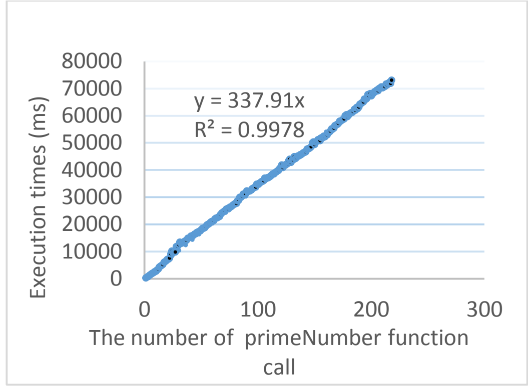


Figure 54 The regression analysis of the processing time in the smartphone

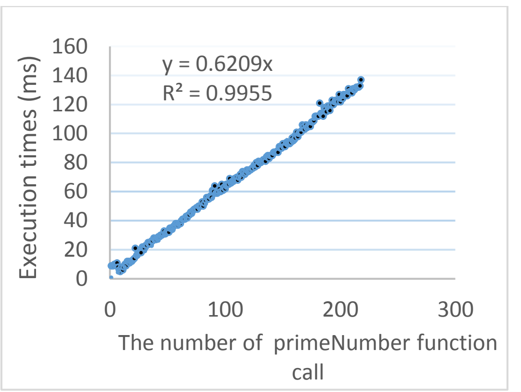


Figure 55 The regression analysis of the processing time in the LAN server

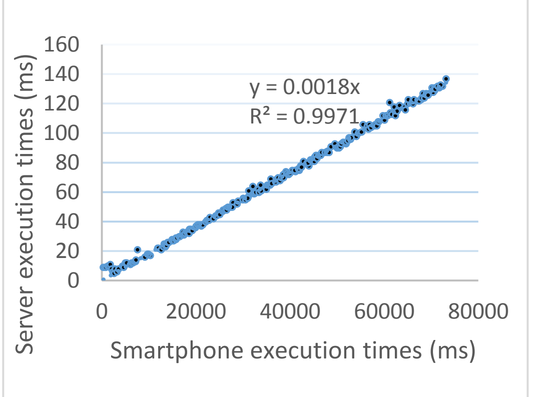


Figure 56 The regression analysis of the speed up function in the LAN server

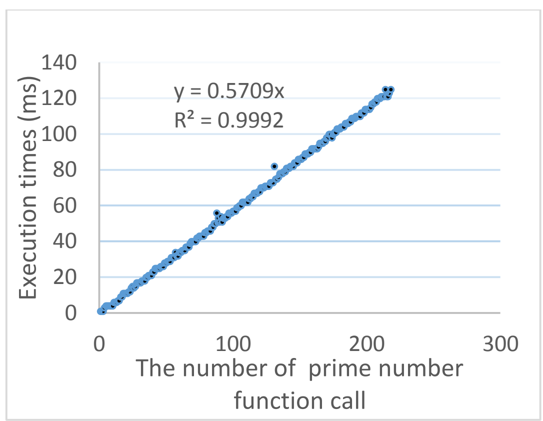


Figure 57 The regression analysis of the processing time in the cloud server

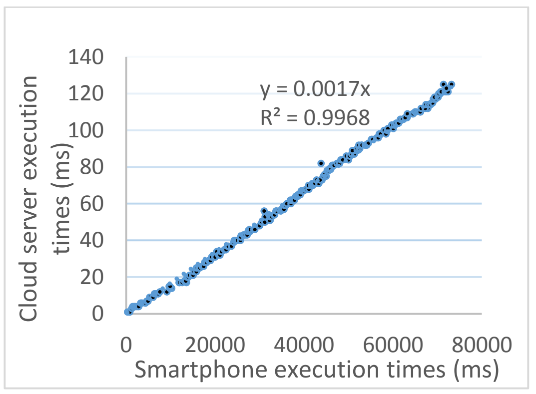


Figure 58 The regression analysis of the speed up function in the cloud server

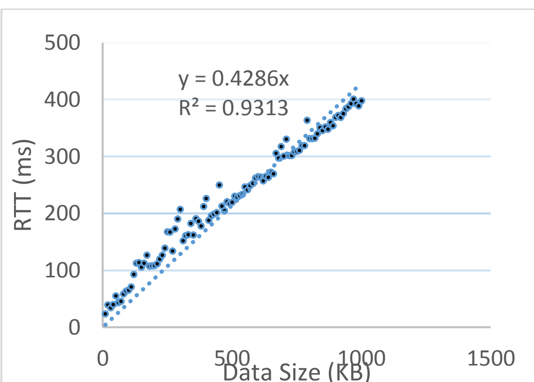


Figure 59 The regression analysis of network cost (Wi-Fi connection) in the LAN server

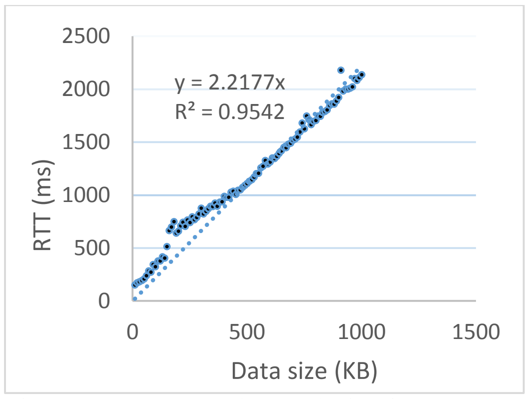


Figure 60 The regression analysis of the network cost (Wi-Fi connection) in the cloud server

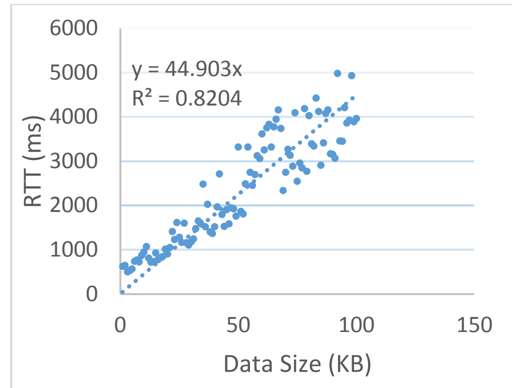


Figure 61 The regression analysis of the network cost (3G connection) in the cloud server

To estimate the network cost from the data size of the method arguments and return value, 1000 KB data started from 10 KB and increased by 10 KB every measurement were sent to the remote server and 1 KB data was received over the Wi-Fi connection. The times spent on the network were measured to perform the regression analysis. The regression equations are given in Figure 59 for the LAN server and Figure 60 for the cloud server using the WiFi connection, respectively.

Figure 61 presents network cost function for the cloud server using the 3G connection. Using these, the decision manager calculates the network time based on the data size, and the resulting network cost function is thereafter used in Equation 2. We assume that the speed up (S) and network cost (C) functions are linear mappings in Section 4.1. This experiment proves the assumption.

In regression analysis R^2 is the coefficient of determination that indicates how well data fit a statistical model. R^2 varies between 0 and 1. If it is close to 1, the regression line fits the best to the data.

$$R^2 = 1 - \frac{\sum_i (y_i - f_i)^2}{\sum_i (y_i - \bar{y})^2}$$

Equation 3 the coefficient of determination

In Equation 3, the nominator of the equation indicates the residual sum of squares. $(y_i - f_i)$ shows the differences between actual and predicted value. If this difference is small, the nominator will become small and R^2 closes to 1. The denominator part of the equation presents the total variation of data.

For the energy model, we estimated α and β constants by carrying out regression analysis. We run the prime number function and observe the energy consumption to find relationship between execution time and energy consumption. α constant for execution time is 0.06 Joule/ms. In addition, we also observe the relationship between transmission time and energy consumption by transmitting data to cloud server for WiFi connection. β constant for transmission time is 0.004 Joule/KB. In addition, Corral et al. [89] provide α constant between 0.03 and 0.07 Joule/ms for functions that have different complexities and Rice and Hay [90] provide β constant 0.005 Joule/KB. These constant proportionalities can also be chosen by the software developers.

6.4 Experiment 3

The aim of this experiment is to present the evaluation of decision processes in case of offloading to a LAN server or a cloud server. In addition, whether offloading gains measured fit to offloading gains of the graph model or not is also investigated. Smartphones benefit from offloading, if proper classes of an application are offloaded. The components which require intensive data communication such as large data passed as the method argument and high frequency of the method call located in different side are not suitable for offloading because of network cost. Especially, if offloading location is a cloud server rather than a LAN server, the offloading framework can suffer from the high WAN latency. Hence, choosing a nearby LAN server for offloading can decrease the network cost.

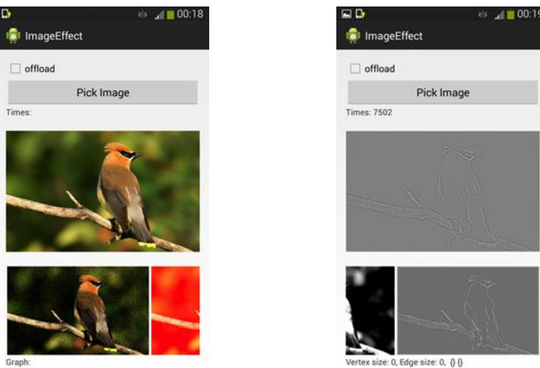


Figure 62 An image filter application

Figure 62 presents user interfaces of an image filter application. This application consists of 18 classes including 17 different image filter algorithms. These algorithms take an image as an input and apply a filter. After applying the filter, the modified image is returned.

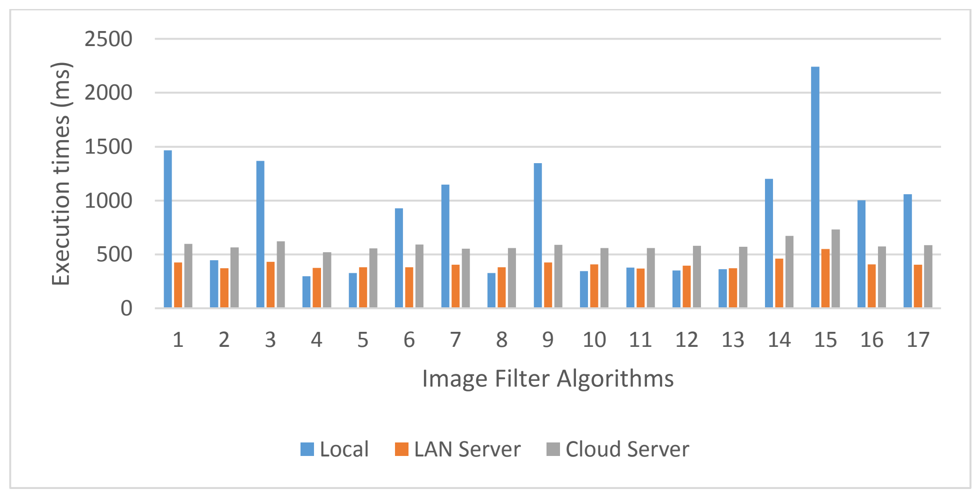


Figure 63 Comparison of execution times on local, LAN server and cloud

The comparison of response times of each image filter algorithm in terms of execution location is presented in Figure 63. As can be seen from the eighth column of Table 19, the offloading framework decides to offload 11 image filter algorithms if a LAN server was preferred for offloading location. On the other hand, if a cloud server was selected for

offloading, 9 image filter algorithms were suitable for offloading (Table 20). For example, 2- EmbossFilter and 11- GammaFilter were offloaded when a LAN server was selected but in case of a cloud server as an offloading location, they were not selected for offloading because of network cost. The offloading framework dynamically re-solves the decision model to find classes to be offloaded when the server location is changed.

Table 19 Comparison of Offloading gains for a LAN server

	Algorithms	**Local**	**LAN Server**	**Gain of the measurement**	**Vertex cost**	**Edge cost**	**Gain of the model**
1	GaussianFilter	1467	425	1042	1190	191	999
2	EmbossFilter	448	373	74	235	172	63
3	EdgeFilter	1368	432	936	1114	214	900
4	GrayscaleFilter	298	377	-79	124	170	-46
5	ContrastFilter	327	382	-55	181	191	-10
6	SharpenFilter	928	381	547	733	196	537
7	BlurFilter	1149	405	744	947	190	757
8	InvertFilter	327	382	-55	162	191	-29
9	TwirlFilter	1347	426	921	1172	202	970
10	SolarizeFilter	347	408	-62	155	213	-58
11	GammaFilter	377	370	7	194	191	3
12	BlockFilter	353	397	-44	153	189	-36
13	PosterizeFilter	363	373	-10	160	207	-47
14	DiffusionFilter	1202	463	739	881	273	608
15	MedianFilter	2240	552	1688	1961	195	1766
16	PinchFilter	1003	408	595	757	192	565
17	BumpFilter	1058	406	652	820	220	600

Table 20 Comparison of Offloading gains for the cloud

	Algorithm	**Local**	**Cloud Server**	**Gain of the measurement**	**Vertex cost**	**Edge cost**	**Gain of the model**
1	GaussianFilter	1467	599	869	1331	627	704
2	EmbossFilter	448	565	-118	236	578	-342
3	EdgeFilter	1368	624	745	1138	689	449
4	GrayscaleFilter	298	520	-223	123	572	-449
5	ContrastFilter	327	556	-228	156	628	-472
6	SharpenFilter	928	592	337	689	640	49
7	BlurFilter	1149	555	594	935	624	311
8	InvertFilter	327	560	-233	160	628	-468
9	TwirlFilter	1347	590	757	1135	654	481
10	SolarizeFilter	347	559	-212	148	687	-539
11	GammaFilter	377	560	-182	188	628	-440
12	BlockFilter	353	580	-227	158	624	-466
13	PosterizeFilter	363	571	-208	168	671	-503
14	DiffusionFilter	1202	674	528	900	807	93
15	MedianFilter	2240	733	1508	1932	636	1296
16	PinchFilter	1003	576	427	667	629	38
17	BumpFilter	1058	586	472	815	703	112

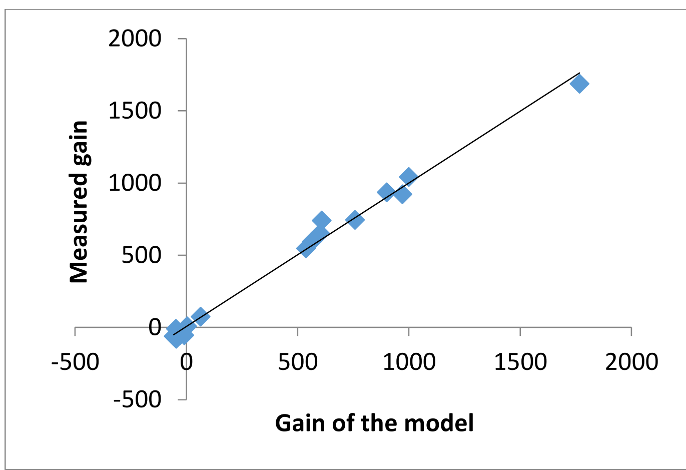


Figure 64 Scatter Plot of the gain of the model and the measured gain for LAN server

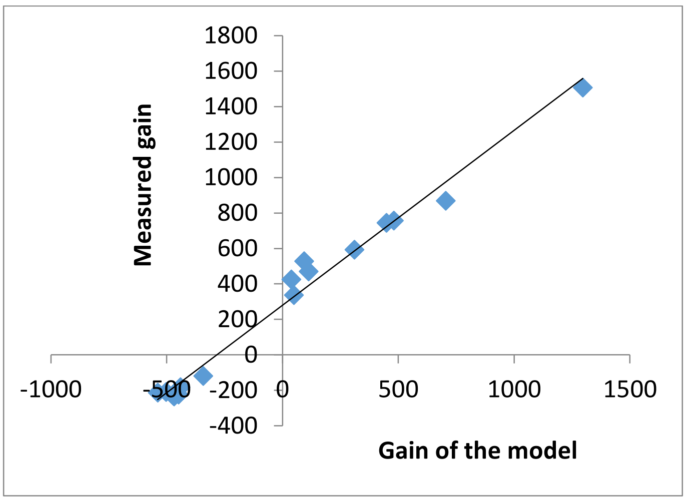


Figure 65 Scatter Plot of the gain of the model and the measured gain for the cloud

Although the primary focus of this experiment to observe the offloading gain according to the server location parameter, this experiment also facilitated the verification of the correctness of the call graph based model. Table 19 and Table 20 present the comparison of offloading gains which were calculated from the graph model and the measured data. Each algorithm was executed in the smartphone and in the remote server and the execution times were collected. If the execution time on the local client is extracted from the execution times on the remote server, it becomes the measured offloading gain. The positive value for the offloading gain means a benefit for smartphones. However, cases which have negative value are not suitable for offloading. As shown in the fifth and eighth column of Table 19 and Table 20, there is a consistency between the measured offloading gains and graph model' offloading gains. In addition, Figure 64 and Figure 65 present the scatter plots of the offloading gain of the measured and graph model. As seen in these graphs, the graph model reflects the application behavior. There is a significant positive relationship between the gain of the model and the measured gain for LAN server Pearson's $r = 0.991$, $p<.001$ and the cloud server Pearson's $r = 0.983$, $p<.001$. Thus, it is clear that our graph model reflects the application behavior successfully.

According to results of this experiment, Research Question 2 was answered, that is, a mobile application was profiled at runtime. The decision model of the framework dynamically

changed the classes to be offloaded at runtime according to the server location parameter and the optimal solution for the application was achieved.

6.5 Experiment 4

This section presents the experiment using the offloading framework to validate the behavior of the proposed call graph based model and to evaluate its effectiveness in terms of improving the performance (i.e. execution duration). The first goal of this experiment is to show whether offloading is productive or counterproductive. The offloading usually becomes counterproductive if application components which have higher data communication cost than processing cost are offloaded. Therefore, different combinations of the application classes were offloaded to find cases where offloading is counterproductive.

The second aim of this experiment is that does the call graph based model confirm the application behavior (measured data). As explained in Chapter 4, the execution times of the application at runtime were collected to construct a call graph. After that, the decision making algorithm was applied to calculate the offloading gain of each combination. The measured offloading gain and the graph model results were drawn in the scatter plots to investigate them. Correlation coefficient value was calculated for both of the data sets.

The following experimental procedures are common to all experiments. First, all object creations were converted to request the object from the offloading framework. Then, mobile applications were executed on the smartphone with various input values. The components which are eligible for offloading were presented to the software developer. The software developer can also monitor offloading results.

An OR application was chosen as a computation intensive smartphone application (Figure 67). In the OR application, there are two modes; learning and recognition. In the learning mode, the OR application takes a bitmap image as an input and computes the feature vector saved with the specified object and author name. The recognition mode compares the feature vector of the bitmap object with the stored feature vectors and returns the most related object information. In this study, the Eyedentify application [31] was used. The sequence diagram and a code snippet from the application are given in Figure 33 and Figure 35.

Table 21 Offloading cases for the OR application

	A = Image Compression, B = FeatureVectorServiceImp, C = MyWeibull, D = CxWeibullFit, E= FitWeibull , F = CxPatTask (✓ = instance of the classes are offloaded to the server)				
Cases/Classes	**A**	**B**	**C**	**D**	**E**	**F**
1	✓					
2		✓				
3			✓			
4	✓	✓				
5		✓	✓			
6		✓	✓			✓
7		✓	✓		✓	✓
8		✓	✓	✓	✓	✓
9	✓	✓	✓			
10	✓	✓	✓	✓	✓	
11	✓	✓	✓	✓	✓	✓
12	✓	✓	✓	✓		
13			✓	✓		
14			✓	✓	✓	
15			✓	✓	✓	✓
16				✓		
17				✓	✓	
18				✓	✓	✓
19					✓	
20					✓	✓
21						✓
22		✓		✓		
23		✓		✓	✓	
24		✓		✓	✓	✓
25		✓			✓	
26		✓			✓	✓
27		✓				✓
28		✓	✓	✓		
29		✓	✓		✓	
30		✓	✓	✓	✓	
31		✓	✓	✓		✓
32			✓	✓		✓

Table 21 presents the 32 combinations of application classes to be offloaded to the remote server to investigate offloading results for different conditions.

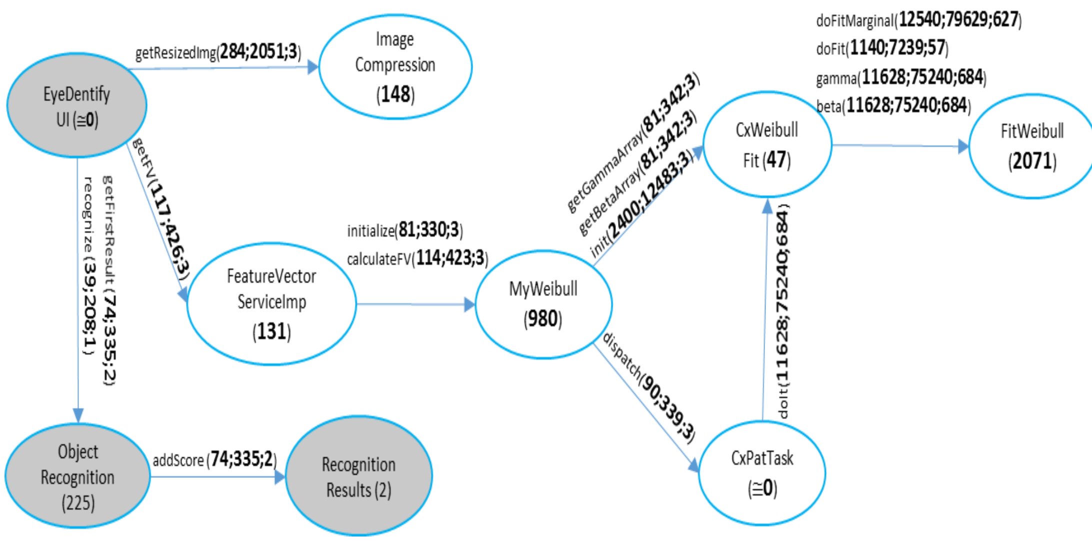


Figure 66 A graph representation of the OR application

Figure 66 shows the graph representation of the method call of the OR application (profiling results). The numbers following the edge name present the edge weight in the LAN server, cloud server and the frequency of the method call, respectively. The vertex and edge costs are also presented in this graph. The grey circles (classes) present the non-offloadable classes marked by the offloading framework. The FM heuristic was applied to this graph to find the best offloading gain for offloading. Table 22 presents the graph partitioning results. In addition, the optimal algorithm that checks all combinations to identify the optimal offloading gain was also implemented. However, if the number of classes (vertices) increases (for instance, to more than 15 classes) the optimal algorithm becomes costly. The optimal algorithm was only used for comparison to show the effectiveness of the FM heuristic.

Table 22 Graph partitioning results

Partitioning Algorithms	**Offloading Gains**	**Offloaded Classes**
Optimal Algorithm	3112	MyWeibull, FeatureVectorServiceImpl, CxWeibullFit, FitWeibull, CxPatTask
FM Heuristic	3112	MyWeibull, FeatureVectorServiceImpl, CxWeibullFit, FitWeibull, CxPatTask

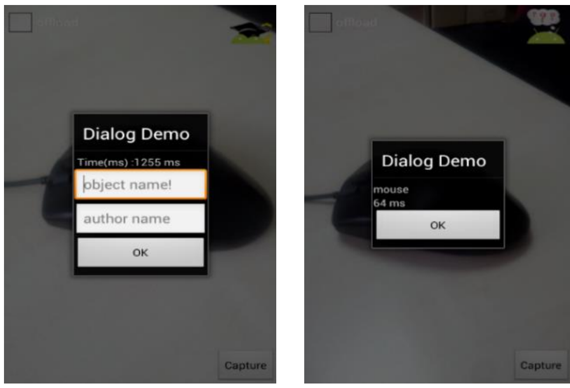


Figure 67 The screenshots of the OR application using our offloading framework

6.5.1 Results on the execution time (response time)

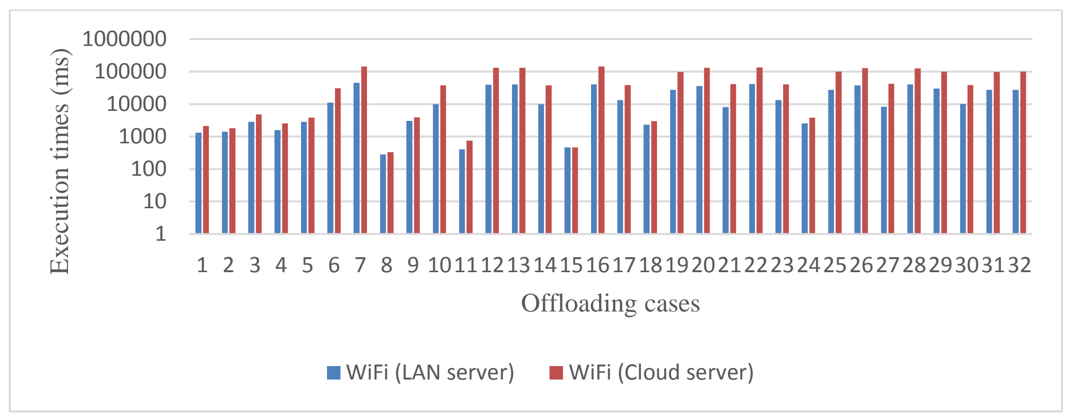


Figure 68 The offloading cases of the OR application

This application was executed 10 times using various inputs in the smartphone. Figure 68 presents response times (logarithmic scale) for the offloading cases given in Table 21. If the classes sent by the offloading framework to the server have higher communication costs compared to the processing costs, offloading becomes counterproductive. Therefore, these results also present the importance of finding the optimal solution. The FM heuristic identifies case 8 as the best solution for offloading (response time: 282 ms) and thus offloads the instances of classes of this case. In addition, executing the application locally on the smartphone takes 1131 ms (response time). To measure the overhead of the framework for profiling and solver, the application was run in profiling and solver mode, which takes 2057 ms. The framework runs in the solver mode if it finds any updates in history-based profiles. There are also significant differences between a nearby server and a cloud server in certain cases such as cases 6, 7 and 10. These differences result from the method call frequency of the objects located in the different side in Figure 66.

Table 23 Offloading gains for LAN server

Cases	**Offloading gain (Measured)**	**Offloading gain (Graph Model)**	**cases**	**Offloading gain (Measured)**	**Offloading gain (Graph Model)**
1	-198	-136	17	-12104	-12072
2	-263	-194	18	-1187	-444
3	-1712	-1743	19	-26035	-34865
4	-442	-317	20	-35251	-46583
5	-1718	-1697	21	-7002	-11718
6	-9852	-13196	22	-39644	-51260
7	-44049	-48061	23	-12248	-12253
8	849	3112	24	-1411	-625
9	-1882	-1794	25	-26261	-35046
10	-8702	-8625	26	-36208	-46764
11	722	2976	27	-7084	-11899
12	-38644	-47749	28	-39302	-47613
13	-39132	-47822	29	-28563	-36523
14	-8745	-8725	30	-9024	-8606
15	669	2903	31	-26312	-35895
16	-39354	-51079	32	-26398	-36104

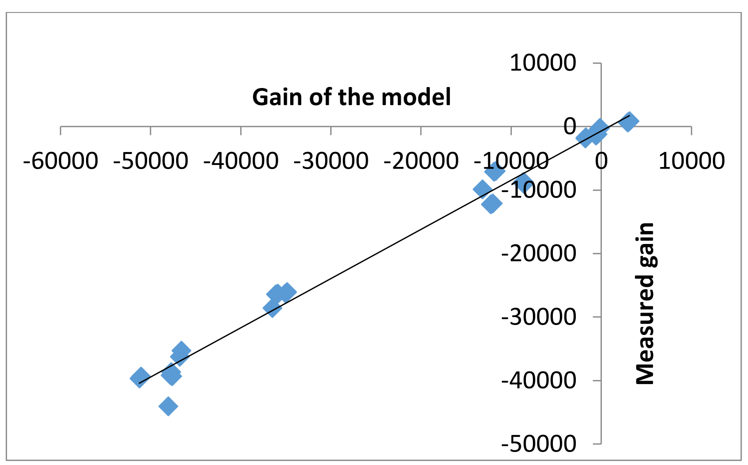


Figure 69 Scatter Plot of the gain of the model and the measured gain for LAN server

Table 24 Offloading gains for Cloud Server

Cases	Offloading gain (Measured)	Offloading gain (Graph Model)	cases	Offloading gain (Measured)	Offloading gain (Graph Model)
1	-998	-1903	17	-37725	-86289
2	-674	-1048	18	-1814	-11388
3	-3684	-12940	19	-96724	-235277
4	-1382	-2951	20	-131203	-310856
5	-2673	-12482	21	-39969	-75579
6	-29714	-87722	22	-131978	-326756
7	-141701	-322999	23	-38924	-87337
8	798	2803	24	-2677	-12436
9	-2814	-14724	25	-98713	-236325
10	-36691	-74679	26	-127775	-311904
11	387	900	27	-40784	-76627
12	-130023	-314098	28	-124658	-312195
13	-128684	-312653	29	-98621	-248098
14	-36421	-73234	30	-37104	-72776
15	668	2345	31	-97423	-236616
16	-142328	-325708	32	-97801	-237074

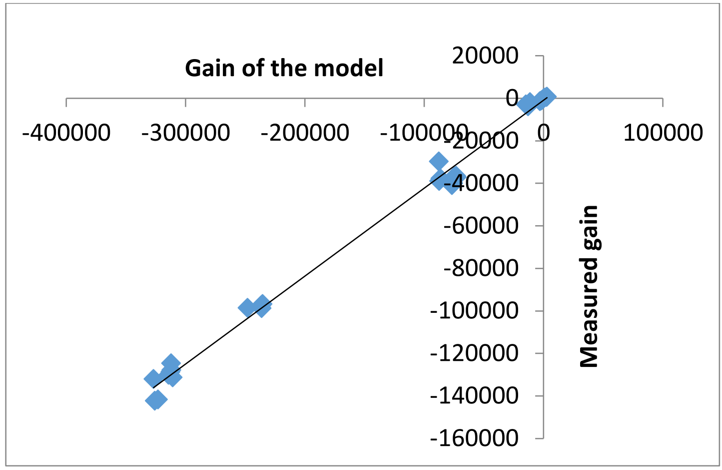


Figure 70 Scatter Plot of the gain of the model and the measured gain for the cloud

In order to validate the graph model, 32 combinations of classes as shown in Table 21 were offloaded. The overall execution time of each combination is measured and the offloading gain is calculated. The measured offloading gain is found by extracting the local execution time (1131 ms: executing application locally) from the overall execution time of each combination. Figure 69 and Figure 70 present the scatter plots of the offloading gain of the measured and graph model. Table 23 and Table 24 present offloading gains for a LAN server and the cloud. Figure 69 and Figure 70 indicate that the offloading gains of the measured and the graph model are correlated. Furthermore, There is a significant positive relationship between the gain of the model and the measured gain for LAN server Pearson's r = 0.987, p<.001 and the cloud server Pearson's r = 0.994, p<.001. Thus, it is clear that our graph model reflects the application behavior successfully.

The OR application experiments for changing the complexity of the OR algorithm were also carried out. In these experiments, case 11 was used. Using the OR algorithm, the images were converted to 64x48, 128x96 and 256x192 resolutions to calculate the feature vectors. Although results obtained from higher resolutions are more precise, the 256x192-resolution image causes an out-of-memory exception in the smartphone. Figure 71 and Figure 72 present the execution time for 480x800 resolution images taken by the smartphone camera. The execution times are reduced by 60% to 83%. Figure 72 shows the network transmission and the LAN server execution times for offloading. As a result, when complexity of the algorithms in an application increase, the resource of the smartphone may become insufficient and offloading can be inevitable.

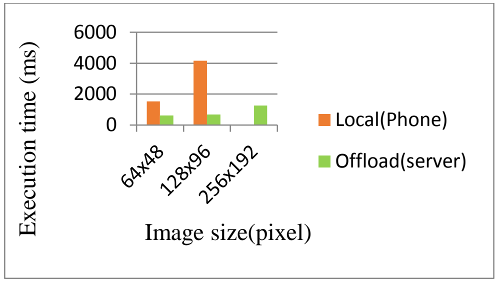


Figure 71 Execution time of the OR

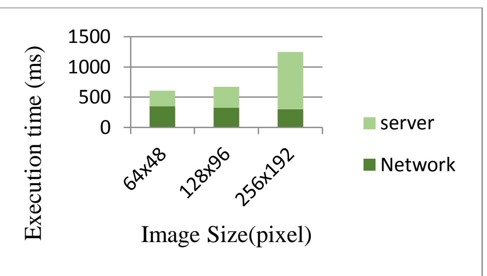


Figure 72 Execution time of offloading

According to results of this experiment, Research Question 3 was answered, that is, distribution transparency was completely achieved. The different offloading cases of an application were successfully migrated to the remote server, and especially the cases requiring callback were dynamically handled by the framework at runtime.

6.5.2 *Power consumption results*

PowerTutor [91] analysis tool was used to measure display, CPU, and Wi-Fi power consumption of the application. In the OR application, the energy consumptions for the first eleven cases given in Table 21 were measured.

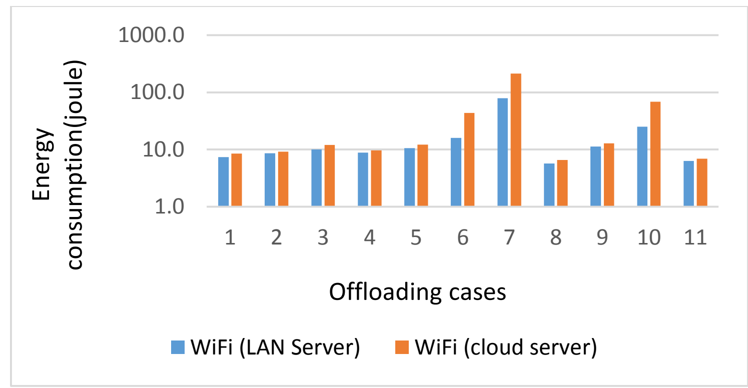


Figure 73 The energy consumption of the OR application cases

In order to observe the relation between execution time and energy consumption, this experiment was carried out. Figure 73 presents the energy consumption (logarithmic scale) of the first eleven cases given in Table 6. Case 8 is the optimal solution for application partitioning and gives the least energy consumption when compared with other cases. It is clear there is a direct correlation between the response times and energy consumption of the cases (the correlation coefficient is 0.99). The energy consumption of locally executing the application is 8.1 joule, which reaches 9.2 joules with the solver overhead. There are also significant differences due to the network cost between the nearby server and cloud server in cases 6, 7 and 10. The energy consumption of the cases 6, 7 and 10 are significantly higher than other cases because of the network cost resulted from objects resided in different machines. In addition, the energy model of the framework determined the case 8 as the best solution for offloading.

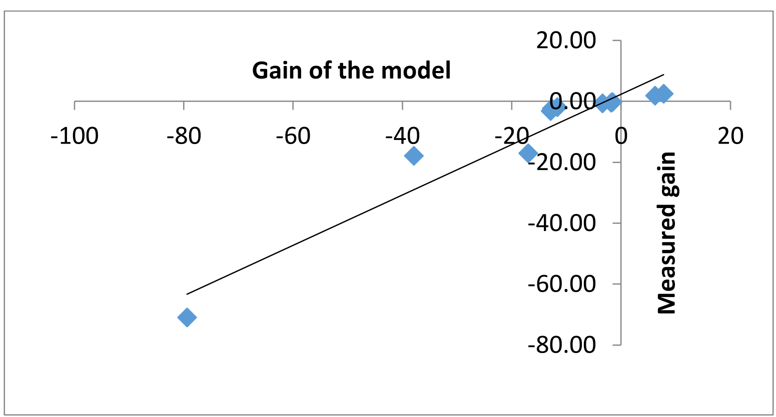


Figure 74 Scatter Plot of the gain of the model and the measured gain for LAN server

Figure 74 indicates that the measured gain of energy consumption and the gain of the energy model are correlated. Furthermore, There is a significant positive relationship between the gain of the energy model and the measured gain for LAN server Pearson's $r = 0.958$, $p<.001$. Thus, it is clear that the energy model reflects the application behavior successfully.

6.6 Performance Comparison of Frameworks

In order to compare the performance of our framework with existing approaches, Cuckoo [7] and Flores and Srirama [83] frameworks are downloaded from GitHub. For comparison of

frameworks, the object recognition application is implemented and run 10 times. Experiments are carried out by using the same hardware setup for all frameworks (Section 6.2). Based on 10 runs, the proposed framework turned out to be better in all runs as shown in Figure 75, and on average, it performed 11% better than the Cuckoo framework [7], and 9% better than that of Flores and Srirama's [83].

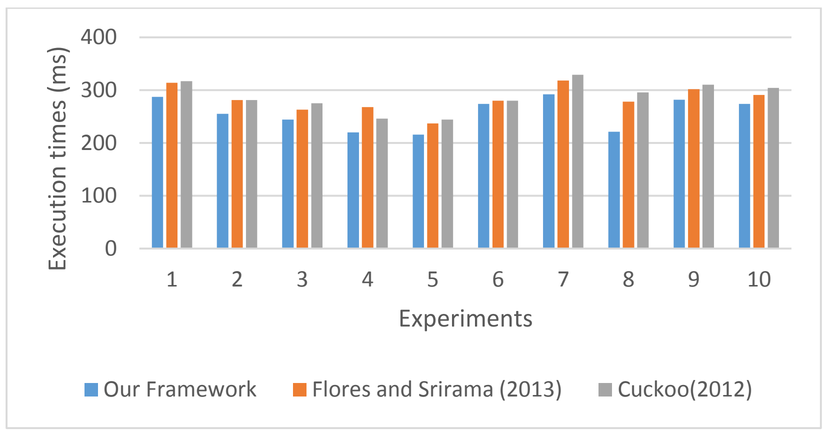


Figure 75 Comparison of the frameworks

6.7 Discussion on Roaming, SSL connection, Failure and Load on Server

The proposed framework handles user mobility during offloading of application components. Deploying components to be offloaded to the cloud server and the LAN server for remote computing are cases where user mobility is needed to be addressed. First, if the server location is a cloud server and a mobile user moves to a different location where he connects to the remote server with a new access point, at this point, the network connection costs will change and the graph model needs to be re-solved to adapt to the network cost change. The proposed framework dynamically adapts itself in case of attached network change. The mobility of the user is handled at the same way for LAN server. During the mobility, if the request or response messages are lost, the mobile client sends a FT_REQUEST to remote server to handle fault tolerance. In addition, the software developer may require a secure connection for specific components by providing SSL connection flag when requesting object creation. Since the SSL encrypts the messages, the data transmitted between mobile client and remote server increases. This situation also increases the cost of communication between components which reside in different machines and the graph model solution will be changed when comparing cases in which a SSL connection is not implemented. The other issue needs to be addressed is the load on the server. The cloud servers inherently provide scalability when the demand to computing services increases. The cloud server increases the server resources at runtime when more users request the services provided by the cloud. For the LAN server, the proposed framework can monitor the requests and provide a new server instance in case of an increase for requesting deployed services. Since the synthetic applications that were implemented for testing the developed algorithms present cases where edge costs randomly assigned based on vertices costs, they can also address the situations such as an SSL connection (an increase in data communication between objects) and roaming (mobile user mobility). However, the applications that specifically evaluate the situations where an SSL connection, roaming, failure and load on server are explicitly addressed; are left as future work.

CHAPTER 7

CONCLUSION

This thesis is concluded in this chapter by providing a summary of the research goals and realization of these goals. Furthermore, major implications of the developed framework on mobile application development, practical benefits and contributions are presented. Finally, the main limitations of this work and the direction of future work to improve certain aspects of the proposed framework are given.

7.1 Summary

The goal of this study were 1) to improve the performance of mobile applications through offloading their computation intensive components to resourceful servers 2) to automatically collect metrics for each component of the mobile application and construct a call graph at runtime 3) to dynamically partition the mobile application in order to find computation intensive components. The realization of these goals resulted in the proposed solutions facilitated in the framework. In particular, an offloading framework should have the ability to decide which location (local or remote) is beneficial for smartphones to execute the computation intensive components by answering how, what, where and when to offload with precision. These questions were considered during the design of the effective offloading framework and answered respectively.

Mobile devices will evolve to become the dominant computing devices and they will also provide computationally intensive applications with the help of mobile cloud computing. Since computation intensive applications generally require more memory and processing capacity, they can be adapted as a mobile application via migrating their computation intensive parts to a remote server. Thus, in order to overcome resource limitations of smartphones, client-server based delegation model and offloading mechanisms can be utilized to augment mobile devices. In the delegation model, pre-defined services consumed by mobile devices are implemented in the server. However, this approach not only decreases the flexibility of the application but also increases the overhead on software developers. On the other hand, most of the offloading mechanisms in the literature offer partial benefits for mobile applications in terms of development efforts and functionality. The main disadvantage of the current offloading mechanisms is that the callback functionality, which is essential for complete distribution transparency, is not taken into account in the frameworks. They offload computation intensive components which are annotated by software developers and should not depend on any resource residing in the smartphone side. Furthermore, to best of our knowledge, an effective model for determining computation intensive components dynamically has not been presented yet.

This framework is based on the Inversion of Control mechanism that delegates the object creation task to the offloading factory. Thus, the developer requests object creation from the offloading factory. The offloading factory then decides whether to create a proxy or an

object in the local client at runtime. The main advantages of the framework are; distribution transparency of offloading and the program structure not being changed by the developer.

Developing a distributed application via object mobility in object oriented programming has been extensively studied. Although Java RMI, OMG's CORBA and Microsoft DCOM are well structured frameworks, they are not explicitly targeted for mobile platforms. In addition, Android IDL and OSGi-service based mechanisms have faced problems such as the difficulty of callback implementations, dependency on specific platforms and argument inconsistency. In these approaches, application components such as methods or classes need to be annotated by a software developer and then the frameworks convert these components to proper services which can be migrated to a remote server. Since these services should not be dependent on the other components, complexity and overhead over the software developer increases. On the other hand, this study proposes a lightweight framework in which software developers only need to deal with object creation. All other details including remote object creation, remote method call, parameter passing and communication between devices are the responsibility of the proposed offloading framework, and they are not even visible to the software developers. In addition, since the same offloading framework is implemented on both the mobile device and the cloud, when the cloud needs to call back mobile device resources, the offloading factory creates the reverse proxy that automatically handles the callback functionality. Furthermore, delegating object creation to a factory method which provides flexibility at runtime by giving a local or a proxy object, which enables smartphones to migrate method call through these proxies and utilize cloud resources transparently.

The most important aspect of an offloading framework is to decide what to offload in order to obtain a benefit. The components that require higher processing cost than communication cost are determined. First, as opposed to many of existing works, the execution times of the components and their dependencies are obtained at runtime and represented as a call graph. A novel call graph based model is proposed in this study. Constructing this call graph, the offloading decision is converted to a graph partitioning problem. Hence, a well-known graph partitioning (min-cut) algorithm (FM heuristic) is implemented to make an offloading decision at runtime. Moreover, the inclusion of a new heuristic algorithm is achieved through a modular structure of the proposed framework. This algorithm identifies the most productive offloading decision according to the proposed call graph based model. Different combinations of application components are offloaded to demonstrate the importance of the offloading decision. Code offloading can be counterproductive if sending classes to remote servers has a higher network cost than processing them on the mobile client.

The offloading location (a nearby server or a cloud) is also an important decision which needs to be handled. Although cloud servers provide scalability and on-demand computing, as presented in Section 2.5, high WAN latencies have a negative effect on the offloading mechanism. Therefore, a nearby server in vicinity of smartphones should be preferred. Moreover, existing studies except Verbelen et al. [9] did not consider the LAN server or service discovery. The proposed framework implemented a DNS-SD based service discovery to find nearby servers at first rather than finding a cloud server. Where to offload determines the benefit smartphones receive. In addition, when to offload is determined according to the network bandwidth. In implementation of the framework, determining network bandwidth is handled during offloading on-the-fly and the mobile application adapts itself according to network bandwidth changes. Furthermore, the framework security is handled via providing SSL connection. An authentication and single sign-on mechanism were implemented by using Twitter OAuth mechanism.

Experimental results show that the proposed graph model fits well to the application partitioning problem. The gains of the graph model and the measured gains are strongly correlated, that is, the graph model reflects the application behavior successfully. According to the results of several experiments, offloading reduced the execution time by 1% to 83% and decreased power consumption by 65% to 88%. More importantly, the proposed

framework did not incur significant overhead. Smartphones obtain more benefits by offloading to a nearby server rather than the cloud server. The complete distribution transparency achieved in the framework enables smartphones to utilize server resources as part of them.

7.2 Contributions

The key contributions of this study are summarized as:

- The framework eases the burden on the developer by transparently handling the distribution of the application components. The programming model allows the objects of a mobile application to be distributed, and the classes to be offloaded are identified on-the-fly by the framework.
- The offloading programming model dynamically creates local or proxy objects on both sides (smartphone or server side) according to the result of a call graph based decision model. This allows simultaneous object access capability for both directions and contributes to the transparency of the object distribution.
- The framework provides a globally optimal partitioning using a call graph based decision model in order to identify the set of classes to be offloaded, thereby producing the optimum benefit under current runtime context. The framework utilizes a heuristic graph partitioning algorithm for this purpose.
- The framework constantly updates the call graph according to changing conditions as the application is running, hence allowing dynamic adaptation. That is, as the conditions associated with the network connectivity change, the framework recalculates the offloading decisions according to the updated call graph model on the server side, at runtime.
- The framework has been evaluated and compared to other frameworks through real application scenarios. In the experiments, measurements are made on real smartphones and servers running practical applications. Moreover, a significant correlation is observed between the offloading gain measured and the offloading gain suggested by the graph based model.
- The framework enables smartphones to implement complex libraries that are not available for smartphones.
- The framework achieves fault tolerance by providing logging and recovery mechanism.

7.3 Limitations and Future Work

The proposed offloading framework has certain limitations. During the application development phase, the software developer should implement the get and set method of public global class variables. In the framework, software developers need to use the offloading factory to create objects. By a VM code modification, the object creation code snippet presented by a "new" keyword can be detected and this code snippet can be modified to delegate the object creation to the offloading factory automatically. The AspectJ library, which is an aspect-oriented programming extension, can be adapted for smartphones to enable automatic code modification to create objects.

The computation intensive tasks should not be implemented in user interface classes since these classes are all marked as non-offloadable components. However, the computation intensive methods in the user interface can be converted to a method of a new class and the required code modification should be made in the user interfaces. The edge cost conversion equations of the graph model were given for the specific smartphone, LAN server and cloud virtual machine. These equations can be provided for several smartphones and different server configurations by software developers, and can be saved to a cloud repository.

REFERENCES

[1] ITU, “Measuring the information society report 2014,” Geneva, Switzerland, 2014.

[2] S. Ou, K. Yang, and J. Zhang, “An effective offloading middleware for pervasive services on mobile devices,” *Pervasive Mob. Comput.*, vol. 3, no. 4, pp. 362–385, 2007.

[3] Oracle, “Java remote method invocation API,” 2010. [Online]. Available: http://docs.oracle.com/javase/7/docs/technotes/guides/rmi/. [Accessed: 15-Nov-2015].

[4] Android, “Android interface definition language,” 2008. [Online]. Available: http://developer.android.com/guide/components/aidl.html. [Accessed: 15-Nov-2015].

[5] OSGi, “OSGi Architecture,” 2010. [Online]. Available: https://www.osgi.org/developer/architecture/. [Accessed: 15-Nov-2015].

[6] T. Verbelen, P. Simoens, F. De Turck, and B. Dhoedt, “AIOLOS: Middleware for improving mobile application performance through cyber foraging,” *J. Syst. Softw.*, vol. 85, no. 11, pp. 2629–2639, 2012.

[7] R. Kemp, N. Palmer, T. Kielmann, and H. Bal, “Cuckoo: a computation offloading framework for smartphones,” in *Mobile Computing, Applications, and Services*, Springer Berlin Heidelberg, 2012, pp. 59–79.

[8] F. Plášil and M. Stal, “An architectural view of distributed objects and components in CORBA, Java RMI and COM/DCOM,” *Softw. - Concepts Tools*, vol. 19, no. 1, pp. 14–28, 1998.

[9] P. Chung, Y. Huang, S. Yajnik, D. Liang, and J. Shih, “DCOM and CORBA side by side, step by step, and layer by layer,” *C++ Rep.*, vol. 10, no. 1, pp. 18–29, 1998.

[10] B. Chun, S. Ihm, and P. Maniatis, “Clonecloud: elastic execution between mobile device and cloud,” in *EuroSys ’11 Proceedings of the sixth conference on Computer system*, 2011, pp. 301–314.

[11] E. Cuervo, A. Balasubramanian, D. Cho, A. Wolman, S. Saroiu, R. Chandra, and P. Bahl, “MAUI: making smartphones last longer with code offload,” in *Proceedings of the 8th international conference on Mobile systems, applications, and services*, 2010, vol. 17, pp. 49–62.

[12] M. D. Kristensen and N. O. Bouvin, “Scheduling and development support in the Scavenger cyber foraging system,” *Pervasive Mob. Comput.*, vol. 6, no. 6, pp. 677–692, 2010.

[13] E. Chen, S. Ogata, and K. Horikawa, "Offloading Android applications to the cloud without customizing Android," in *2012 IEEE International Conference on Pervasive Computing and Communications Workshops, PERCOM Workshops 2012*, 2012, pp. 788–793.

[14] H. Flores, P. Hui, S. Tarkoma, Y. Li, S. Srirama, and R. Buyya, "Mobile code offloading: from concept to practice and beyond," *IEEE Commun. Mag.*, vol. 53, no. 3, pp. 80–88, 2015.

[15] M. Satyanarayanan, P. Bahl, R. Cáceres, and N. Davies, "The case for VM-based cloudlets in mobile computing," *IEEE Pervasive Comput.*, vol. 8, no. 4, pp. 14–23, 2009.

[16] M. Fowler, "Inversion of control containers and the dependency injection pattern," 2004. [Online]. Available: http://www.martinfowler.com/articles/injection.html. [Accessed: 15-Nov-2015].

[17] M. Kaya, A. Kocyigit, and P. E. Eren, "A mobile computing framework based on adaptive mobile code offloading," in *Software Engineering and Advanced Applications (SEAA), 2014 40th EUROMICRO Conference on*, 2014, pp. 479–482.

[18] K. Kumar, J. Liu, Y. H. Lu, and B. Bhargava, "A survey of computation offloading for mobile systems," *Mob. Networks Appl.*, vol. 18, no. 1, pp. 129–140, 2013.

[19] M. Shiraz, M. Sookhak, A. Gani, and S. A. A. Shah, "A study on the critical analysis of computational offloading frameworks for mobile cloud computing," *J. Netw. Comput. Appl.*, vol. 47, pp. 47–60, 2015.

[20] G. Karypis and V. Kumar, "A fast and high quality multilevel scheme for partitioning irregular graphs," *SIAM J. Sci. Comput.*, vol. 20, no. 1, pp. 359–392, 1998.

[21] B. Hendrickson and R. Leland, "A multi-level algorithm for partitioning graphs," in *Proceedings of the IEEE/ACM SC95 Conference*, 1995, pp. 1–14.

[22] B. W. Kernighan and S. Lin, "An efficient heuristic procedure for partitioning graphs," *Bell Syst. Tech. J.*, vol. 49, no. 2, pp. 291–307, 1970.

[23] C. M. Fiduccia and R. M. Mattheyses, "A linear-time heuristic for improving network partitions," in *Design Automation, 1982. 19th Conference on*, 1982, pp. 175–181.

[24] F. Zhu, M. W. Mutka, and L. M. Ni, "Service discovery in pervasive computing environments," *Pervasive Comput. IEEE*, vol. 4, no. 4, pp. 81–90, 2005.

[25] DNS-SD, "DNS Service Discovery (DNS-SD) protocol," 2013. [Online]. Available: http://www.dns-sd.org/. [Accessed: 15-Nov-2015].

[26] W. K. Edwards, "Discovery Systems in Ubiquitous Computing Ubiquitous," *Pervasive Comput.*, vol. 5, no. 2, pp. 70–77, 2006.

[27] Oracle, "Secure Sockets Layer (SSL) protocol," 2010. [Online]. Available: https://docs.oracle.com/cd/E19509-01/820-3503/6nf1il6ek/index.html. [Accessed: 15-Nov-2015].

[28] S. Fahl, M. Harbach, T. Muders, M. Smith, L. Baumgärtner, and B. Freisleben, "Why Eve and Mallory Love Android: An Analysis of Android SSL (In)Security Categories

and Subject Descriptors," in *ACM Conference on Computer and Communication Security (CCS)*, 2012, pp. 50–61.

[29] Twitter Oauth, "OAuth mechanism," 2014. [Online]. Available: https://dev.twitter.com/oauth. [Accessed: 15-Nov-2015].

[30] D. Hardt, "The OAuth 2.0 Authorization Framework," 2012.

[31] R. Kemp, N. Palmer, T. Kielmann, F. Seinstra, N. Drost, J. Maassen, and H. Bal, "eyeDentify: Multimedia cyber foraging from a smartphone," in *ISM 2009 - 11th IEEE International Symposium on Multimedia*, 2009, pp. 392–399.

[32] H. Gani and C. Ryan, "Improving the transparency of proxy injection in Java," in *Thirty-Second Australasian Conference on Computer Science*, 2009, pp. 55–64.

[33] M. Philippsen and M. Zenger, "JavaParty - transparent remote objects in Java," *Concurr. Pract. Exp.*, vol. 9, no. 11, pp. 1225–1242, 1997.

[34] S. Yang, D. Kwon, H. Yi, Y. Cho, Y. Kwon, and Y. Paek, "Techniques to Minimize State Transfer Costs for Dynamic Execution Offloading in Mobile Cloud Computing," *IEEE Trans. Mob. Comput.*, vol. 13, no. 11, pp. 1–1, 2014.

[35] N. Geoffray, G. Thomas, and B. Folliot, "Transparent and dynamic code offloading for java applications," in *On the Move to Meaningful Internet Systems 2006: CoopIS, DOA, GADA, and ODBASE*, Springer Berlin Heidelberg, 2006, pp. 1790–1806.

[36] J. Liu, E. Ahmed, M. Shiraz, A. Gani, R. Buyya, and A. Qureshi, "Application partitioning algorithms in mobile cloud computing: Taxonomy, review and future directions," *J. Netw. Comput. Appl.*, vol. 48, pp. 99–117, 2015.

[37] E. Abebe and C. Ryan, "Adaptive application offloading using distributed abstract class graphs in mobile environments," *J. Syst. Softw.*, vol. 85, no. 12, pp. 2755–2769, 2012.

[38] Android, "Android SDK," 2014. [Online]. Available: https://source.android.com/devices/tech/dalvik/. [Accessed: 15-Nov-2015].

[39] T. Verbelen, T. Stevens, P. Simoens, F. De Turck, and B. Dhoedt, "Dynamic deployment and quality adaptation for mobile augmented reality applications," *J. Syst. Softw.*, vol. 84, no. 11, pp. 1871–1882, 2011.

[40] T. Verbelen, P. Simoens, F. De Turck, and B. Dhoedt, "Adaptive deployment and configuration for mobile augmented reality in the cloudlet," *J. Netw. Comput. Appl.*, vol. 41, pp. 206–216, 2014.

[41] D. Kovachev and R. Klamma, "Framework for computation offloading in mobile cloud computing," *Int. J. Interact. Multimed. Artif. Intell.*, vol. 1, no. 7, pp. 6–15, 2012.

[42] Y. Zhang, G. Huang, X. Liu, W. Zhang, H. Mei, and S. Yang, "Refactoring android Java code for on-demand computation offloading," *ACM SIGPLAN Not.*, vol. 47, no. 10, p. 233, 2012.

[43] H. Y. Chen, Y. H. Lin, and C. M. Cheng, "COCA: Computation offload to clouds using AOP," in *Proceedings - 12th IEEE/ACM International Symposium on Cluster,*

Cloud and Grid Computing, CCGrid 2012, 2012, pp. 466–473.

[44] C. Ling, C. Ming, W. Zhang, and F. Tian, "AR Cloudlets for Mobile Computing," *Int. J. Digit. Content Technol. its Appl.*, vol. 5, no. 12, pp. 162–169, 2011.

[45] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design patterns: Elements of reusable object-oriented software*. Upper Saddle River, NJ: Pearson Education, 1998.

[46] P. Eugster, "Uniform proxies for Java," *ACM SIGPLAN Not.*, vol. 41, no. 10, pp. 139–152, 2006.

[47] Oracle, "Dynamic proxy classes," 2010. [Online]. Available: https://docs.oracle.com/javase/8/docs/technotes/guides/reflection/proxy.html. [Accessed: 12-Nov-2015].

[48] C. G. Lasater, *Design Patterns*. 2010.

[49] E. Tilevich and Y. Smaragdakis, "J-orchestra: Automatic java application partitioning," in *ECOOP '02 Proceedings of the 16th European Conference on Object-Oriented Programming*, 2002, pp. 178–204.

[50] R. Caspar and W. Christopher, "Application adaptation through transparent and portable object mobility in Java," in *On the Move to Meaningful Internet Systems 2004: CoopIS, DOA, and ODBASE*, Springer Berlin Heidelberg, 2004, pp. 1262–1284.

[51] J. S. Rellermeyer, G. Alonso, and T. Roscoe, "R-OSGi: Distributed Applications through Software Modularization," in *Proceedings of the 8th International Conference on Middleware (Middleware'07)*, 2007, vol. 4834, pp. 1–20.

[52] J. S. Rellermeyer, O. Riva, and G. Alonso, "AlfredO: an architecture for flexible interaction with electronic devices," in *Middleware '08 Proceedings of the 9th ACM/IFIP/USENIX International Conference on Middleware*, 2008, pp. 22–41.

[53] I. Giurgiu, O. Riva, D. Juric, I. Krivulev, and G. Alonso, "Calling the cloud: Enabling mobile phones as interfaces to cloud applications," in *ACM/IFIP/USENIX 10th international conference on Middleware*, 2009, pp. 83–102.

[54] D. C. Schmidt, "Android AIDL," 2013. [Online]. Available: http://www.dre.vanderbilt.edu/~schmidt/cs282/ServicesAndIPC.pdf. [Accessed: 25-Nov-2013].

[55] P.-O. Fjallstrom, "Algorithms for graph partitioning: A survey," *Linkoping Electron. Artic. Comput. Inf. Sci.*, vol. 3, no. 10, pp. 1–34, 1998.

[56] X. Gu, K. Nahrstedt, A. Messer, I. Greenberg, and D. Milojicic, "Adaptive offloading for pervasive computing," *IEEE Pervasive Comput.*, vol. 3, no. 3, pp. 66–73, 2004.

[57] K. Yang, S. Ou, and H. H. Chen, "On effective offloading services for resource-constrained mobile devices running heavier mobile internet applications," *IEEE Commun. Mag.*, vol. 46, no. 1, pp. 56–63, 2008.

[58] M. Stoer and F. Wagner, "A Simple Min-Cut Algorithm," *J. ACM*, vol. 44, no. 4, pp. 585–591, 1997.

[59] H. Flores and S. N. Srirama, “Mobile Cloud Middleware,” *J. Syst. Softw.*, vol. 92, pp. 82–94, 2014.

[60] J. H. Christensen, “Using RESTful web-services and cloud computing to create next generation mobile applications,” in *Proceedings of the 24th ACM SIGPLAN conference companion on Object oriented programming systems languages and applications*, 2009, pp. 627–634.

[61] M. Kaya, M. Özpınar, Y. Çetin Kaya, and T. Taşkaya Temizel, “MobileMETU: A Mobile Campus project based on web services,” *Glob. J. Technol.*, vol. 3, pp. 1620–1625, 2013.

[62] Shuai Zhang, Xuebin Chen, Shufen Zhang, and Xiuzhen Huo, “The comparison between cloud computing and grid computing,” in *International Conference on Computer Application and System Modeling (ICCASM 2010)*, 2010, vol. 11, no. 46, pp. 72–75.

[63] Y. Wei and M. B. Blake, “Service-oriented computing and cloud computing: Challenges and opportunities,” *IEEE Internet Comput.*, vol. 14, no. 6, pp. 72–75, 2010.

[64] I. Foster, Y. Zhao, I. Raicu, and S. Lu, “Cloud computing and grid computing 360-degree compared,” in *Grid Computing Environments Workshop 2008 (GCE ’08)*, 2008, pp. 1–10.

[65] M. Malathi, “Cloud computing concepts,” in *ICECT 2011 - 2011 3rd International Conference on Electronics Computer Technology*, 2011, vol. 6, pp. 236–239.

[66] R. Buyya, C. S. Yeo, S. Venugopal, J. Broberg, and I. Brandic, “Cloud computing and emerging IT platforms: Vision, hype, and reality for delivering computing as the 5th utility,” *Futur. Gener. Comput. Syst.*, vol. 25, no. 6, pp. 599–616, 2009.

[67] T. Dillon, C. Wu, and E. Chang, “Cloud Computing: Issues and Challenges,” in *2010 24th IEEE International Conference on Advanced Information Networking and Applications*, 2010, pp. 27–33.

[68] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. H. Katz, A. Konwinski, G. Lee, D. A. Patterson, A. Rabkin, I. Stoica, and M. Zaharia, “Above the clouds: A Berkeley view of cloud computing,” 2009.

[69] J. Varia, “Best practices in architecting cloud applications in the AWS cloud,” in *Cloud Computing: Principles and Paradigms*, Hoboken, New Jersey: John Wiley & Sons, Inc, 2011, pp. 459–490.

[70] Google, “Google app engine,” 2014. [Online]. Available: http://appengine.google.com. [Accessed: 25-May-2014].

[71] Microsoft, “Microsoft windows azure,” 2014. [Online]. Available: https://azure.microsoft.com/. [Accessed: 20-May-2014].

[72] S. Clinch, J. Harkes, A. Friday, N. Davies, and M. Satyanarayanan, “How close is close enough? Understanding the role of cloudlets in supporting display appropriation by mobile users,” in *2012 IEEE International Conference on Pervasive Computing and Communications, PerCom 2012*, 2012, pp. 122–127.

[73] R. Gupta, S. Talwar, and D. P. Agrawal, "Jini home networking: a step toward pervasive computing," *Computer (Long. Beach. Calif).*, vol. 35, no. 8, pp. 34–40, 2002.

[74] P. Dobrev, D. Famolari, C. Kurzke, and B. a. Miller, "Device and service discovery in home networks with OSGi," *IEEE Commun. Mag.*, vol. 40, no. 8, pp. 86–93, 2002.

[75] G. Moritz, C. Cornelius, F. Golatowski, D. Timmermann, and R. Stoll, "Differences and Commonalities of Service-Oriented Device Architectures, Wireless Sensor Networks and Networks-On-Chip," in *Proceedings, 4th International IEEE Workshop on Service Oriented Architectures in Converging Networked Environments (SOCNE2009)*, 2009, pp. 482–487.

[76] M. Giordano, "DNS-Based discovery system in service oriented programming," in *Advances in Grid Computing-EGC*, Springer Berlin Heidelberg, 2005, pp. 840–850.

[77] S. Cheshire and M. Krochmal, "DNS-Based Service Discovery," 2013.

[78] C. J. Lamprecht and A. P. A. van Moorsel, "Runtime Security Adaptation Using Adaptive SSL," in *4th IEEE Pacific Rim International Symposium on Dependable Computing*, 2008, pp. 305–312.

[79] S. Pai, Y. Sharma, S. Kumar, R. M. Pai, and S. Singh, "Formal verification of OAuth 2.0 using alloy framework," in *Proceedings - 2011 International Conference on Communication Systems and Network Technologies, CSNT 2011*, 2011, pp. 655–659.

[80] J. Niu, W. Song, and M. Atiquzzaman, "Bandwidth-adaptive partitioning for distributed execution optimization of mobile applications," *J. Netw. Comput. Appl.*, vol. 37, pp. 334–347, 2014.

[81] R. Martin and S. Totten, "Introduction to fault tolerant CORBA," 2003. [Online]. Available: http://sett.ociweb.com/cnb/CORBANewsBrief-200301.html. [Accessed: 15-Nov-2015].

[82] A. Gokhale, B. Natarajan, D. C. Schmidt, and K. C. Cross, "Towards real-time fault-tolerant CORBA middleware," *Cluster Comput.*, vol. 7, no. 4, pp. 331–346, 2004.

[83] H. Flores and S. Srirama, "Adaptive code offloading for mobile cloud applications: exploiting fuzzy sets and evidence-based learning," in *Proceeding of the fourth ACM workshop on Mobile cloud computing and services*, 2013, pp. 9–16.

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

[85] B. H. C. Cheng, R. de Lemos, H. Giese, P. Inverardi, J. Magee, J. Andersson, B. Becker, N. Bencomo, Y. Brun, B. Cukic, G. di Marzo Serugendo, S. Dustdar, A. Finkelstein, C. Gacek, K. Geihs, V. Grassi, G. Karsai, H. M. Kienle, J. Kramer, M. Litoiu, S. Malek, R. Mirandola, H. A. Muller, S. Park, M. Shaw, M. Tichy, M. Tivoli, D. Weyns, and J. Whittle, "Software engineering for self-adaptive systems: a research roadmap," in *Software Engineering for Self-Adaptive Systems*, vol. 5525, Springer Berlin Heidelberg, 2009, pp. 1–26.

[86] M. Ayad, T. Mohamed, and S. Ashraf, "Mobile GPU Cloud Computing with real

time application," in *Energy Aware Computing Systems & Applications (ICEAC), 2015 International Conference on. IEEE*, 2015, pp. 1–4.

[87] J. Hauswald, M. Thomas, Z. Qiang, D. Ronald, C. Chaitali, and M. Trevor, "A hybrid approach to offloading mobile image classification," in *In Acoustics, Speech and Signal Processing (ICASSP), 2014 IEEE International Conference on*, 2014, pp. 8375–8379.

[88] R. Smith, "Tesseract OCR," 2006. [Online]. Available: https://code.google.com/p/tesseract-ocr/. [Accessed: 15-Nov-2015].

[89] L. Corral, A. B. Georgiev, A. Sillitti, and G. Succi, "Can execution time describe accurately the energy consumption of mobile apps? an experiment in Android," in *Proceedings of the 3rd International Workshop on Green and Sustainable Software - GREENS 2014*, 2014, pp. 31–37.

[90] A. Rice and S. Hay, "Measuring mobile phone energy consumption for 802.11 wireless networking," *Pervasive Mob. Comput.*, vol. 6, no. 6, pp. 593–606, 2010.

[91] L. Zhang, B. Tiwana, R. P. Dick, Z. Qian, Z. M. Mao, Z. Wang, and L. Yang, "Accurate online power estimation and automatic battery behavior based power model generation for smartphones," in *Hardware/Software Codesign and System Synthesis (CODES+ISSS), 2010 IEEE/ACM/IFIP International Conference on*, 2010, pp. 105–114.

APPENDICIES

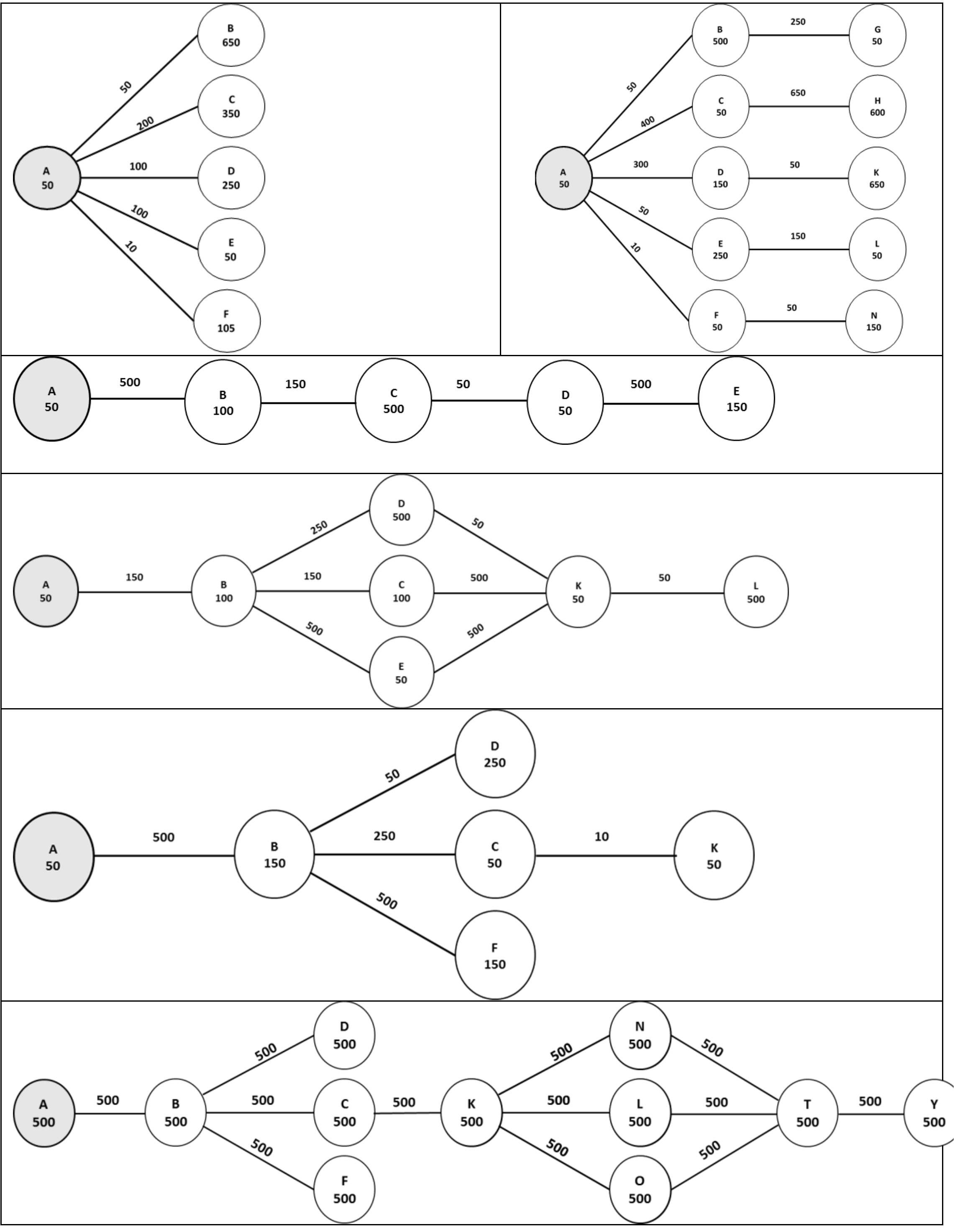
APPENDIX A:

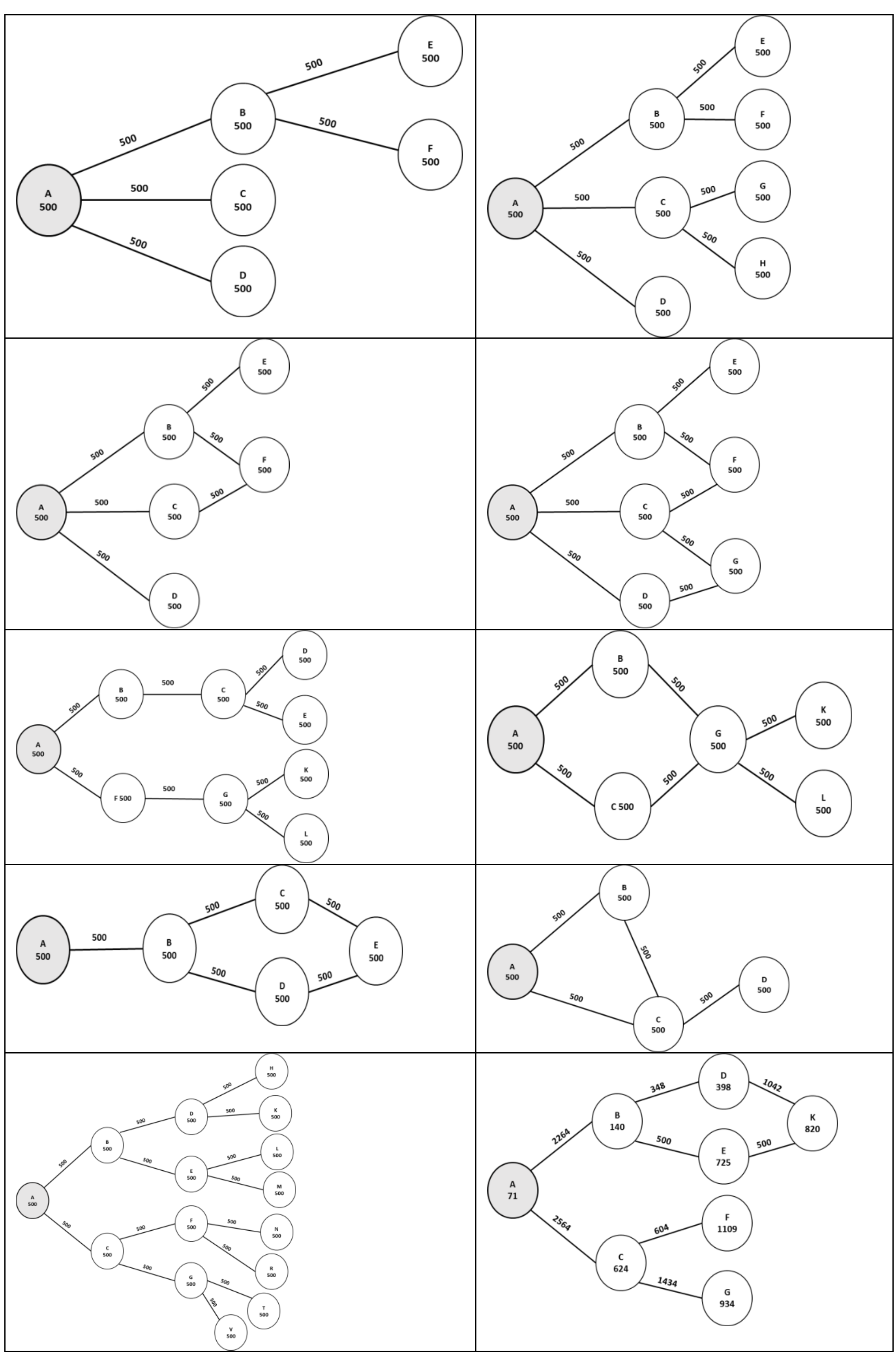
```
KL HEURISTIC
  determine a balanced initial partition of the nodes into sets A and B
   Set<Vertex> A ; Set<Vertex> B
   Set<Vertex> UnswappedA ; Set<Vertex> UnswappedB
        Until  Max Gain > 0
          compute D values for all a in A and b in B
            Until  |V|/2   vertices are traversed (UnswappedB)
              find a from A and b from B, such that g = D[a] + D[b] - 2*E(a, b) is maximal
              swap (a, A , B, b)
              remove a and b from UnswappedB and UnswappedA
              update D values for the elements of A = A \ a and B = B \ b
            End Until
          find k which maximizes g_max, the sum of g[1],...,g[k]
          if (g_max > 0) then
          Swap av[1],av[2],...,av[k] with bv[1],bv[2],...,bv[k]
    End Until
```

Pseudo code of KL algorithm is presented in this part. A graph is separated in two partition according to edge weights. This algorithm provides min-cut of the graph.

APPENDIX B:

Application topologies which were implemented to verify the offloading framework are presented in this section.




A 500
500
B 500
500
C 500
500
D 500
500
E 500
500
F 500
A 500
500
B 500
500
E 500
500
F 500
500
C 500
500
G 500
500
H 500
500
D 500
A 500
500
B 500
500
E 500
500
F 500
500
C 500
500
500
D 500
A 500
500
B 500
500
E 500
500
F 500
500
C 500
500
500
G 500
500
D 500
500
A 500
500
B 500
500
C 500
500
D 500
500
E 500
500
F 500
500
G 500
500
K 500
500
L 500
A 500
500
B 500
500
G 500
500
K 500
500
L 500
500
C 500
500
A 500
500
B 500
500
C 500
500
E 500
500
D 500
500
A 500
500
B 500
500
C 500
500
D 500
500
A 500
500
B 500
500
D 500
500
H 500
500
K 500
500
E 500
500
L 500
500
M 500
500
C 500
500
F 500
500
N 500
500
R 500
500
G 500
500
T 500
500
V 500
A 71
2264
B 140
348
D 398
1042
K 820
500
E 725
500
2564
C 624
604
F 1109
1434
G 934

CIRCULUM VITAE

MAHİR KAYA

Information Systems Department

Middle East Technical University

Informatics Institute, Universiteler Mahallesi, Dumlupınar Bulvarı, No:1, 06800, Ankara,Turkey

TEL: +90 312 210 77002 FAX: +90 312 210 3745

EMAIL: mahirkaya@gmail.com

AREAS OF INTEREST

- Code offloading
- Mobile cloud computing
- Service oriented architecture and web services
- Computer and communications networks

EDUCATION

Middle East Technical University, Ankara, Turkey

Doctor of Philosophy in Information Systems

2010-2016

Thesis Title: An Optimal Application Partitioning and Computational Offloading Framework for Mobile Cloud Computing

Advisor: Assoc. Prof. Dr. Altan Koçyiğit

Middle East Technical University, Ankara, Turkey

Master of Science in Information Systems

2007-2010

Thesis Title: E-Cosmic: A Business Process Model Based Functional Size Estimation Approach

Advisor: Prof. Dr. Onur Demirörs

Istanbul Technical University (ITU), Istanbul, Turkey

Bachelor of Science in Industrial Engineering

1995-2000

TEACHING ASSISTANTSHIPS

- Object Oriented Analysis and Design, Assistant
- Software Architecture, Assistant
- Software Design Patterns, Assistant
- Introduction to Software Engineering, Assistant
- Security Engineering, Assistant

PAPERS

- **Kaya, Mahir**, and Altan Koçyiğit. "A mobile computing framework based on adaptive mobile code offloading." Software Engineering and Advanced Applications (SEAA), 2014 40th EUROMICRO Conference on. IEEE, 2014, Verona, Italy.
- Çetin Kaya Y., Özkan Yıldırım S. & **Kaya M**. Effects of the Usability and Expected Benefit on M-Service Usage: The Case of a Location-Based Mobile Campus Service, 2014, 8th Interfaces and Human Computer Interaction (IHCI), Lisbon, Portugal.
- **Kaya, Mahir**, and Altan Koçyiğit. "Mobil Uygulamalarda Vekil Tabanlı Kod Taşıma Yönteminin Farklı Seviyelerdeki Bulut Bilişim Altyapılarının Kullanılması Durumundaki Başarımının Karşılaştırılması." UYMS - 2014, Güzelyurt, KKTC.
- Çetin Kaya Y., **Kaya M**., Özkan S., Lokasyon Tabanlı Mobil Kampus Uygulaması ve Kullanılabilirlik Değerlendirmesi, UYMS-2014, Güzelyurt, KKTC.
- **Kaya, Mahir**, and Onur Demirors. "E-Cosmic: A Business Process Model Based Functional Size Estimation Approach." Software Engineering and Advanced Applications (SEAA), 2011 37th EUROMICRO Conference on. IEEE, 2011, Oulu, Finland.
- B.Bakır, S.Özkan, R.Köseler, T.Taşkaya Temizel, D.İncebacak, and **M.Kaya** (2009), The Attitudes of students with diverse backgrounds on computer and information literacy subjects: Evidence from a first year course, Proceedings of the 39th Annual Frontiers in Education (FIE) Conference.
- R.Köseler, T.Taşkaya Temizel, B.Bakır, D.İncebacak, **M.Kaya** and S. Özkan (2009), Work in progress: Iterative curriculum development for an interdisciplinary online-taught IT course. Proceedings of the 39th Annual Frontiers in Education (FIE) Conference.

JOURNAL PAPERS

- **Kaya, Mahir**, Koçyiğit A. and Eren P.E. A Mobile Cloud Computing Framework Using a Call Graph Based Model, Revision submitted to Journal of Network and Computer Applications.

- **Kaya Mahir**, Özpınar M., Çetin Y., & Tuğba Taşkaya Temizel, MobileMETU: A Mobile Campus project based on web services, Global Journal on Technology, Vol 3 (2013): 3rd World Conference on Information Technology (WCIT-2012).