UNIVERSITY OF TARTU
Institute of Computer Science
Software Engineering Curriculum

Talha Mahin Mir

Incentive Models for Mobile Code Offloading to Imporive it's Adaptability

Master Thesis (30 ECTS)

Supervisor: Prof. Satish Srirama

Tartu 2018

Incentive Models for Mobile Code Offloading to Imporive it's Adaptability

Abstract: Mobile cloud computing has been rising in popularity in recent years due to the advantages it brings to the mobile devices. Mobile devices are mostly resource constrained and using cloud computing technologies they can perform even highly resource intensive tasks efficiently. For performing resource intensive tasks on the cloud, mobile devices need to delegate those tasks to the cloud for which two major techniques are in use today namely Task delegation and Code offloading. In task delegation model, mobile device consumes web services provided by the cloud through an API. Whereas in Code offloading model, app is partitioned to identify resource intensive tasks which are then transferred to the server for remote processing. Various techniques have been in use for performing code offloading but none of them are economically viable due to which this model is not frequently used in the industry. In this thesis, we tried to address the issues which make code offloading expensive and came up with code offloading model that can make the process economically viable. We developed a game theoretic model that provides incentive to mobile users to open their devices for offloading. Simulation and a small prototype have also been developed to validate the mathematical model.

Keywords: Mobile cloud, mobile applications, code offloading, mobile web services, IOT

CERCS: P170 - Computer science, numerical analysis, systems, control

Mobiilset koodi mahalaadimist soodustavad mudelid, et parandada selle kohanemisvõimet

Abstract: Mobiilne pilvearvutus on viimastel aastatel populaarsemaks muutunud, kuna see toob mobiiliseadmetele mitmeid eeliseid. Mobiiliseadmed on enamasti piiratud ressurssidega kuid pilvearvutustehnoloogiate abil suudavad need tõhusalt täita isegi väga ressursimahukaid ülesandeid. Pilves ressursimahukate ülesannete täitmiseks peavad mobiiliseadmed delegeerima need ülesanded pilvele, mille jaoks tänapäeval kasutatakse kahte peamist tehnikat:

ülesannete delegeerimist ja koodi offloadimist. Ülesannete delegeerimise mudelis kasutab mobiiliseade API kaudu pilve poolt pakutavaid veebiteenuseid. Koodi offloadimise mudelis jagatakse rakendus osadeks, et tuvastada ressursimahukad ülesanded, mis seejärel edastatakse serverile kaugtöötlemiseks. Koodi offloadimiseks on seni kasutatud erinevaid tehnikaid, kuid ükski neist pole majanduslikult elujõuline, mistõttu seda mudelit tööstuses sageli ei kasutata. Selles töös püüdsime lahendada probleeme, mille tõttu koodi offloadimine on kallis ja pakkusime välja koodi offloadimise mudeli, mis muudab antud protsessi majanduslikult elujõuliseks. Oleme välja töötanud mänguteoreetilise mudeli, mis motiveerib mobiilikasutajaid oma seadmeid offloadimiseks kasutama. Matemaatilise mudeli valideerimiseks on välja töötatud ka simulatsioon ja väike prototüüp.

Keywords: Mobiilne pilv, mobiilsed rakendused, koodide mahalaadimine, mobiilsed veebiteenused, IOT

CERCS: P170 - Arvutiteadus, arvuline analüüs, süsteemid, kontroll

List of Figures

List of Tables

Contents

1 Introduction

1.1 Problem

Mobile devices have limited computational capabilities and performing tasks that demand lots of computational resources drains down the battery life of the mobile device pretty quickly and are usually slow as well. In the recent years, mobile cloud has become increasingly popular. More and more mobile devices are using cloud services to perform resource intensive tasks on the cloud. To transfer resource intensive tasks to the cloud from the mobile device there are two major methods in use. Task delegation method and Code offloading. Task delegation follows the traditional client-server architecture where the data required for a certain operation is passed to an online server and the server then performs the task on its end and transfers the results back to the mobile device. In contrast, in code offloading, instead of passing the data, whole chunk of code that's considered resource intensive is transferred to the server where this code runs on a surrogate device and once the server is done with computation, results are passed back to the mobile. This technique has two apparent benefits - firstly because the code is running on the same device on the server as the device from which it has been offloaded, it means that in case the internet is down the code can still run on the local device. That's something which is not possible with the Task delegation model. In task delegation model, in case the internet is not available, app cannot run because the computational logic is on the server and we are just passing the data, which cannot be done if internet is down. Secondly, in case of code offloading, no major development is required on the server end as all the logic is still on the mobile end. Unlike in the task delegation model where we have to setup the server using some platform like .NET, Java etc, in code offloading we are just setting up some VM environment and all the application logic is still on the client end.

So, considering the benefits of the mobile code offloading model, it's a good candidate to be used in the research and industry for increasing the computational capabilities of the mobile devices. But despite the benefits, it's not widely used in the industry because all the frameworks and models available for code offloading right now are not economically viable.

Code offloading faces many challenges when it comes to applying it to the real time applications. Most of the time, code has non-deterministic behaviour and the time code will take to run depends on a lot of factors like current available memory of the device, device memory state, connectivity bandwidth and so on. So, deciding what, when and how to offload is an involved decision. But, major

problem with code offloading is the cost associated with it. Experiments show that in order to offload a chess game based on min-max algorithm from Samsung Galaxy S3 to Amazon's EC2 instances, in order for code offloading to be effective *m3.medium* instance should be used [3]. The *m3.medium* instance costs $0.067 per hour and if we have to make sure continuous availability of that instance, it proves to be very cost ineffective.

So, goal of this thesis was to come up with a model for mobile code offloading that is economically viable. We ended up creating a game theory based mathematical model that provides incentive to the mobile users to open their devices for offloading. This way, we tried to eliminate the dependence of the system on the central server running virtual machine instances which, as we will see in the later chapter, is the main source of cost in any code offloading framework.

1.2 Research Questions

Deriving from the problem statement, our main research task is as follows:

RQ1: To find incentive models for mobile code offloading that make the process economical.

1.3 Structure

The thesis is structured as follows: *Chapter 2* gives overview of the state of the art. *Chapter 3* describes the process we used for testing our hypothesis. *Chapter 4* describes our contribution in terms of mathematical model and framework we developed. *Chapter 5* explains the implementation details of the simulation and prototype. In *Chapter 6* we conclude the thesis, and in *Chapter 7* we touch upon the possible future paths. All references can be found in *References* section.

2 State of the art

In this chapter, we will briefly explain what code offloading is and what are some of the code offloading solutions currently available. Then we will describe *RAPID* code offloading framework [4], the framework that we extensively used for testing in this thesis. We will end the chapter by explaining our literature review that we did for finding out what incentive schemes, if any, are available right now.

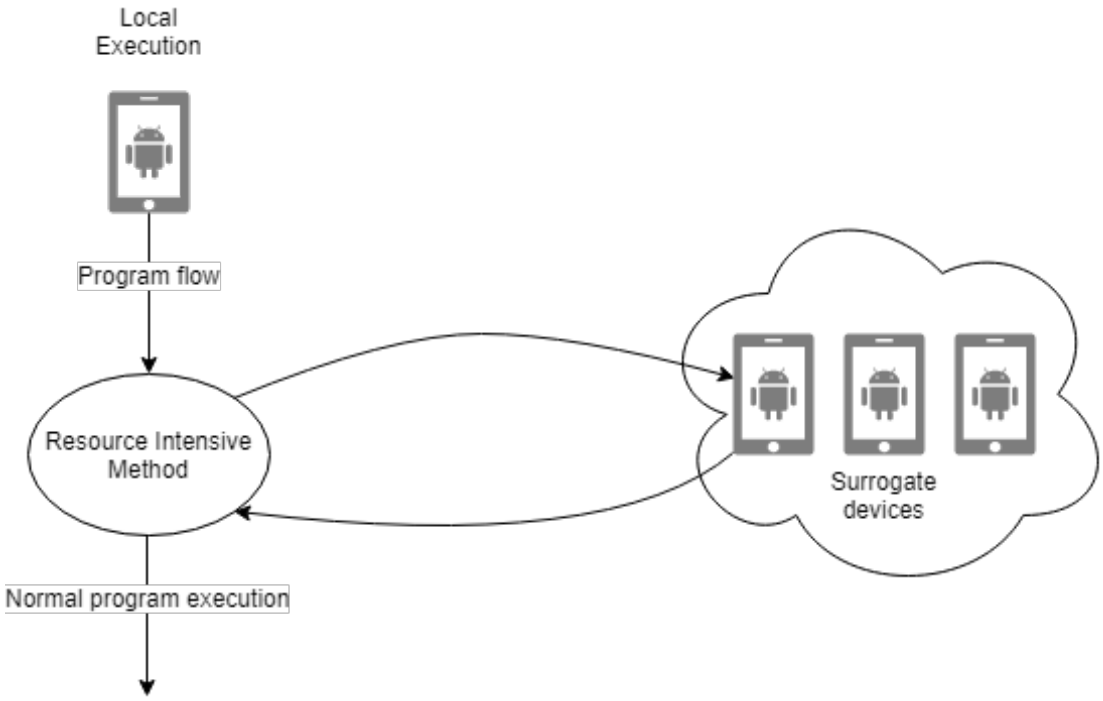

Figure 1: High-level code offloading architecture

2.1 Background

As briefly described before, code offloading is a technique of partitioning a code to identify which tasks are resource intensive and which are not. Those resource intensive tasks are then offloaded to the server where they run on the same surrogate device from which they are offloaded. Unlike on the mobile devices, on cloud those methods have access to virtually unlimited amount of resources. So, the total execution time is less as compared to when those tasks are run locally on the mobile device. The overall purpose of the offloading technique is to increase the throughput and save the processing power and energy of the device.

Now, there are couple of techniques involved in the code offloading process and different frameworks implement it in their own way. But despite the differences, they all follow almost the same architecture even when the underlying implementation details are different. Now, we will explain that code offloading architecture.

2.1.1 Code offloading architecture

A very high level representation of code offloading architecture is shown in Figure 1. It follows the traditional client-server architecture where we have a client part running on user's mobile device and a server application running on a surrogate device on a cloud. Every code offloading framework contains logic for identifying the resource intensive tasks in the program. The whole code offloading framework is essentially a framework to determine *What* to offload, *When* to offload and *How* to offload.

For deciding *what* to offload, *code profilers* are used. Code profilers are of two types namely *static* profilers and *dynamic* profilers. Static profilers are mostly annotation based profilers where the application developer annotates a certain method by using annotations like *@Offloadable*, *@Remote* etc. to specify that this method is resource intensive and should be offloaded at run-time. This techniques is certainly not very efficient as there can be many other constraints at run-time that will make the offloading of a particular method not feasible, for instance a code can be efficiently run on one device but not on other devices. To counter such issues, some frameworks use dynamic profilers. Dynamic profilers decide on run-time which code to offload. For that, there are various techniques like static analysis of the code or keeping history traces of the previous executions of the methods. This approach is more adaptable as the developer do not have to change the code for each individual devices and the offloading code can adapt to the device on which it is running.

Similarly, for deciding *when* to offload, *system profilers* are used. System profilers are used to collect device parameters like the battery state of the device, processor load, memory state, network bandwidth state etc. These parameters are important as these parameters indicate whether in a given circumstances, code offloading will result in reduction of throughput or device energy consumption or not.

Finally, *decision engine* is used to decide *how* to offload. Decision engine is the component that takes all the parameters provided by code profilers and system profilers and run some logic (like linear programming) on those parameters to decide whether offloading will be beneficial for the device or not. Usually the desired outcomes of code offloading are reduced program execution timing, energy saving, reducing device's processor load or some combination of all of these. If the decision engine can see that offloading will yield these benefits, it will decide to offload the code otherwise the code will be run locally.

The server side of the framework consists of the surrogate devices which are hosted on a cloud. Offloading code to these surrogates results in higher

Framework	Offloading approach	Offloading decision logic	Benefits	Limitations
MAUI	Code annotations	code and system profilers	Reduced battery consumption Increased performance	Not scalable
CloneCloud	Thread level VM-syncing	Static analysis dynamic profilers	20x execution speed-up 20-fold energy decrease	Limited multithreading
ThinkAir	Code annotations	code and system profilers	Energy saving Increased performance	Lacks adaptability
COMET	Thread level VM-syncing	Greedy algorithms	Speed gain of 2.88x	No cluster optimization

Table 1: Code offloading framework comparison

throughput of the program as these devices on the cloud have much more computational power available at their disposal in comparison to the general smart phones available.

Next, we will briefly explain and compare some of the code offloading frameworks currently available.

2.1.2 Related Work

A brief comparison of some of the prominent code offloading frameworks is shown in table 1. MAUI [5] is a framework that uses code annotations to indicate which methods should be offloaded. It's primarily designed to reduce the energy consumption of the device. It uses code and system profilers to decide if a code should be offloaded or not. MAUI helps to bring down the energy consumption of the device and also increases the performance of the programs but it's not very scalable as we have to create a new server proxy for every new application developed on the MAUI framework.

CloneCloud [6] is another code offloading framework. It uses static analysis to dynamically partition the code into resource intensive tasks and normal tasks and then the tasks are run seamlessly on either local device or remote surrogate

device based on their identified status by the framework. CloneCloud also encapsulates the whole app into a virtual machine on the cloud and syncs the state of the application on both ends. Whenever the thread on the smart phone end reaches a resource intensive task, it's execution is migrated to the remote server where it executes in a similar thread on a surrogate device and once the execution is done, the thread is migrated back to the smart phone.

Like MAUI, ThinkAir [7] is another code offloading framework that uses code annotations to indicate the method that should be offloaded at run-time. This limits it's adaptability as each new application developer needs to explicitly identify the method that should be offloaded. On the other hand, it solves the scalability issues of MAUI as it allows to create, destroy and reallocate VM as needed.

COMET [8] is another code offloading framework. It focuses more on how to offload instead of what and when. COMET allows multi-threaded applications to run on multiple machines and allows the threads to migrate freely between different machines. This creates resilience against network failures, even if one machine's network fails, other machine's have enough information to carry on the task.

Similarly, there are other code offloading frameworks like Odessa [9], EMCO [10] and COSMOS [11] that provide the same kind of benefits. Looking at all the offloading frameworks available, we can identify that there are mainly two types of offloading frameworks currently available namely method based and thread based. Method based frameworks offload the code on function or method level and usually use techniques like annotations combined with dynamic profiling. Thread based frameworks, on the other hand, split the programs into threads and works by syncing thread on both the client and server end. They use techniques like static analysis to identify resource intensive tasks and usually work by migrating threads between client and server for performing offloading.

Despite the technical differences, one thing that's common in each of these frameworks is the presence of central server that contains virtual machines running mobile device images, which is the integral part of any code offloading framework. This central server is the one that drives the cost of the code offloading up. So, if somehow this central server is removed or at-least the dependence of the whole system is reduced on this server, we can bring down the cost of the process. In section 2.3, we will also see some D2D offloading frameworks identified during research and we will also discuss their pros and cons and how they can be adapted for our purpose of making the offloading process economical.

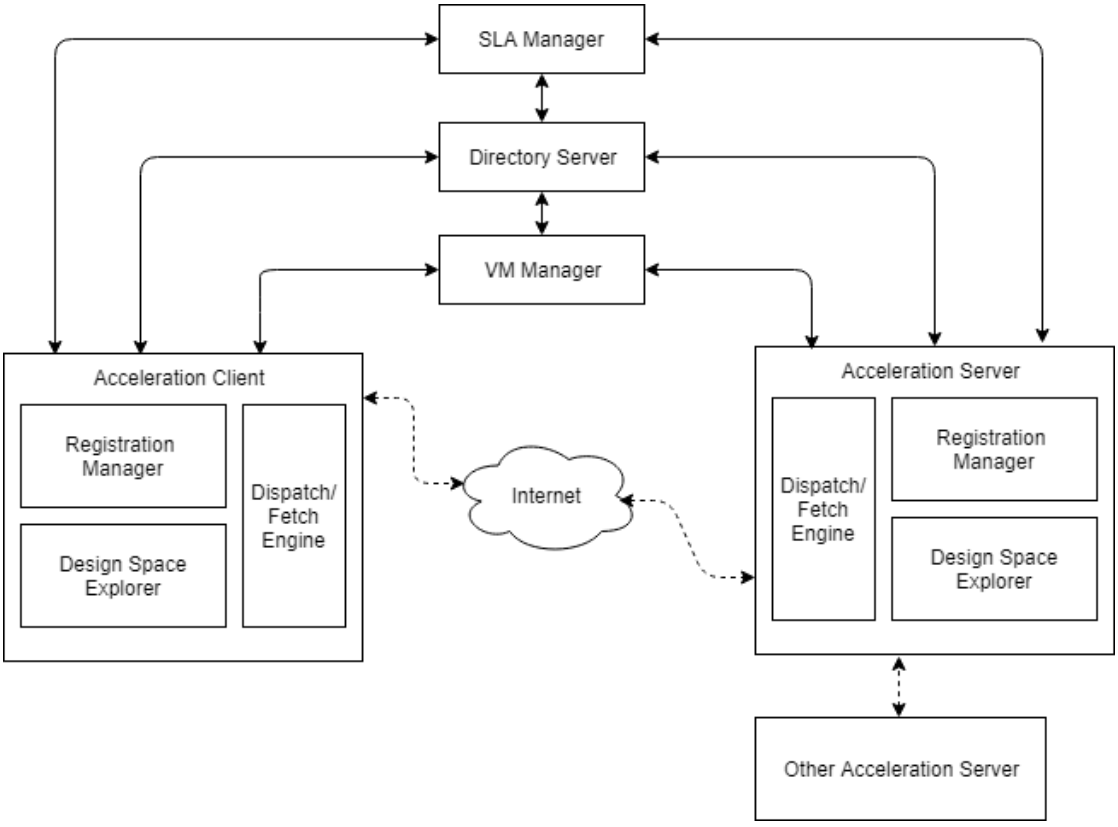

Figure 2: High-level RAPID architecture [1]

2.2 Rapid Code offloading framework

Many of the components of this thesis are built on top of the Rapid code offloading framework [4]. We've used it to test our hypothesis and also built our prototype on top of this framework. So, in this section we will briefly explain the components of the code offloading framework and we'll also look at how those components interact with one another.

Rapid is a framework which provides code offloading services for heterogeneous devices, but for the purpose of our testing we only focused on mobile devices, specifically android devices. The high level RAPID architecture is shown in Figure 2.

It has five major components:

- Acceleration Client.
- Acceleration Server.

- Directory Server.
- Service Level Agreement Manager.
- VM Manager.

2.2.1 Acceleration client (AC)

Acceleration client (AC) is a runtime library that enables code offloading on Android applications. It enables android applications to find devices to which to offload code and also decides whether the tasks defined by the programmer should be executed locally or offloaded remotely. As can be seen in Figure 2 AC is further composed of 3 major sub-components.

2.2.1.1 Design Space Explorer (DSE)

This module's main responsibility is to decide the execution location of the code - whether it should be executed locally or remotely. A simple approach is used to make this decision by keeping the history of the executions of any particular task. So, whenever a task is offloaded, DSE records it's performance parameters like execution time and energy consumption etc and every time it needs to make a decision it will look at the performance results of the previous executions of the task. Consequently, when the task is being offloaded for the first time, it simply offloads it to the server to see how it performs.

2.2.1.2 Registration Manager

The main task of the Registration Manager at the acceleration client side is to register the device to the Directory Server (DS). Directory Server allocates a unique ID to the device and provides the list of all the acceleration servers satisfying user's requirements. RM's responsibilities also include selecting one remote node from the list provided by DS and connecting to the VM of the selected entity through VM manager so that the devices can communicate and offloading can be done.

2.2.1.3 Dispatch and Fetch Engine

On the client side, this module is responsible for transferring the task's data to the acceleration server or executing the task on the local device, depending on the decision taken by the DSE. When client connects for the first time, DFE will

send the application's bytecode to the server. Afterwards, Java reflection is used to execute the methods on the server whenever offloading is done from the client side.

2.2.2 Acceleration Server (AS)

AS is an android application that runs on the VM and is responsible for executing the offloaded tasks. It's composed of more or less the same components as AC but their responsibilities differ a little.

2.2.2.1 Design Space Explorer (DSE)

On server side, DSE's responsibility is to decide if the task should be executed locally or if it should be offloaded to some other server. If enough resources are available, offloaded tasks are executed locally, otherwise they are offloaded to some other server. In either case, DFE is informed about the decision which takes the relevant action.

2.2.2.2 Registration Manager

Like on the client side, on server side Registration Manager is responsible for registering the Acceleration Server and the VMs hosting them to the Directory Server (DS). When the VM for the server starts, it informs the VMM about the availability of the VM for task offloading after which the registration manager registers the server with DS so that it is discoverable by the client applications. More detailed description of the interaction between the components can be found in the official documentation here [1].

2.2.2.3 Dispatch and Fetch Engine

On server side, DFE takes the code offloaded to the server, runs it on the VM on the server and passes the results back to the device from which the code was offloaded. In cases the DSE on server end decides that code should be offloaded to another device, it takes care of transferring the task to the other server as well.

2.2.3 Directory Server (DS)

Directory Server keeps track of all the computational resources available in the cloud infrastructure so that the mobile clients can easily find the available

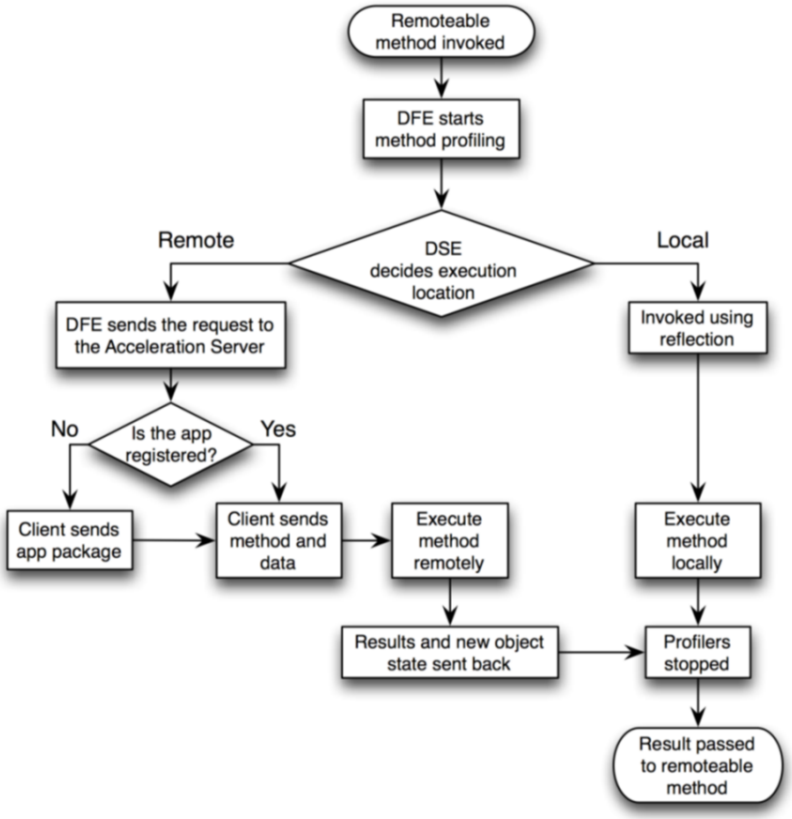

Figure 3: RAPID task execution flow [1]

resources for task execution.

Finally, **SLA Manager** ensures that the offloading to the remote machines will respect certain predefined Quality of Service (QoS) parameters whereas **VM Manager** manages the computational resources of the VMs participating in the system.

To sum up, the interaction between the different components in the system and the complete task execution flow through the RAPID framework is shown in Figure 3.

2.3 Literature Review

In this section we will explain our research methodology and will summarize the material reviewed during the literature survey.

2.3.1 Research Methodology

To identify the state of the art for incentive models for economical code offloading, we loosely followed the *Systematic Literature Review* methodology proposed by Kitchenham [12]. Search terms were derived from the research questions. Terms used were "Code offloading incentive", "computational offloading incentive", "economic code offloading", "cyber-foraging" etc. These terms were mostly used to search on Scopus and IEEE/ACM websites and papers were selected based on their relevance with the research question, number of citations and publish date. We only selected papers published in the last 2 years (2016 & 2017). None of the papers found dealt directly with economical aspect of code offloading. However, we identified different code offloading frameworks and techniques and also found some research that dealt with incentive model for code offloading in the context of autonomous and D2D networks. So initially, papers were selected for reading that dealt with different approaches for code offloading process. Then, based on the ideas of these papers and personal idea about the possible technique to be used to make code offloading economical, search has been done on finding social sharing approaches in game theory and social networking. The result of the survey is summarized below.

2.3.2 Reviewed Material

Chen et al. [13] focuses on context aware offloading. As the context of the device changes, it needs to decide to which cloud instance to offload the code. Most of the code offloading frameworks assumes that the code will be offloaded to a fixed server. In contrast, this paper uses an dynamic approach. So, if the mobile device has a strong network connection, it can offload the code to the server but if the device has poor internet connectivity, offloading to the server will not be a good idea. In such a case, following the approach proposed in the paper, it will offload the code to a nearby cloudlet.

The paper has three elements.

1. A **Design Pattern** which implements the techniques to offload code from mobile device. It divides the main program into two modules - main module and movable module. Any method that is using specific device hardware is a main module as such modules cannot be offloaded while rest of the modules are movable module.

2. An **Estimation Model** which calculates the reduced execution time and network delay and decides the appropriate resource for offloading. It

consists of two components - information models and selection algorithms. The information models calculate the reduction in execution time that can be obtained from offloading and selection algorithm decides the optimal cloud resource for code offloading

3. And finally **Framework** which implements the design pattern and estimation model to provide offloading support for mobile devices under context aware situation.

Our goal with this research is to reduce the costs associated with code offloading and this paper can be helpful in this regard as it helps reduce the dependency of the framework on the online server(major cost element) by providing context-aware offloading. So, whenever the context allows the device to offload to a nearby device (mobile, computer or a cloudlet), it will do so and will only offload to the server when no other node is available nearby and when it's absolutely necessary to do so. But the drawback of this approach is that the server running mobile virtual machine images is still up even if the user device decides to offload to the nearby cloudlet. So, irrespective of the offloading decision of the end user, the system is still incurring the cost of hosting virtual images of devices. This technique helps the end user but doesn't help the developers of the system much in reducing their infrastructure cost.

Flores et al. [14] developed a social aware hybrid code offloading system called HyMobi. The idea is to increase the availability of the code offloading system. As mobiles are almost always nearby some kind of network, so by providing code offloading opportunities to the nearby cloudlet, devices and remote server, we increase the possibilities to do code offloading. It's a kind of social network of offloading devices. The system also incorporates reward and credit system. So, whenever someone provides one's device for code offloading, one gets credits which one can later use to offload code from one's device to someone else's device. Users are also given reputation points to build trust in the network.

The proposed system has two components - peers and super-peers. Peer can be any node in the system. It includes cloudlets, remote server and any mobile device. Super-peers are the nodes that provide system level services to the other nodes. Remote server and cloudlets are the natural candidates to be a super peer but any device can become a super peer if it is running system level service on it. Super-peers keep the information about all the peers available in a certain locality and also acts as a mediator between code offloading resource provider nodes and consumer nodes. So, whenever some node wants to offload the code, it contacts the nearest super-peer and get's the list of all the nodes available. After offloading is done, the same super-peer performs the actual transaction between

provider and requester of code offloading service in terms of addition or deduction of credits

This framework helps to increase the effectiveness of the code offloading process. It aims to create a partially autonomous code offloading community of devices that works on trust system and also encourages it's users to participate more in the community by offering them rewards and reputation points. This type of community can be a helpful in creating a cost incentive model for code offloading as it reduces the dependency on online servers which brings in the major costs associated with the whole process.

Jedari et al. [15] provides a novel incentive game theoretic model to encourage selfish nodes in mobile social networks to forward data in relay services. This paper deals with bundle delivery in mobile social networks. Mobiles can have limited resources due to which some nodes in the MSN can behave selfishly by not participating in bundle delivery in the network. This paper provided a game theory model to encourage the selfish nodes in the network to forward the packet and act as a relay.

The system has two main components:

1. **Virtual currency:** Each node in the system has a virtual currency. Whenever a node provides it's services for bundle delivery, it earns the currency and whenever a node uses some other node to deliver a bundle to some destination, it loses currency.

2. **Nodes:** Each node in the system is a electronic device that have limited resources like power and memory. Nodes in the system are divided into two categories namely cooperative nodes and selfish nodes. Only the cooperative nodes (the one's which transfer the bundle) gets paid and selfish nodes (the one's which drop the bundle) cannot earn any currency units

The process employs a bargain game strategy between the bundle carrier and the relay node. There are a couple of factors that can affect the node's willingness to transfer the bundle such as node's battery state, memory state (buffer), and time to live (TTL) of the bundle itself. So, both the bundle carrier node and the relay node calculates their reserve prices for the service based on the parameters above and then based on those reserve prices, a game is played which decides the final cost of the service.

This paper, while not directly related to code offloading at all, explains a process that can be adapted for creating an incentive model for code offloading

frameworks. The paper helps to increase the contributions of the selfish nodes in the system by providing them an incentive in the form of a virtual currency, which can help increase the performance and reliability of the message transfer in MSNs. By employing the same techniques in the code offloading framework, we can create an effective community of off-loaders which can help bring down the server costs involved in the process.

Similarly, Gan et al. [16] proposed a game-based approach for multi-resource sharing among friends in social networks. The approach uses Vickrey-Clarke-Groves (VCG) based approach to offload the computation to the friends of any particular user in a certain social network. For giving users an incentive to participate in the system, the system gives monetary rewards to the user who handle the offloaded computations and there's also reputation rewards in the system. For every successful handling of code offloading request, reputation points were given to the users. Higher reputation users earn more in the system as compared to low reputation user. Real world traces from Facebook has been used to prove that the proposed algorithm satisfies the game theoretic properties of truthfulness and individual rationality.

Chen et al. [17] proposed a scheme for multi-user computation offloading in the context of mobile-edge cloud computing. Mobile edge computing [18] is network architecture where cloud computing services are provided at the edge of the mobile network. Despite the subtle differences, mobile edge computing is also sometimes referred to as fog computing [19]. Benefit of edge computing is that it helps reduce the large latency delay for offloading the code to remote cloud. Unlike other approaches which deal with offloading the code to just one device, this paper proposes a scheme for offloading to multiple devices. It's been proved through calculations that the system reaches Nash equilibrium and also gives superior computational offloading performance.

2.3.3 Summary

To wrap the review up, we found that there are no models or papers available previously that deals specifically with economical aspect of code offloading directly. But, as explained in the previous sections, there are a number of techniques and models available for offloading. Some rely on cloud servers [5], [6], [7], [8], some work by offloading code to nearby cloudlets [20], [21], [22] and some frameworks are completely device-to-device [23]. In the cloud and cloudlets case, there's central infrastructure that involves setting up and running multiple virtual machine images depending on application requirements. In a complete D2D system, there's no network infrastructure involved and all the mobile

devices communicate directly with one another. Usually WiFi-Direct or Bluetooth is used for such techniques. D2D offloading is certainly beneficial for cutting central server cost but this technique introduces extra challenges due to discontinuous connectivity between devices due to user mobility and maintenance of the whole offloading infrastructure on client devices. So, for our purposes, we resorted to a middle way of creating a partially D2D model where the central server is responsible for the connection between the 2 devices but the code is still offloaded to user devices and not to any central server. A dedicated mathematical model was also developed to provide incentive to the user. This will be explained in the subsequent sections.

3 Hypothesis Testing

In order to test our hypothesis that code offloading is indeed economically not feasible, we tested the code offloading process using Rapid code offloading framework [4] and Genymotion cloud instances on Amazon AWS [24]. *Rapid* is a framework which provides code offloading services for heterogeneous devices, but for the purpose of our testing we only focused on mobile devices, specifically android devices. The framework has 2 components - a client component and a server component. Client part of the framework was installed on a Nexus 5 device and server part of the framework was installed on Amazon AWS Genymotion instances. AWS Genymotion is a cloud based android emulator that provides all the services you can get on a local Genymotion android instance but it does it on the cloud. So, the instances are readily available and can be made accessible to anyone, anywhere in the world.

Genymotion AWS instances also come in different variants [2] optimized for different types of tasks. For instance, there are compute optimized ($C5$) instances which are designed for computationally intensive workloads. Similarly, there are memory intensive workloads ($X1e$) which are designed for memory intensive tasks and applications like high-performance or in-memory databases. For our purposes, we used general purpose ($T2$) instances which provide baseline level of CPU performance. Within general purpose ($T2$)instances, there are multiple sub instances. Their brief description is given in Figure 4.

N-Queens algorithm was used to test the hypothesis with 7 queens. N-Queens is a puzzle of arranging N queens on a N by N chess board so that no two queens can attack each other. In other words, there should be only one queen in a given row, column and diagonal. For each machine instance on the Genymotion cloud, algorithm was run 10 times and it's mean was recorded. Figure 5 shows the

Model	vCPU	CPU Credits / hour	Mem (GiB)	Storage
t2.nano	1	3	0.5	EBS-Only
t2.micro	1	6	1	EBS-Only
t2.small	1	12	2	EBS-Only
t2.medium	2	24	4	EBS-Only
t2.large	2	36	8	EBS-Only
t2.xlarge	4	54	16	EBS-Only
t2.2xlarge	8	81	32	EBS-Only

Figure 4: T2 instances specifications [2]

results.

As evident from test results, code offloading provided clear increase in the throughput of the algorithm. Algorithm was tested locally on Nexus 5 and remotely on four cloud instances of Genymotion devices and the execution time in each of the cloud instances is clearly much shorter than when the algorithm was run locally. The price structure for various cloud instances is given in Figure 6 [25]:

As can be seen in Figure 6, the cheapest cloud instance (t2.small) costs 0.151$ and if we have to make the instance available 24/7, the costs per month comes up to be 109$. Such a cost for a single, most basic level cloud instance is certainly not economically feasible. If we have to provide a code offloading solution for an industrial level application, such a cost for a single instance can become substantial as in industrial level applications we may have to provide higher end instances of the android devices with higher computational capabilities and also we will have to provide multiple instances not just a single one. Hence, we have validated our hypothesis that with the solutions available so far, code offloading is not economically viable.

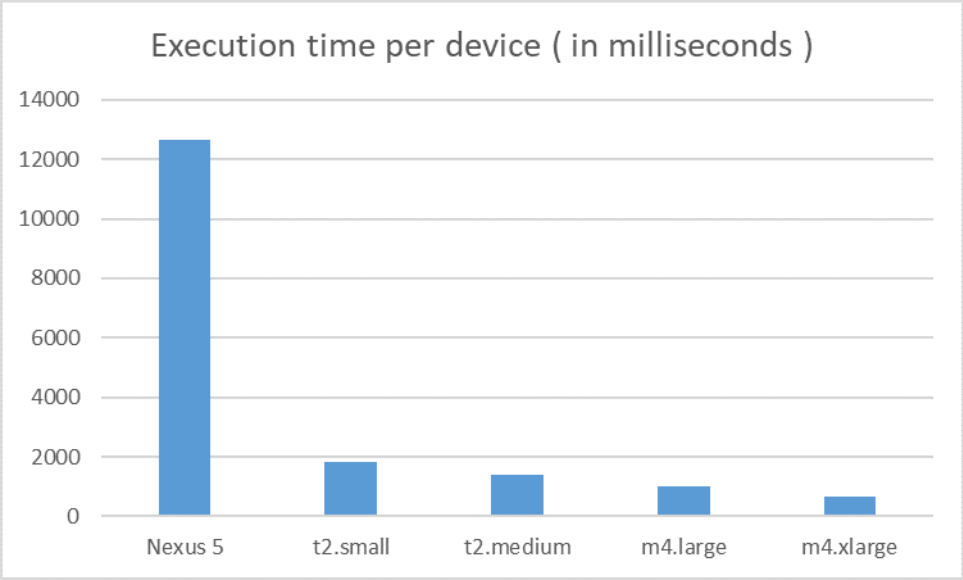

Figure 5: Hypothesis Testing Results

Genymotion on-Demand : Android 5.1 (lollipop) - Hourly			
EC2 Instance Type	Software /hr	EC2 /hr	Total /hr
m4.xlarge	$0.50	$0.232	**$0.732**
c4.8xlarge	$6.00	$1.902	**$7.902**
t2.medium	$0.253	$0.052	**$0.305**
t2.small	$0.125	$0.026	**$0.151**
m4.2xlarge	$1.00	$0.464	**$1.464**
m4.large	$0.50	$0.116	**$0.616**

Figure 6: Cloud instance rates

4 Mathematical Modeling

In this chapter, we will derive our game theoretic model for incentive mobile code offloading and will also explain the process of code offloading we are trying to propose.

4.1 System Modeling

The system we are trying to propose is the peer to peer network of off-loaders where each node can offload its tasks to some other node and can also perform operations for some other nodes. In the context of our system, the node offloading its tasks is the buyer of the service and the one executing the offloaded tasks is the seller of the service. To ensure fairness in the system and to provide incentive to the node to open their devices for offloading, we introduced a virtual currency in the system. Each time some node has to offload a task, it has to pay certain amount of currency units to the node to which it is offloading. The seller node can then use that currency units later to buy services of some other device. A competition based game theoretical model is used to calculate the price. A central server is there that keeps track of all the devices in the system. The devices continuously update their status like memory availability, battery state, network bandwidth etc to the central server. Every time a device needs to offloads it's code, it asks the server for the ip of the device to which to offload. Server, upon receiving the request, will return the most suitable device and will also calculate the price of service based on the mathematical model that will be explained in the next section. Pictorial description of the interaction between devices in the system is shown in Figure 18 and the system will be described in more detail in section 5.2.

4.2 Game theoretic model formulation

Before deriving the game theoretic model for code offloading, we'll explain the game theory concepts a bit, specially the techniques used in our mathematical analysis.

4.2.1 Background

Game theory is the study of mathematical modeling of cooperation or conflict between two or more rational decision makers. It has applications in economics,

politic science and in computer science.

Games can be of many types. There are cooperative and non-cooperative games. As their names indicate, cooperative games deal with the situations where an alliance can be formed between the players and the problem in such a games is often to find out the conditions under which the coalitions will form, which coalitions will form and what will be the payoffs of each individual player under those coalitions. On the other hand, non-cooperative games focuses on predicting individual player's strategies and payoffs under those strategies.

Similarly, there are simultaneous games and sequential games. In simultaneous games, all the players take turns simultaneously or are not aware of the actions taken by the other players hence making their turns effectively simultaneous whereas in sequential games players take turns in sequence and each latter player knows about the turns taken by the players before him. These two games differ in their representations. Quite often normal form is used to represent simultaneous games. In normal form, games are represented by a matrix indicating the individual strategies and payoffs for all those strategies for all the players in the game. On the other hand, extensive form is used to represent sequential games. In extensive form, graphs are used to represent the game with each node indicating a certain player and edge representing the action taken by that player. Every leaf node contains a tuple of payoffs for all the players for that particular set of actions i.e if that particular path in the tree is followed. Sequential games are further divided into perfect information and imperfect information game. In perfect information games, each player knows about the set of actions taken by the players before him whereas in imperfect information games all players need not have that information about the previously made moves of other players.

4.2.1.1 Elements of a game

Each game in game theory consists of some basic elements or entities. Every game contains two or more players, the set of possible actions available to those players also known as strategies and the payoffs for each player as a function of their strategies. Payoffs are represented in the form of functions called *utility functions*. These functions take the strategies or user action as input and return the payoff of that action or strategy as output.

Game models also have a notion of equilibrium most commonly known as Nash equilibrium [26] named after the American mathematician who coined it. A non-cooperative game is said to be in Nash equilibrium for a given set of strategies if no player can attain a gain in his payoff by deviating from his strategy given the other players stick to their strategies. A sub-game perfect

Nash equilibrium is a subset of Nash equilibrium used in sequential games. A set of strategies is said to constitute sub-game perfect Nash equilibrium if it is in Nash equilibrium for every subset of a larger sequential game. A common method used for determining the sub-game perfect Nash equilibrium in a sequential game is called backward induction. In backward induction we reason backwards starting from any step of the game and moving backwards, finding the optimal action for each player at every step of the game.

4.2.1.2 Summary

To sum up, in order to derive a game theoretic model for any problem, following steps are required:

1. Identify the players. Who are they and how many?
2. Identify the set of actions available to all the players.
3. Identify the payoffs associated with those actions and derive a utility function for each player.
4. Calculate the condition for Nash equilibrium to see under which actions the payoffs of each player is optimal.

4.2.2 Incentive Scheme

In our system, at any given time, there are n number of off-loaders and m number of nodes willing to accept off-loaded computation. To keep the model simple, we formulate the game as a competition between 2 nodes only, a single buyer and seller. The selection of the seller of the service is done before hand depending on factors like computation required, time constraints etc.

The interaction between the buyer and the seller of the service is modeled using the Rubinstien-Stahl bargaining game model [27]. The process starts with seller making an initial offer to the buyer. The buyer can either accept the offer in which case the game ends or buyer can make a counter offer to the seller. The game continues in this fashion until one of the nodes accept the offer. To make an initial offer, both seller and buyer needs to calculate their reserve prices. In seller's case it's the bare minimum price it requires for the service and in the case of buyer it's the maximum price it is willing to pay for the service. A number of factors affect the reserve prices which are given below:

4.2.2.1 Node Memory

Each node has limited amount of memory and it affects the node's willingness to accept the tasks for offloading or to hold it during the process. Let M_{i} denote the average remaining memory of a particular node. Then it can be represented as

$$M_i = \frac{M_{re_i}}{M_{max_i}} \times 100\% \tag{1}$$

Here $M_{\mathrm{re_i}}$ represents the remaining or free memory of the node i and $M_{\mathrm{max_i}}$ represents the maximum available memory of the node.

4.2.2.2 Node Battery

Node battery status is another factor that plays a part in determining the reserve price of the nodes. We will represent the average remaining battery by B_{i}.

4.2.2.3 Node Computational Power

Node's computational capability is another factor that affects the reserve price of the node. Greater the available computational power of the device, greater is it's willingness to open it's device for code offloading and lower it's price. Let P_{i} denote the average remaining processor speed of a particular node. Then it can be represented as

$$P_i = \frac{P_{re_i}}{P_{max_i}} \times 100\% \tag{2}$$

Here $P_{\mathrm{re_i}}$ represents the remaining or free processor speed of the node i and $P_{\mathrm{max_i}}$ represents the maximum speed of the node.

The above three factors - memory available, battery state and processing speed affects the code offloading decision of any device at any given time. So, we define a status metric out of these parameters.

$$SM_i = \alpha \log_2(M_i) + \beta \log_2(B_i) + \gamma \log_2(P_i) \tag{3}$$

Here SM_{i} is the status metric of the node i and α , β and γ are the weighing factors of the individual parameter used to adjust their importance. Here, $\alpha + \beta + \gamma = 1$

4.2.2.4 Reserve prices of nodes

Given the status metric of the nodes, we can calculate the reserve price of the nodes. For the buyer node it is given as

$$RP_B^t = \begin{cases} \frac{C_B}{SM_B} & \text{if } h_{avg}^t = 0 \\ h_{avg}^t & \text{if } h_{avg}^t > 0 \ \& \ h_{avg}^t <= C_B \end{cases}$$

Here, $RP_{\mathrm{B}}{}^{\mathrm{t}}$ denote the reserve price of the buyer node for the task t and C_{B} denote the currency units buyer node has at the moment. As can be seen, there are two different conditions for the calculation of the reserve price for any given task. If the task is being offloaded for the first time then the reserve price will be calculated on the basis of currency units available to the buyer node and the status metric as calculated in the previous section. It can be seen that greater the amount of currency available to the buyer the higher is it's willingness to pay more for the service. Conversely, the higher the remaining memory, the remaining battery and the computational power of the node, the lower is the reserve price, because the device can afford to retain that computation for some time.

This calculation for the case when the task is being offloaded for the first time can be problematic as there's a chance that the buyer will set it's reserve price pretty high in case it has accumulated a lot of currency units. To rectify that, every time the task is being offloaded, we record the cost of the task and every subsequent reserve price for the offloading of the similar task will be calculated as the average of the previous reserve prices of the task. So, the device may overpay for the first few times but gradually it will learn the price.

In the above equation h_{avg} is calculated as

$$h_{avg} = \frac{\sum_{i=1}^{n} h_i}{n} \tag{4}$$

In this equation, h_t represents the price payed for a task in the past and n is the number of times payment is done for a particular task in the past.

Similarly, for seller node, we have the equation:

$$RP_S = \begin{cases} \frac{C_S}{SM_S} & \text{if } h_{avg} = 0 \\ h_{avg} & \text{if } h_{avg} > 0 \end{cases}$$

Here, RP_{S} denote the reserve price of the seller node, C_{B} denote the currency units seller node has at the moment and h_{avg} is the average of the prices offered

to the node in the past. It can be seen that greater the amount of currency available to the seller, the lower is it's willingness to take offloading request and hence higher the reserve price. Conversely, the higher the remaining memory, remaining battery, or the remaining computation power of the device the lower is the reserve price, because the device can easily afford to perform some code offloading requests in contrast to the situation where node's memory or battery is low, in which case reserve price of the node will be high.

Here again we have two scenarios. When starting out, the seller will determine it's price based on the currency units it has but this can create a problem if seller gathers a lot of currency because then the reserve price set by the seller will be pretty high and there's a chance that it will stop receiving the offloaded code completely. Presence of such seller devices in the system with lots of currency units available can also clog the system as there's a chance that the buyers will not be able to pay for the service at all. To cater this problem, every time a seller device receives the offloaded code, the price will be recorded and the subsequent prices will be set on the previous prices offered, so gradually the seller node will adapt it's price to the market trend going on at any particular moment.

4.2.3 The bargain game

The bargain game is essentially the game to determine the division of the difference between the reserve prices of two nodes which is given by:

$$D = RP_B - RP_S \tag{5}$$

As nodes in the code offloading network are selfish, they will try to get as much proportion of the division for themselves as possible. Their utility functions under this condition are given as:

$$u_S(x_S) = x_S D - R_S \tag{6}$$

$$u_B(x_B) = x_B D - T_B \tag{7}$$

Here, u_{S} and u_{B} are the utility functions of seller and buyer respectively. x_{S} and x_{B} are the proportions of the price difference they will receive and R_{S} and T_{B} are the costs of receiving and transmitting the code offloading task respectively.

In the above equations, we have to calculate the values of R_{S} and T_{B}. As these are the costs of transferring and receiving the packets, they are directly

proportional to the size of data that needs to be transferred between the buyer and the seller node. The greater the amount of data, the longer it will take to transfer it and hence greater will be the associated cost.

Another factor that affects these parameters is the connectivity speed of the buyer and seller node. The faster the speed of connection, the shorter is the time to transfer the data and lesser will be the cost of it. Hence those costs are inversely proportional to the connection speed.

So, R_{S} can be expressed as follows

$$R_S \propto \frac{P_s}{d_s} \tag{8}$$

Here, P_{s} is the packet size and d_{s} is the download speed of the seller node. To change it into an equation, we introduce a constant, let's call it t, so now R_{S} can be written as

$$R_S = \frac{P_s}{d_s} t \tag{9}$$

Here, t should be a factor of the currency units. To calculate that, we have to consider the reserve price equation again, which, in general terms, is given by

$$RP = \frac{C}{SM} = \frac{C}{\alpha \log_2(M) + \beta \log_2(B) + \gamma \log_2(P)} \tag{10}$$

We want to base the cost of transferring the data on the minimum possible reserve price, because we want the transferring cost in the system to be minimal. RP will be smallest when the status metric(SM) will be highest. SM will be highest when all the parameters(memory, battery and processing power) are at 100%. Putting all those parameters to hundred percent and solving the above equation gives us:

$$RP_{min} = 0.152C \tag{11}$$

So, for one currency unit available, the reserve price would be 0.152. We'll use this factor in equation 9. Hence, the packet receiving cost for sender will be

$$R_S = 0.152 \frac{P_s}{d_s} \tag{12}$$

Similarly, for the buyer node, T_{B} can be calculated as

$$T_B = 0.152\frac{P_B}{U_B} \tag{13}$$

Here, P_{B} is the size of the code that's offloaded from the buyer node and U_{B} is the upload speed of the buyer node.

In the given game, the game can go on as long as no node accepts the other node's offer. This means that the game may never end. So, in order to ensure that the game will end each node has a discount factor or a patience factor denoted by δ. If we incorporate this discount factor in the system , the equations become:

$$u_S^r(x_S) = \delta_S^{r-1}(x_S D - R_S) \tag{14}$$

$$u_B^r(x_B) = \delta_B^{r-1}(x_B D - T_B) \tag{15}$$

Here, r represents the bargaining round number. Discount factor is directly linked to the factors affecting the reserve price of the nodes as well. So, in case of buyer of the service, it's discount factor will be lower if the remaining memory or the remaining battery is low. Same is the case with the seller. Discount factor's value ranges from 0 to 1. 0 being the worst and 1 being the best. So, the discount factor for node i can be given with the following formula:

$$\delta_i = \frac{\alpha M_i + \beta B_i + \gamma P_i}{100} \tag{16}$$

Here, M_{i}, B_{i} and P_{i} are the remaining memory, battery and processing power percentages of the node i as calculated previously.

Next, we have to make sure that the game stops in finite steps otherwise the proposed scheme will have no utility. For that purpose, we'll try to find the sub-game perfect Nash equilibrium in the system using backward induction. We'll assume that the game end in third round when seller makes an offer and buyer accepts it.

We'll assume x to be the proportion of difference received by seller so $x_S = x$ and $x_B = 1 - x$

Starting from round 3, let's assume the proportion offered to buyer from seller is $1 - x_3$, so the proportion received by seller is x_3.

Moving one step back to round 2, buyer has to make an offer x_2 to the seller. In order for the seller to accept the offer, it's utility in second round against that

offer should be greater than or equal to the utility seller is going to receive in round 3. This is given as:

$$u_S^2 \geqslant u_S^3 \tag{17}$$

$$\delta_S(x_2 D - R_S) \geqslant \delta_S^2(x_3 D - R_S) \tag{18}$$

Solving this for x_2 gives us:

$$x_2 \geqslant \frac{\delta_S x_3 D - \delta_S R_S + R_S}{D} \tag{19}$$

Moving one step back to round one, seller will make an offer to the buyer. If seller takes x_1 for itself in round one, then the proportion offered to buyer will be $1 - x_1$. In order for buyer to accept this offer, it's utility in round one should be at-least equal to the utility received in round two.

$$u_B^1 \geqslant u_B^2 \tag{20}$$

$$((1 - x_1)D - T_B) \geqslant \delta_B(x_B^2 D - T_B) \tag{21}$$

Here, $x_B^2 = 1 - x_2$. Now, in round 2 as the buyer was offering the proportion to the seller, it wanted to maximize it's proportion while still making the seller accept the offer which is only possible when x_2 offered by buyer is the minimum acceptable one. So the inequality above changes into an equality.

$$x_2 = \frac{\delta_S x_3 D - \delta_S R_S + R_S}{D} \tag{22}$$

Now, plugging this value into equation $x_B^2 = 1 - x_2$ and solving equation for round 1 gives us:

$$x_1 \leqslant \frac{D - \delta_B D + \delta_S \delta_B x_3 D + \delta_S \delta_B R_S - \delta_B R_S + \delta_B T_B + T_B}{D} \tag{23}$$

We've assumed that the proportions in round three are in state of equilibrium. Then $x_1 = x_3 = x$

Putting x into equation 23 and solving for x gives us

$$x = \frac{D - \delta_B D + \delta_S \delta_B R_S - \delta_B R_S - \delta_B T_B + T_B}{D(1 - \delta_S \delta_B)} \tag{24}$$

It should be noted here that this analysis is only for D2D offloading and we have not considered remote clouds and cloudlets in our system for now. As we will see in chapter 7, as part of the future work we intend to include remote clouds and cloudlets to cater for the special case where the device is not able to find any other mobile device to offload in the system. In short, offloading to clouds and cloudlets will only be done in case offloading is not possible to any mobile device and the user will have to pay for it outside the system of our code offloading framework.

4.3 Code offloading framework

Now that we have the mathematical model in place, we can list down the steps involved in the whole code offloading process. The interaction between the buyer and the seller node will take place in the following manner:

1. Whenever a node wants to offload the code, it sends the request to the central server along with all the parameters of the device and offloading task needed for price calculation as done in the previous section.

2. Central server containing the list of all the seller nodes along with their device parameters searches for the appropriate device for offloading task. It goes through the list of all the nodes available to which code can be offloaded and selects the first one with whom the deal can be made which means the reserve price difference is a positive number. The computational capability of this node should still be higher than the computational capabilities of the buyer node or should at least be equal, otherwise there would be no benefit from offloading the code. Also, the remaining memory should be higher than the memory requirements of the code to be offloaded to the node.

3. Once the central server selected seller node for offloading, it sends the device info back to the buyer node along with price information.

4. Buyer device then connects to the seller node using it's ip address and sends the offloading task to it.

5. Once the buyer device receives the result of the offloading task, it informs the server that the task is completed. The server then performs the

transaction, taking money from the buyer's account and putting it in seller's account. At this point the offloading task is considered completed.

5 Implementation

Once we have the mathematical model and the general flow of the code offloading framework in place, we went on to create the simulation to validate our mathematical model and see how it performs in terms of saving overall execution time when offloading is done. Another important factor to check was how reliable the system is in terms of offloading device availability and given a device needs to offload a certain piece of code, how often will it find a suitable device to do so. We've also run the simulation multiple times with increasing number of offloading devices to check the scalability of the system.

5.1 Simulation

For creating the simulation, first we needed to profile the task that is fairly resource intensive. N-Queens algorithm with 7 queens was chosen for the task. For profiling, the algorithm was run on an android device and Android Studio's profiler tools [28] are used to log the running time and memory consumption of the algorithm.

Simulation is done as a java project which mimics the whole code offloading framework. Devices are represented as Java classes which contains all the necessary data required for the code offloading. Every device also contains information about the bandwidth available to the device using which the time required for transferring the data during code offloading is calculated. For simulation, 1000 instances of 'buyer' and 'seller' nodes are created. All the devices contain randomly generated internal parameters. Those parameters include 'total memory', 'remaining memory', 'remaining battery', 'currency units available', 'network bandwidth available' and MIPS (Million instructions per second). The bounds of the data are chosen depending on the general parameters of the android devices available in the market. So, the total memory of the device is randomly generated between 1GB and 8GB. The processing power is randomly generated between 1.5 and 2.5 GHz. Available memory can also be any value from 0 to 8GB. Remaining battery level is in percentage so it's from 1 to 100. Currency units are upper bounded by 1000. Network bandwidth speeds are generated between 0.5 Mbps to 4Mbps. And MIPS are generated between 5000 and 1995000.

The simulation scenario is that each of the 1000 buyer nodes will try to offload to 1000 available seller nodes. For the simulation purposes, it is assumed that all the devices are offloading the code simultaneously. This also helps mimicking the high competition environment between the devices. A device will try to offload even if the local execution time is pretty less, as it could still get the benefit of saving battery life and computational power even if the remote execution time is a bit longer than local execution time. Each device will check the parameter of the available seller nodes one by one until it finds a suitable seller node. It then tries to offload the code to that seller node. If it's successful i.e the deal is reached between the buyer and seller based on the game theory model we developed, the job is considered done and timings for execution and transfer of the code are logged. Also, the device that received the offloaded code is removed from the list of available nodes, because for the sake of simulation we assumed that one node can only serve one request at a time. If the deal is not reached, the buyer node moves on to search among other available nodes until it finds one or runs out of available nodes. If that happens, the code is considered to be not off-loadable at that instance and the result is logged.

Below are the high level results of the simulation:

5.1.1 Execution time analysis

Results were obtained from running the experiment 5 times in succession to test how the system behaves in real world scenario. After every offloading, device's currency and status metric parameters are changed accordingly. The comparison of the running time on local and remote nodes is as given below:

Figure 7 shows the scatter plot of local execution times against remote execution times. This graph shows a single instance of the simulation, so individual graphs may differ slightly if simulation is run again but the distributions are more or less the same. It is evident from the figure that the processes took more time on the local machines as compared to the remote machines. Remote execution times here also include the data transfer time between buyer and seller nodes. Similarly, Figure 8 shows the density plot of the two execution times with local execution time density shown in red color. Here also we can see that more processes are centered around lower execution times for remote execution and in comparison the curve for local execution time density is more spread out.

Hence, the simulation results show that the suggested code offloading framework works and it's also viable in terms of reducing the execution time of the resource intensive task. According to the results from running simulation multiple times, on average it takes 10.23 seconds to execute the task on remote machines as

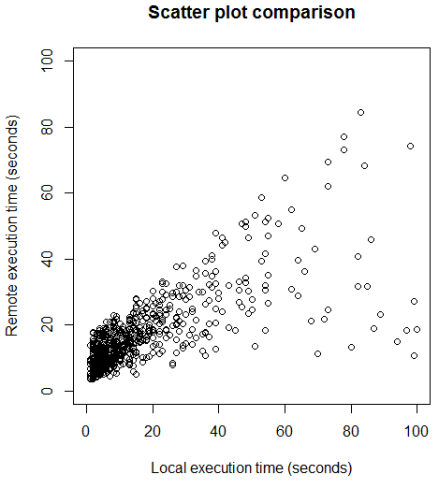

Figure 7: Scatter plot comparison

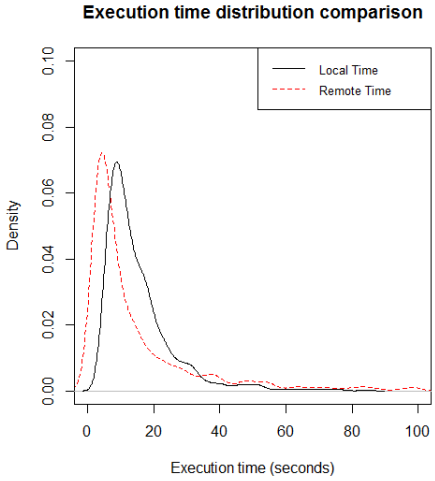

Figure 8: Density plot comparison

compared to taking 21.1 seconds on average on the local machine.

5.1.2 Reliability analysis

Just looking at the reduced execution time doesn't give us the complete picture about how reliable the system is. In order to test that, we ran the simulation with varying number of offloading devices available in the pool and for each scenario logged the cases where the buyer node was not able to find any seller node suitable for code-offloading and hence the buyer node had to perform the resource intensive task locally. The result of the analysis is shown in Figure 9.

It can be seen that as the number of devices increase in the system, the failure rate decreases pretty sharply. So, if the system has only 100 devices which are acting as seller, the failure rate would be 32% but if we have 25,000 devices in the system the failure rate goes down to 8.5%. This is a good number given that there's no central server involved in the system and the offloading results depend on the devices available to the off-loader at the time he's trying to offload the code.

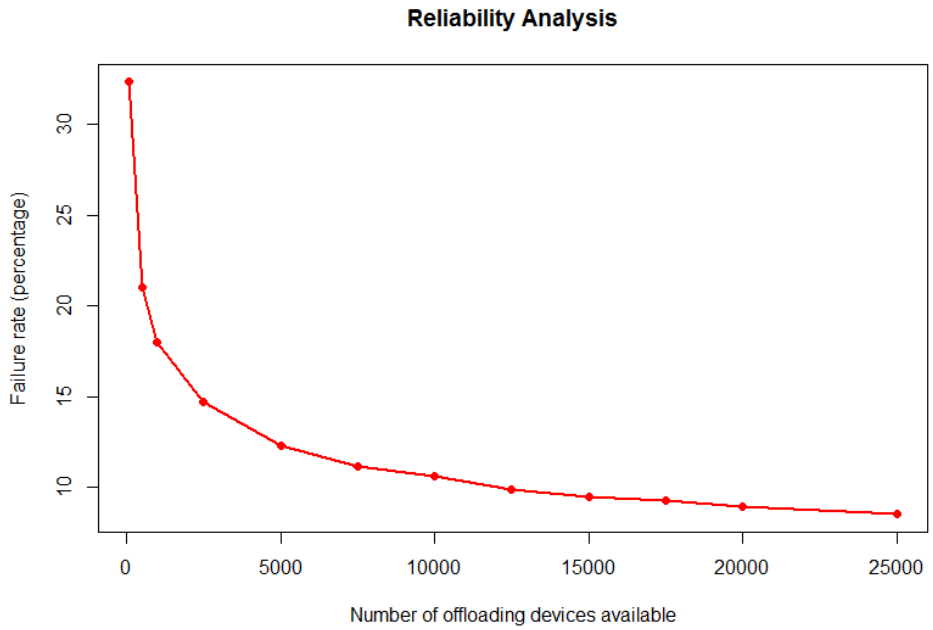

Figure 9: Reliability Analysis

5.1.3 Scalability analysis

Similar to reliability analysis, in order to test if our proposed approach is scalable or not, we again ran the simulation with varying number of offloading devices available in the pool and calculated the average time taken by the central server to find the device for offloading. It's important for our system to find the seller device for the buyer efficiently because otherwise our approach will result in increasing the execution time of the process and hence decreasing the benefits of the offloading system. The result of the analysis is shown in Figure 10.

Figure 10 shows the graph between number of offloading devices available and the time taken to get the offloading device from the server. It can be seen that the server processing time is pretty low and as the number of devices increase, the processing time also increases linearly. Hence the system is pretty scalable and will be able to figure out the device to which to offload code pretty efficiently. But still if there are too many devices in the pool, it can result in long processing time. To resolve that, we intend to implement user device clustering on server end as a future work which is discussed further in chapter 7.

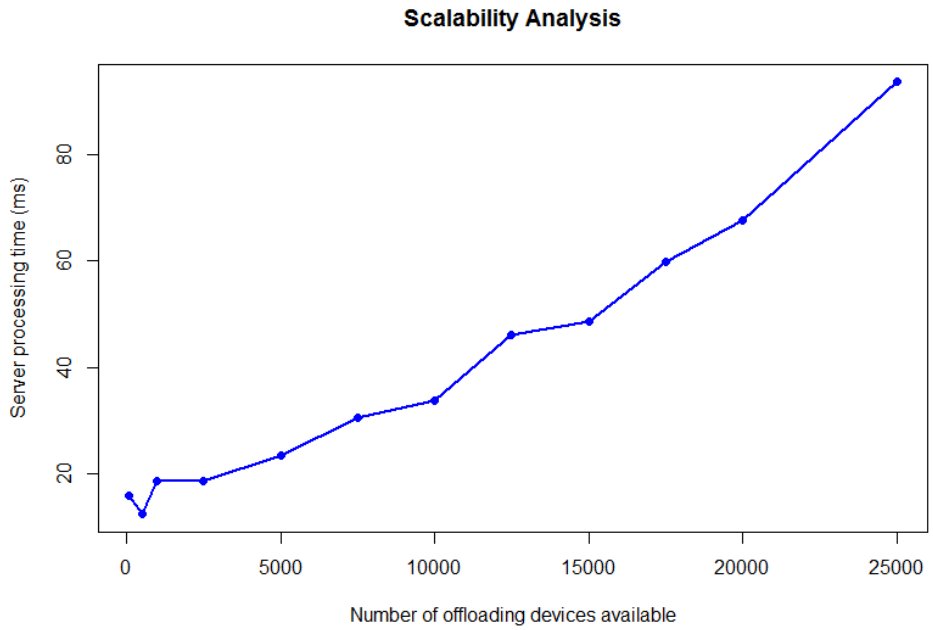

Figure 10: Scalability Analysis

5.2 Prototype implementation

In order to see how our proposed system performs in a real world environment, we developed a small prototype of the system. The prototype has 3 main components, a central server and 2 android applications, one for client end (buyer node) and one for server end (seller node). First, we will describe the individual components and then the interaction between the components will be explained.

5.2.1 Central Server

Central server is the place where we keep track of all the buyer and the seller nodes currently present in the system. All the buyer and the seller nodes periodically update their node statuses to this server. The server is developed using Java Spring Framework [29] and is backed by H2 database engine which is a Java SQL database [30]. H2 is an in-memory database meaning that it only exists for a session when the application is running. Once the application is down or stopped, the database is gone as well. We've chosen H2 over general SQL or PostgreSQL due to the simplicity of it's use. Unlike PostgreSQL, we don't have to create any separate database and create references in application

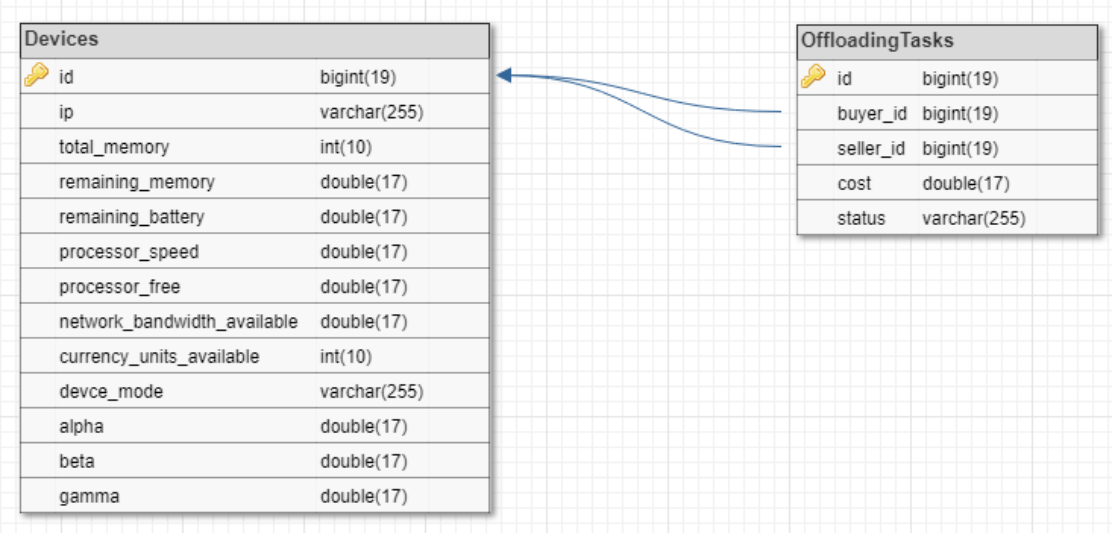

Figure 11: Central server DB Schema

configuration. We just have to include a dependency for h2 and it creates a database from scanning available Spring entities [31]. This serves our purpose for developing a prototype, but for an industrial level application, SQL or PostgreSQL is recommended.

5.2.1.1 Database Schema

Database schema for the central server is given in Figure 11 below. It consists of 2 tables -*Devices* and *OffloadingTasks*.

Devices table contains all the information associated with the devices, their total and remaining memory, battery and processor info, balance, their current ip address and device modes (Buyer or Seller), and other parameters required for the calculation of reserve prices and other factors in our game theoretic model.

OffloadingTasks contains info about the offloading tasks that are being performed right now or being performed in the past in the system. It contains a reference to the buyer and seller devices and also contains the cost of the offloading task and the status of the task (in-progress, completed, unsuccessful).

5.2.1.2 Spring web services

The central server provides access to the underlying database through Spring web services. Web services are RESTful and provide couple of endpoints to access

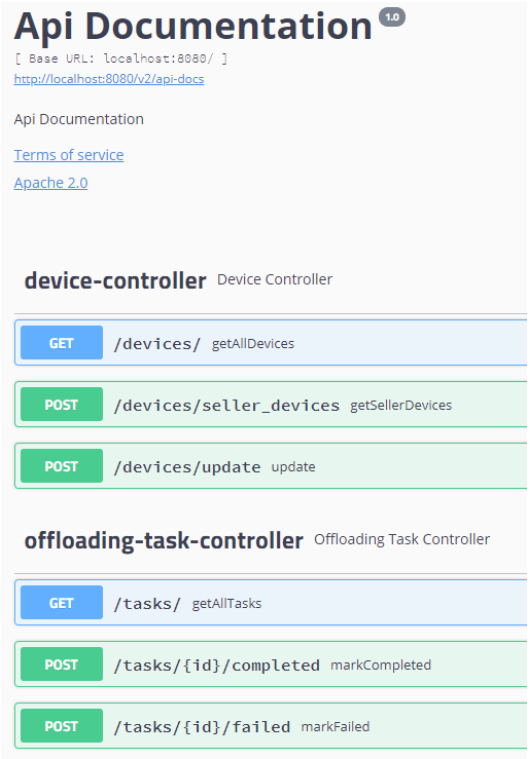

Figure 12: API endpoints

and change the status of the database. The web services are implemented in general Spring MVC pattern [32]. Database tables are created as spring entities, entities are wrapped by spring repositories [33] which exposes JPA CRUD functions to be performed over the entities. These repositories are accessed from spring services, which contains all the business logic of the code offloading framework and finally, we have RestController classes which are an initial point of contact for nodes in the system and exposes the endpoints of the system to the outside world. These controllers use services to perform business logic and return the results to the user. Results are returned in the JSON format.

For documenting the API, Swagger [34] has been used. Swagger is an API documentation software that also enables you to test your endpoints by sending the requests to your server. A snapshot of the swagger API documentation of the endpoints available in our central server is given in Figure 12.

As can be seen in the figure, the API provides endpoints for updating the status of the devices, getting list of all the devices, getting device for offloading and marking offloading tasks complete or failed.

This web service is hosted on Heroku [35]. Heroku is a cloud platform as a service (PaaS) that allows application developers to build, run and manage their application completely on the cloud. Web services are openly accessible and the Swagger documentation of the web service can be accessed here [36].

5.2.1.3 Implementation details of central server

Platform	**Lines of code**	**Number of files**
Central Server	819	32
Android client	648	10
Android Server	470	5
Simulation	529	6
Total	**2466**	**53**

Table 2: Software effort for developed platforms

The software effort for all the developed platforms is shown in table 2.

The application structure of the central server is shown in Figure 13. Application starts at the main class which is *CentralServerApplication*. Spring framework works on inversion of control principle(IoC) also known as dependency injection [37]. Under this principle, *Java Beans* [38] define their dependencies on other objects or beans through their constructor arguments and the Spring framework instantiates them for the dependent bean. So the user doesn't have to instantiate all the objects. A brief description of all the packages are as follows:

- **entities:** These are the first classes that you develop when working on any Spring application. It contains entities corresponding to the database tables in the application. There are two entities in our system - *Device* and *OffloadingTask*. This package also contains mapper classes that are responsible for converting entity classes into DTOs and vice versa.
- **repositories:** These are the Spring wrapper classes for entities and allow us to perform database operation like INSERT, GET, DELETE etc on the the underlyng entities. In our case, we just have 2 repositories - *DeviceRepository* and *OffloadingTaskRepository* corresponding to the two entities we have.

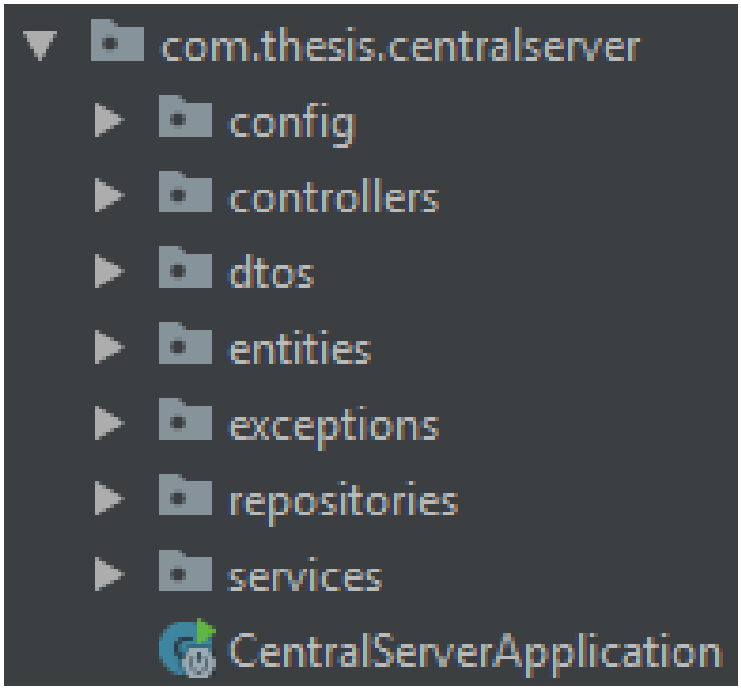

Figure 13: Central server application structure

- **services:** These are the classes that contain business logic of the application and also calls repository functions to perform database operation. We have 2 services for either one of our repositories.
- **dto:** This package contains the data transfer objects for our entities. Whenever the result is returned from any endpoint, we convert our entities into the corresponding dto through the mapper classes and pass it back to the user.
- **controllers:** These are the classes that expose the endpoints to the user of the web service. They are all annotated with *@RestController* annotaion that indicates that these are rest controllers. They use the functions from the service classes to get the required results and then pass those results back to the user. We have two controllers - *DeviceController* and *OffloadingTaskController*
- **config:** It just contains Swagger config file.
- **exceptions:** All the handled exceptions are defined in this package.

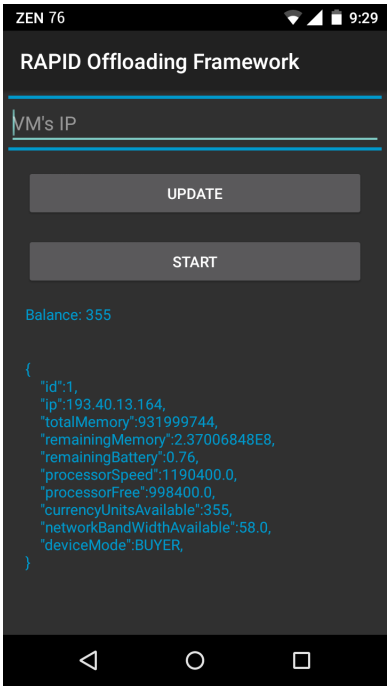

Figure 14: Buyer app main page

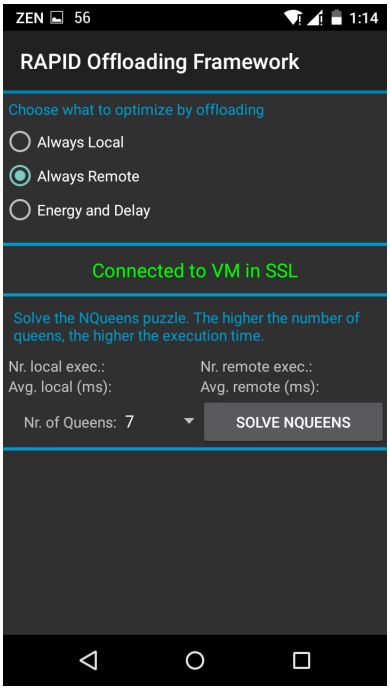

Figure 15: Offloading page

5.2.2 Mobile client app

The client or buyer end of the android app is based on the sample provided by the rapid framework [39]. The underlying logic for code offloading is the same as the original sample application but we added additional modules to make the app talk to our server and update it's status and ask the server for feasible devices for offloading.

The screen-shots of the application are shown in Figures 14 and 15.

On the main page, there's an update button that gets all the data required for price calculation like memory, battery, processor state etc and sends it to the server. For networking, Square's Retrofit networking library [40] is used. It controls the communication between the mobile app and the server and also converts the result from JSON into POJOs using the underlying GSON library [41]. GSON is a Java serialization/deserialization library that converts Java objects into JSON and vice versa.

Whenever a user wants to offload a task, he can press the *Start* button. This will result in a web call to the server which will return the device suitable for offloading(more on this in the coming sections). Once the application receives the

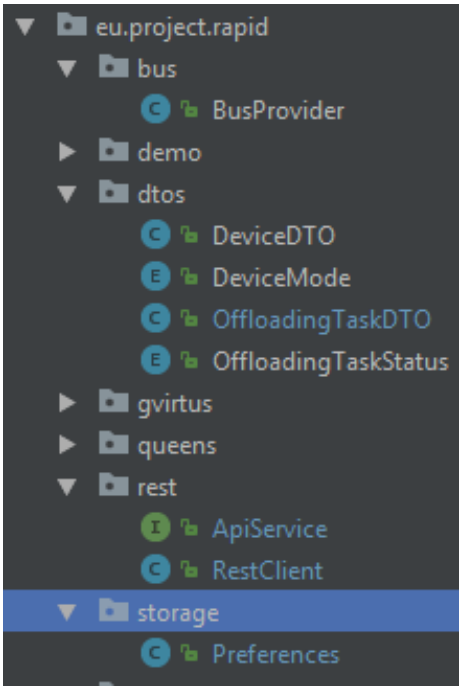

Figure 16: Android client application structure

results, it populates the text-box with *Seller* device's ip and moves to the second screen.

On second screen (Figure 15), the connection is established with the seller device and now the user can tap the *Solve Queens* buttons to send the N-Queens task to the seller device for execution. Resulting time of the tasks is also logged on the screen for user's information.

5.2.2.1 Implementation details of mobile client app

Project structure for the android client application is shown in Figure 16. The project contains lots of files and dependencies but we will just briefly explain the one's that are directly associated with our prototype.

- **Demo:** This package contains the android activity [42] classes that are the main classes responsible for showing UI on the screen. The application starts with the *MainActivity* and then transitions to *DemoActivity* when *Start* button is pressed.
- **Rest:** This package contains the classes resposible for doing web calls to our server, getting the result from the server and passing it back to the

activities from where the operation is requested. It contains *ApiService* which is an interface that defines all the REST endpoints available. *RestClient* implements the ApiService endpoints and contains the logic for interacting with the server and handling response messages from the server.

- **Bus:** This package contains just one file - *BusProvider*. It's an interface on top of Otto's [43] Bus class. Otto is an event bus for android that allows easy communication between different parts of the application. It is helpful as it let's the developer decouple different parts of the application while still allowing the communication between them. In our application we are using it to pass results from our central server from RestClient class to the Activity classes in the demo package.

- **dtos:** Like central server, we have data transfer objects on mobile end as well and since they are used to communicate with the server, they are the same as on the server.

- **storage:** This package contains *Preferences* class. This class is responsible for storing app specific data locally. We are using android's *SharedPreferences* [44] interface for that. SharedPreferences allow us to save and access data as a key value pair. We are using it to save and retrive device id and balance of the user.

5.2.3 Mobile server app

The server or seller end of the android app is based on the application server component of the rapid framework [45]. Like buyer app, it also updates the central server with the device status periodically but in the background and uses Retrofit and GSON for networking and data parsing. The application doesn't have a UI and works in the background using Android Services [46].

5.2.3.1 Implementation details of mobile server app

Project structure for the android server application is shown in Figure 17. From our prototype's perspective, server part contains almost the same classes as the client part as seller devices are also updating their status on the server in the same way as buyer devices. The only class worth mentioning here is the *AccelerationServer* class. This is an android service [46] class that is responsible for listening to the incoming connections and offloading requests and creating separate threads for each of the connecting clients. Being an android service, it runs in the background and doesn't have a UI.

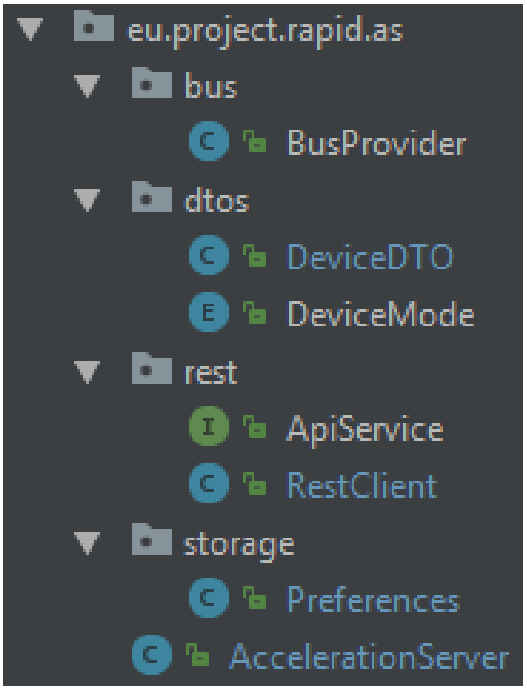

Figure 17: Android server application structure

5.2.4 Interaction of components

In this section we will explain how all the components of the system interact with each other. Complete flow of the interaction between all the components is summarized in the Figure 18. Both the mobile client app and the server app update their device status on the server. In case of client application, for demo purposes, this is done by tapping an *Update* button as shown in Figure 14. In case of server application, this is done in the background. On the client end, the result of the update call is also logged on screen for testing purposes. If the device's status is getting uploaded for the first time, server assigns an id to the device and also assigns an initial credit of 500 to the user(buyer or seller). Upon receiving the result back, the id and the balance of the device is saved locally. In each subsequent update request, this id is sent to the server so that the server knows what device is updating it's status. So, at any given moment all the buyer and seller devices are supposed to be updating their device statuses to the central server.

Now, the code offloading process starts when a device wants to offload the code and presses the *Start* button in our prototype. This fires up a web call to our server asking to get the most appropriate seller device for offloading based on the

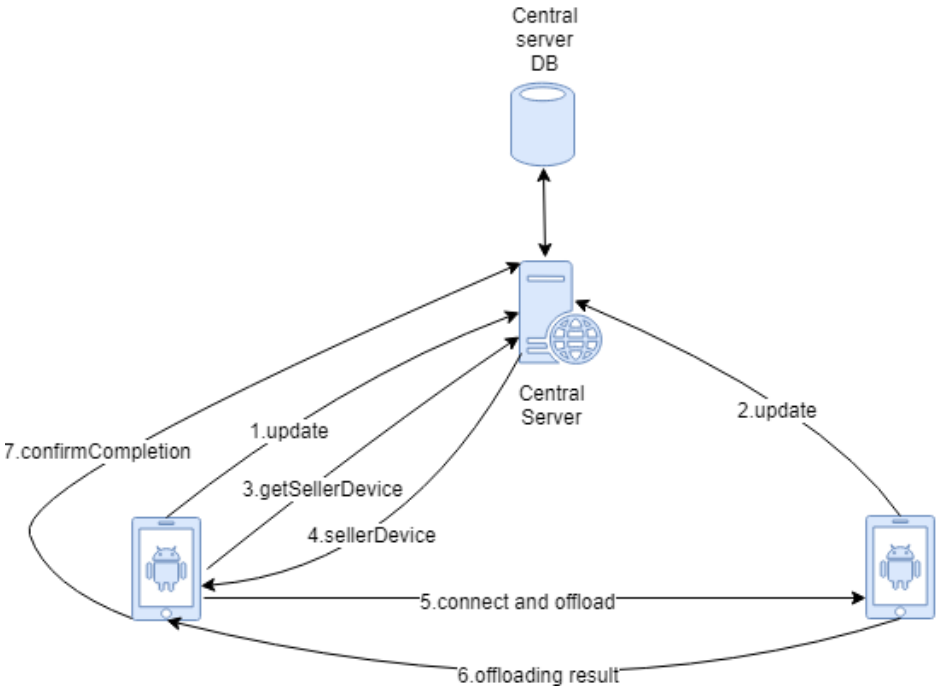

Figure 18: Prototype flow diagram

device and task parameters. All the game theoretical mathematical calculations done in the previous section are implemented on the central server. Upon receiving the request, the server looks through the list of all the seller devices and filters out the one's which have higher computational power than the one from which the offloading request is received. It then goes through the list of those filtered nodes and calculates the reserve prices if each of them and compare them with the reserve price of the buyer node until a suitable match is found which means until a seller device is found whose reserve price is lower than the reserve price of the buyer. When such a seller node is found, the server calculates the cost of the service and creates an offloading task in *OffloadingTasks* table mentioned in previous section. This task's initial status is set to *'IN-PROGRESS'*.Finally, the result is sent back to the buyer device. No payment is done until this point.

Upon receiving the seller device's information from the server, buyer device tries to connect to the device directly using it's ip address. This is done on the *Offloading page* and the status of the connection is shown on screen. Once the buyer device is connected to the seller device, it can offload the code to the seller device by taping *Solve NQueens* button. This will offload the code to the seller device and will wait for the result. Once the buyer device gets the result back

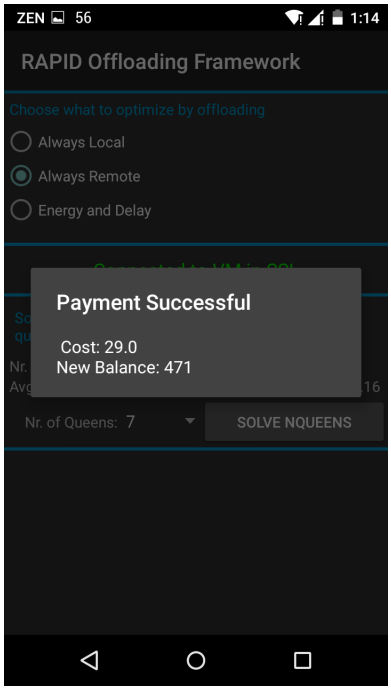

Figure 19: Payment notification

from the seller device, it informs the central server that the job is completed successfully which, in turn, will mark that offloading job as *Completed* and also performs the transaction by deducting the amount of offloading task from buyer's account and crediting them in the seller's account. This information is shown to the user on the buyer end of the device by a floating dialog window as shown in Figure 19. At this point, the whole code offloading flow is considered completed.

5.2.5 Testing of Prototype

Device	Processing power	Memory
Motorola Moto G	1.2 GHz	1 GB
LG Nexus 5	2.3 GHz	2 GB

Table 3: Testing Devices

The devices used for testing the prototype are listed in table 3. Testing was done on a local private network with central server running locally as well as testing

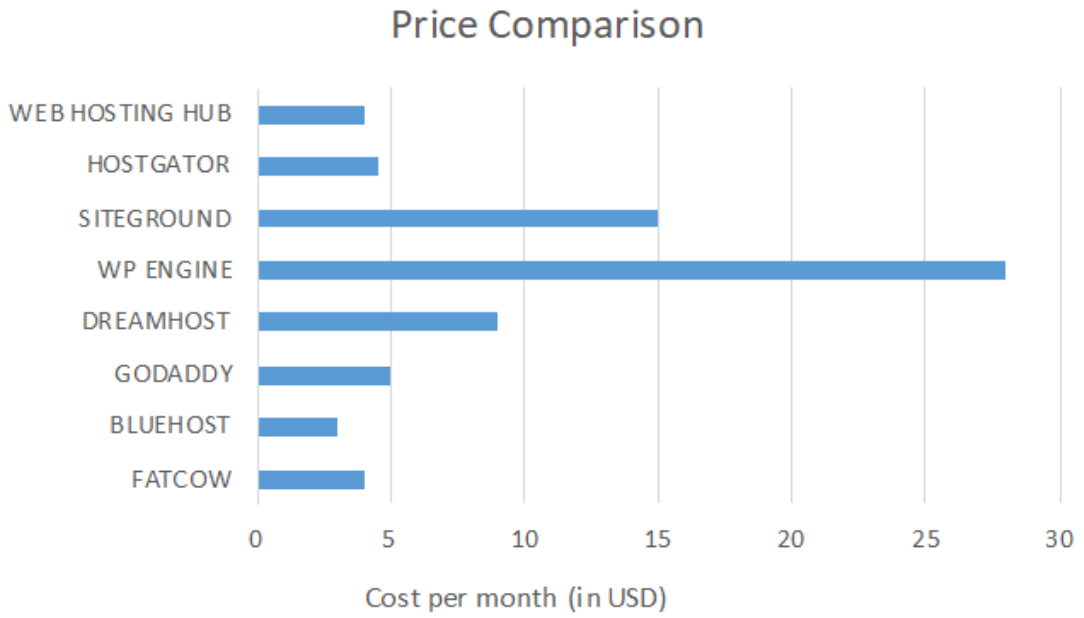

Figure 20: Prototype flow diagram

devices connected to the local network as well. Moto G was used as a buyer device due to it's inferior specifications as compared to Nexus 5 which has better specifications as evident from table 3. N-Queens algorithms was used again with 7 queens as the task to offload. The prototype corroborated our results from the simulation. The server was able to select the device for offloading successfully and offloading resulted in 78% decrease in processing time in this case. Locally on Moto G the task took around 40 seconds on average to run whereas remotely on Nexus 5 it took only 2.7 seconds on average to run.

6 Conclusion

In this thesis we identified the factors that are affecting the adaptability of code offloading frameworks in the research and industry. We identified how the presence of central infrastructure on the cloud consisting of multiple instances of VM devices brings the cost of the system up. We've looked at various techniques of code offloading currently used in the industry. In order to make code offloading economically viable so that it can be adapted in the industry, we suggested to remove the central infrastructure from the cloud and recommended a partial D2D approach for offloading. Instead of hosting surrogate VM instances

on the server, the server now just acts as a mediator between different devices and the actual offloading is done on user mobiles. We've also derived a game theoretic mathematical model to provide incentive for the user to open up their devices for offloading. In order to test our mathematical model, simulations were done. A small prototype was also developed to augment our simulations and mathematical model. Both the simulation and the prototype shows that the mathematical model works and it doesn't affect the code offloading benefits (shorter processing time, battery saving etc) in any way. Although the model is tested on Rapid framework, it can easily be adapted to any other framework as well as it doesn't require any changes on the framework level. Our system just adds an extra layer on top of any framework where it looks for the devices and decides the appropriate device to offload. The offloading is done by the underlying framework itself. So, our proposed solution is easily adaptable as well.

In order to give a brief overview of how much cost can be saved by our proposed approach, the price comparison of some of the available web hosting services is given in Figure 20 [47]. It can be seen in the graph that the most expensive web hosting service costs 27$ per month. In comparison, using cheapest android instance on AWS costs us 109$ and this is the cost for just one instance. In case of our solution with web server, we can support any number of devices as it just holds the current status of all the devices available in the code offloading network. Hence, we can see that our solution results in clear decrease in the cost of the whole process.

7 Future Work

Moving on, we intend to implement clustering of devices on our server end. Right now, the server selects the device for offloading from the pool of all the available devices from across the world. Once the system is deployed online we could have millions of devices worldwide. So, it would be feasible to do clustering based on device location and whenever a device requests the services for offloading we only look for devices in the same cluster in which the requesting device is. This will also help to reduce the network latency as the two devices will be closer to each other.

Moreover, we saw in the simulation section that in 8.5% of the cases the requesting device couldn't find any device to offload (assuming 25000 seller devices in the system). At the moment, in such a scenario device has to run the code locally. In the future, we will consider introducing some central VM server that can handle those failed cases. This will certainly increase the cost of

offloading again as central server with VM instances is exactly what we were trying to remove from the system. But in this case, the cost of the whole process will still be lower as we will not have to create a lot of instances because now only 8.5% of the devices will be offloading to the central server. Previously all the devices were offloading the code to the central server and hence we required VM instances for all of those devices. This offloading to the cloud will incur extra fee to the user of the service. So, in case mobile user is not able to find any other mobile device to offload, he will have an option to offload to the VM instance on the remote cloud by paying extra fee. Note that this extra fee will be the real cost and not the virtual currency of our D2D system. Once this user acquires the cloud resources, he will have the option to delegate offloaded code from other devices in the system to the cloud. This will increase the multi-tenancy on the server and will also provide the chance to the user to gain some virtual currency in the system.

References

[1] [Online]. Available: http://www.rapid-project.eu/_docs/RAPID_D3.1.pdf

[2] "Amazon ec2 instance types - amazon web services (aws)." [Online]. Available: https://aws.amazon.com/ec2/instance-types/

[3] S. N. Srirama, "Mobile web and cloud services enabling internet of things," Jan 2017. [Online]. Available: https://link.springer.com/article/10.1007/s40012-016-0139-3

[4] "Rapid project - welcome." [Online]. Available: http://www.rapid-project.eu/

[5] E. Cuervo, A. Balasubramanian, D.-k. 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*, ser. MobiSys '10. New York, NY, USA: ACM, 2010, pp. 49–62. [Online]. Available: http://doi.acm.org/10.1145/1814433.1814441

[6] B.-G. Chun, S. Ihm, P. Maniatis, M. Naik, and A. Patti, "Clonecloud: Elastic execution between mobile device and cloud," in *Proceedings of the Sixth Conference on Computer Systems*, ser. EuroSys '11. New York, NY, USA: ACM, 2011, pp. 301–314. [Online]. Available: http://doi.acm.org/10.1145/1966445.1966473

[7] 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 *INFOCOM*, A. G. Greenberg and K. Sohraby, Eds. IEEE, 2012, pp. 945–953. [Online]. Available: http://dblp.uni-trier.de/db/conf/infocom/infocom2012.html#KostaAHMZ12

[8] M. S. Gordon, D. A. Jamshidi, S. Mahlke, Z. M. Mao, and X. Chen, "Comet: Code offload by migrating execution transparently," in *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'12. Berkeley, CA, USA: USENIX Association, 2012, pp. 93–106. [Online]. Available: http://dl.acm.org/citation.cfm?id=2387880.2387890

[9] M.-R. Ra, A. Sheth, L. Mummert, P. Pillai, D. Wetherall, and R. Govindan, "Odessa: Enabling interactive perception applications on mobile devices," in *Proceedings of the 9th International Conference on Mobile Systems, Applications, and Services*, ser. MobiSys '11. New York, NY, USA: ACM,

2011, pp. 43–56. [Online]. Available:
http://doi.acm.org/10.1145/1999995.2000000

[10] F. H. H. P. N. P. L. E. T. S. M. J. K. V. L. Y. S. Xiang, "Evidence-aware mobile computational offloading," p. 18, 2017-11-23. [Online]. Available: http://urn.fi/URN:NBN:fi:aalto-201804042037

[11] C. Shi, K. Habak, P. Pandurangan, M. Ammar, M. Naik, and E. Zegura, "Cosmos: Computation offloading as a service for mobile devices," in *Proceedings of the 15th ACM International Symposium on Mobile Ad Hoc Networking and Computing*, ser. MobiHoc '14. New York, NY, USA: ACM, 2014, pp. 287–296. [Online]. Available: http://doi.acm.org/10.1145/2632951.2632958

[12] B. Kitchenham, "© kitchenham, 2004 procedures for performing systematic reviews," 2004.

[13] X. Chen, S. Chen, X. Zeng, X. Zheng, Y. Zhang, and C. Rong, "Framework for context-aware computation offloading in mobile cloud computing," *Journal of Cloud Computing*, vol. 6, no. 1, May 2017.

[14] H. Flores, R. Sharma, D. Ferreira, V. Kostakos, J. Manner, S. Tarkoma, P. Hui, and Y. Li, "Social-aware hybrid mobile offloading," *Pervasive Mob. Comput.*, vol. 36, no. C, pp. 25–43, Apr. 2017. [Online]. Available: https://doi.org/10.1016/j.pmcj.2016.09.014

[15] B. Jedari, L. Liu, T. Qiu, A. Rahim, and F. Xia, "A game-theoretic incentive scheme for social-aware routing in selfish mobile social networks," *Future Gener. Comput. Syst.*, vol. 70, no. C, pp. 178–190, May 2017. [Online]. Available: https://doi.org/10.1016/j.future.2016.06.020

[16] X. Gan, Y. Li, W. Wang, L. Fu, and X. Wang, "Social crowdsourcing to friends: An incentive mechanism for multi-resource sharing," *IEEE Journal on Selected Areas in Communications*, vol. 35, no. 3, p. 795–808, 2017.

[17] X. Chen, L. Jiao, W. Li, and X. Fu, "Efficient multi-user computation offloading for mobile-edge cloud computing," *IEEE/ACM Transactions on Networking*, vol. 24, no. 5, p. 2795–2808, 2016.

[18] S. Wang, X. Zhang, Y. Zhang, L. Wang, J. Yang, and W. Wang, "A survey on mobile edge networks: Convergence of computing, caching and communications," *CoRR*, vol. abs/1703.10750, 2017. [Online]. Available: http://arxiv.org/abs/1703.10750

[19] F. Bonomi, R. Milito, J. Zhu, and S. Addepalli, "Fog computing and its role in the internet of things," in *Proceedings of the First Edition of the MCC Workshop on Mobile Cloud Computing*, ser. MCC '12. New York, NY, USA: ACM, 2012, pp. 13–16. [Online]. Available: http://doi.acm.org/10.1145/2342509.2342513

[20] Y. Zhang, D. Niyato, and P. Wang, "Offloading in mobile cloudlet systems with intermittent connectivity," *IEEE Transactions on Mobile Computing*, vol. 14, no. 12, p. 2516–2529, Jan 2015.

[21] Y. Li and W. Wang, "Can mobile cloudlets support mobile applications?" *IEEE INFOCOM 2014 - IEEE Conference on Computer Communications*, 2014.

[22] P. A. Rego, P. B. Costa, E. F. Coutinho, L. S. Rocha, F. A. Trinta, and J. N. D. Souza, "Performing computation offloading on multiple platforms," *Computer Communications*, vol. 105, p. 1–13, 2017.

[23] C. Shi, V. Lakafosis, M. H. Ammar, and E. W. Zegura, "Serendipity," *Proceedings of the thirteenth ACM international symposium on Mobile Ad Hoc Networking and Computing - MobiHoc 12*, 2012.

[24] "Genymobile inc.,." [Online]. Available: https://aws.amazon.com/marketplace/seller-profile?id=933724b4-d35f-4266-905e-e52e4792bc45

[25] "Genymotion on-demand : Android 5.1 (lollipop)." [Online]. Available: https://aws.amazon.com/marketplace/pp/B06XC9L8F3?qid=1511795324048&sr=0-3&ref_=srh_res_product_title

[26] J. F. Nash, "Equilibrium points in n-person games," *Proceedings of the National Academy of Sciences*, vol. 36, no. 1, pp. 48–49, 1950. [Online]. Available: http://www.pnas.org/content/36/1/48

[27] M. J. Osborne and A. Rubinstein, "Bargaining and markets." *Economica*, vol. 58, no. 231, p. 408, 1991.

[28] "Measure app performance with android profiler," Mar 2018. [Online]. Available: https://developer.android.com/studio/profile/android-profiler.html

[29] "spring.io." [Online]. Available: https://spring.io/

[30] [Online]. Available: http://www.h2database.com/html/main.html

[31] O. Gierke, T. Darimont, C. Strobl, and M. Paluch. [Online]. Available: https://docs.spring.io/spring-data/jpa/docs/current/reference/html/

[32] [Online]. Available: https://docs.spring.io/spring/docs/current/spring-framework-reference/web.html

[33] [Online]. Available: https://docs.spring.io/spring-data/data-commons/docs/1.6.1.RELEASE/reference/html/repositories.html

[34] "World's most popular api framework." [Online]. Available: https://swagger.io/

[35] "Heroku." [Online]. Available: https://www.heroku.com/

[36] [Online]. Available: https://thesis-central-server.herokuapp.com/swagger-ui.html

[37] [Online]. Available: https://docs.spring.io/spring/docs/3.2.x/spring-framework-reference/html/beans.html

[38] [Online]. Available: https://docs.spring.io/spring/docs/current/spring-framework-reference/core.html#beans-definition

[39] RapidProjectH2020, "Rapidprojecth2020/rapid-android-demoapp." [Online]. Available: https://github.com/RapidProjectH2020/rapid-android-DemoApp

[40] "Retrofit." [Online]. Available: http://square.github.io/retrofit/

[41] Google, "google/gson." [Online]. Available: https://github.com/google/gson

[42] "Activities," Feb 2018. [Online]. Available: https://developer.android.com/guide/components/activities/index.html

[43] "Otto." [Online]. Available: http://square.github.io/otto/

[44] "android.content.sharedpreferences," Apr 2018. [Online]. Available: https://developer.android.com/reference/android/content/SharedPreferences.html

[45] RapidProjectH2020, "Rapidprojecth2020/rapid-android." [Online]. Available: https://github.com/RapidProjectH2020/rapid-android

[46] "Services," Mar 2018. [Online]. Available: https://developer.android.com/guide/components/services.html

[47] "A list of the fastest web hosting companies," Jan 2018. [Online]. Available: https://howtogetonline.com/web-hosting/fastest-web-hosting

Non-exclusive licence to reproduce thesis and make thesis public

I, Talha Mahin Mir,

1. herewith grant the University of Tartu a free permit (non-exclusive licence) to:

   1.1. reproduce, for the purpose of preservation and making available to the public, including for addition to the DSpace digital archives until expiry of the term of validity of the copyright, and

   1.2. make available to the public via the web environment of the University of Tartu, including via the DSpace digital archives until expiry of the term of validity of the copyright,

   of my thesis **Incentive model for mobile code offloading to increase it's adaptability** supervised by **Prof. Satish Srirama**

2. I am aware of the fact that the author retains these rights.

3. I certify that granting the non-exclusive licence does not infringe the intellectual property rights or rights arising from the Personal Data Protection Act.

Tartu, 20.05.2018