Offloading Computations to Mobile Devices and Cloudlets via an Upgraded NFC Communication Protocol

Chatzopoulos, Dimitris; Fernandez, C. Bermejo; Kosta, S.; Hui, Pan

Published in:
IEEE Transactions on Mobile Computing

DOI (link to publication from Publisher):
10.1109/TMC.2019.2899093

Publication date:
2020

Document Version
Accepted author manuscript, peer reviewed version

Offloading Computations to Mobile Devices and Cloudlets via an Upgraded NFC Communication Protocol

Dimitris Chatzopoulos, Carlos Bermejo, Sokol Kosta, and Pan Hui, *Fellow, IEEE*

Abstract—The increasing complexity of smartphone applications and services yields high battery consumption, while the growth of battery capacity in smartphones is not keeping up with these increasing power demands. To overcome this problem, researchers introduced the Mobile Cloud Computing (MCC) research area. In this paper, we advance on previous ideas, proposing and implementing a Near Field Communication (NFC)-based computation offloading framework. This research is motivated by the advantages of NFC's short distance communication, its better security, and its low battery consumption characteristics. We design a new NFC communication protocol that overcomes the limitations of the default protocol, such as the need for constant user interaction, the one–way communication restraint, and the limit on low data size transfer. Then, on top of the proposed solution, we implement a framework that can be used for offloading mobile applications to other mobile devices or to cloudlets equipped with NFC readers. We present experimental results of the energy consumption and the time duration of computationally and data intensive representative applications, such as *(i)* RSA key generation and encryption, *(ii)* gaming/puzzles, *(iii)* face detection, *(iv)* media download from the Internet, and *(v)* data transferring between the mobile and the cloudlet. We show that when the helper device is more powerful than the device offloading the computations, the execution time of the tasks is reduced. Finally, we show that devices that offload application parts reduce their energy consumption considerably, thanks to the low–power NFC interface and the benefits of offloading.

Index Terms—Computation Offloading, Near Field Communications, Mobile Cloud Computing, Cloudlets, Mobile Computing

1 Introduction

Mobile users today are increasingly demanding complex functionalities and sophisticated services to be supported by their devices. Unfortunately, the more complex a functionality or a service becomes, the more energy it typically consumes. As a consequence, it has become quite challenging for developers to keep their applications energy–efficient, making them struggle to carefully implement the heavy tasks of an application so as not to drain the battery very quickly, while still offering the desired services to the users. Even though there has been much interest in enhancing the lithium-ion battery capacity in smartphones, any significant improvement would take a significant amount of time to occur [1]. Eventually, a new generation of rapid-charging smartphone batteries, like nanodot-based batteries [2], will be developed, but at the time of writing of this paper these batteries are not yet available [3].

Recently, with the advent of cloud computing, one popular adopted solution is *computation offloading* [4]; a method where resource-intensive computations are executed remotely in one or more powerful machines known as *offloadees* or *surrogates*. Researchers have shown that this execution paradigm helps in reducing the energy consumption, meanwhile improving the execution time of the applications [5], [6], [7], [8]. Computation offloading frameworks make use of Wi-Fi, 3G, 4G, or Bluetooth to transmit the offloaded tasks on the remote side. This way, they can benefit from the reasonably high bandwidth or data rate. However, they face several limitations, inherent to the communication technologies, such as: interference with other WiFi or Bluetooth devices, the Internet connection requirement when offloading occurs towards the cloud (which implies higher energy needs and higher delay [5]) and the difficulty in detecting available nearby devices for computation offloading through WiFi-direct or Bluetooth [8], [9].

In this paper, we advance on the previous ideas and implement a mobile offloading framework over NFC, a protocol based on Radio Frequency Identification (RFID) [10]. NFC enables devices with a distance of less than 10 cm to exchange small amounts of data [11]. The majority of recent Android devices already implement this functionality. Although the bandwidth of NFC is typically about 50-340 times smaller than Bluetooth and Wi-Fi [12], NFC's short range and operational characteristics provide several advantages compared to Bluetooth and Wi-Fi, such as low interference and lower energy consumption [13], [14].

Many researchers believe that NFC has promising potential for future applications; with many research groups working to enhance its security and apply this technology [11], [15]. For example, Haselsteiner and Breitfu [16]

- *Dimitris Chatzopoulos (dcab@cse.ust.hk) is with the Department of Computer Science and Engineering, The Hong Kong University of Science and Technology, Hong Kong.*
- *Carlos Bermejo (cbf@connect.ust.hk) is with the Department of Computer Science and Engineering, The Hong Kong University of Science and Technology, Hong Kong.*
- *Sokol Kosta (sok@cmi.aau.dk) is with the CMI, Aalborg University Copenhagen, Denmark and Sapienza University of Rome, Italy.*
- *Pan Hui (panhui@cse.ust.hk) is with with the Department of Computer Science and Engineering, The Hong Kong University of Science and Technology, Hong Kong and with the Department of Computer Science at the University of Helsinki, Finland.*

Manuscript received: date; revised: date

discuss a list of attacks based on the signal transmission of NFC devices and potential solutions. Moreover, Fun et al. [17] improve the privacy of NFC by preventing users' information leakage on e-payment methods. Also, Dang et al. [18], [19] design attacks on an automated fare collection system by designing an NFC-enabled application that takes advantage of the plaintext-based transmission of messages.

Several advantages of NFC, especially its low–energy demands, its intrinsic security that comes from the short-range communication, and its active development, make this technology highly suitable for computation offloading.

Potential Applications: We envision a not–so–distant future reality where the *Internet of Things (IoT)* will surround us in every aspect of our life, with objects interacting with each–other in a myriad of ways. We believe that smart tables or smart desks, like the Microsoft Surface Tabletop [20], e.g., will be available in homes, offices, and even bars. Combining the computational capabilities of smart surfaces with the potential of NFC communication, we can easily see the benefits that this technology enables when people put their NFC–capable smartphones on the surface, as they usually do today. By using the NFC offloading framework, a mobile device can transfer all the heavy computations to the smart surface, reducing its energy consumption and also improving the execution time of such heavy operations. Extending our previous work, which enabled computation offloading between mobile devices via NFC [21], the enhanced framework presented in this work enables offloading tasks via NFC also to *cloudlets*, an architectural element in *edge computing*, which can be represented as a *datacenter in a box* [22], [23], [24]. The idea is similar to edge computing: *bringing the cloud closer*. The main features of a cloudlet are: it builds on standard cloud technology; it is powerful, well connected, and safe; and is close to the users, removing the need for Internet connection between end-device (e.g., smartphone) and the cloudlet (i.e., one-hop wireless connection). Cloudlets enable low latency constrained applications and services to run seamlessly, by offloading the demanding tasks to the edge.

Our Contributions: In this paper, we investigate and bypass the current limitations of the NFC protocol in order to design and build a fully functional offloading framework and potential applications of NFC-based computation offloading. A detailed explanation of our contributions is as follows:

(*1*) We modify the default NFC communication protocol to make it work without user intervention and to be able to exchange data bidirectionally; (*2*) We measure several characteristics of our new NFC protocol, such as bandwidth and latency, showing that bidirectional transmission of small data is feasible; (*3*) Following the same techniques as previous MCC frameworks, such as remote method invocation, we implement an NFC-based computation offloading framework, which is made possible thanks to our new NFC communication protocol; (*4*) We identify typical smartphone applications that can benefit from NFC computation offloading; (*5*) We evaluate the performance of our framework using five representative applications, showing that not only is offloading possible, but it is also convenient with regards to device energy consumption and tasks' execution performance.

	Technology	Coverage (meters)	Bandwidth (bps)	Latency (ms)	Power (mA)
WLAN	Wi-Fi	∼90	11M∼54M	low	219
	MIMO	∼100	300M	medium	unknown
WPAN	UWB	10 - 100	20M∼1G	low	500∼1000
	Bluetooth	∼10	1M∼3M	medium	<30
	BLE	10	∼200K	medium	<15
	ZigBee	75	20K∼250K	low	30
	NFC	<0.1	424K	low	<15

TABLE 1: A comparison between the types of wireless networks in terms of coverage, bandwidth, latency, and energy consumption. NFC underperforms in coverage and bandwidth but it presents low latency and energy consumption.

Given that there are already more than two billion NFC-enabled devices in the world and analysts estimate that the NFC market will continue to grow by 17.9% over the next decade, reaching nearly $50 billion by 2025 [25], we expect more sophisticated applications will be developed on top of NFC. We believe that the framework proposed in this work will aid in accelerating this development. Our results show that computation offloading via NFC is feasible and can be used by mobile devices that want to conserve battery or want to ask for help from trusted devices and avoid non-secure channels where data can be intercepted. Moreover, we show how the existing limitations of the NFC API can be bypassed using the Host-based Card Emulation (HCE) API. It is worth mentioning that our contributions are all at a software level and can be quickly adopted by any smartphone, smart table, or smart desk manufacturer. Furthermore, but even existing smart devices that support the HCE functionality can support the proposed framework with a simple update.

Due to the transmission speeds and bandwidth limitations of NFC, as shown in Table 1, there is always a trade-off between offloading and transmitted method/task size [26]. From the table, we can see that also Bluetooth Low Energy (BLE) presents similar power characteristics to NFC. For example, BLE has been used in power-constrained scenarios such as beacon-advertisement. However, BLE provides lower bandwidth than NFC at the same power consumption. In addition, due to the limited coverage range of NFC, imposed by the physical chips, we can neglect complex data encryption, as the low distance limits the possibilities for an attack such as man-in-the-middle or rely attacks [27], [28]. Removing the encryption requirement translates in lower tasks overhead and smaller data size to be transferred.

2 Related Work

Much work has been done on exploring the concept of computation offloading and applying it to mobile computing devices. MAUI [5] supports code offloading from smartphones to nearby servers to minimize energy consumption. While the results show significant energy savings when Wi-Fi is used, the results are not satisfactory when using 3G. The dependency on Wi-Fi restricts the usage of the framework, since smartphones do not have access to a Wi-Fi network

all the time. CloneCloud [6] aims to benefit directly from the cloud, transforming a mobile application by migrating parts of its execution to a virtual machine (*surrogate*) on the cloud. ThinkAir [7] combines the advantages of these frameworks and works with Wi-Fi and 3G, offloading to nearby or remote surrogates. Furthermore, ThinkAir allows for the computational power to be dynamically scaled up or down on the cloud, enabling high levels of flexibility for the developers. As a result, computation time and power consumption can be reduced significantly. Unlike MAUI, CloneCloud and ThinkAir can perform well with both Wi-Fi and 3G. Unfortunately, relying on long-range wireless communication increases the number of security issues and the level of wireless interference. None of the previously mentioned works have conducted any experiment using Bluetooth or any other wireless protocol.

More recently, Serendipity [8] introduces the concept of mobile–to–mobile offloading in an environment with intermittent connectivity. This system is capable of conserving energy and increasing the computation speed of low–power devices when these offload the heavy computations to more powerful ones. Nevertheless, similar to other previous frameworks, Serendipity relies only on Wi-Fi, and Lakafosis et al. do not specify how to search for and how to detect the available devices that are willing to help. OPENRP advances in this direction by collecting data from interactions between mobile users and building reputation scores per mobile user and application type [29], while Chatzopoulos et al. [9] and [30] employ a hidden market design approach to allow mobile users to place personalized sharing bounds on their resources. Honeybee [31] is an offloading framework for mobile computing on Bluetooth channels. Without having to rely on Wi-Fi, it guarantees connectivity assuming other mobile devices equipped with Bluetooth are also available. In *Enabling Android-Based Devices to High-End GPGPUs* [32], the authors advance the ideas of traditional computation offloading, introducing the possibility for code written for General-Purpose Graphics Processing Units (GPGPU) using NVIDIA's CUDA language to be offloaded from Android devices to high-end GPGPU servers. Using GPGPU virtualisation and offloading techniques, this work enables the execution of CUDA kernels on Android devices, which otherwise it is impossible as of today, paving the way for advanced scientific applications on low-power devices.

Seeing as mobile code offloading has already become well accepted and its advantages have been widely acknowledged, researchers have recently been focusing on building more solid frameworks that consider so long neglected aspects, such as security, fault–tolerance, and caching, among others. Zhang et al. [33] propose Sapphire, a programming framework that handles fault-tolerance, code-offloading, and caching. Gordon et al. [34] replicate mobile applications, which are split into execution phases in mobile servers, efficiently selecting the proper replica to proceed in the next phase, in order to improve the end users' quality of experience. Bouzefrane et al. [35] propose a security protocol for authentication between NFC applications and proximal cloudlets motivated by the fact that NFC applications can be computationally demanding. Using the idea of offloading computations to cloudlets [36], the research shows some possible scenarios of applications where resource-intensive computations are offloaded via NFC, for instance, text translation and extracting text from an image using an optical character recognition application on the cloudlet.

However, the design is quite limited, since it requires the user to constantly tap on the device: one tap when offloading the security computation and another tap when receiving the result. Conversely, in this work, we have redesigned the NFC communication protocol to eliminate the need for user intervention, which enables a convenient and automated offloading process.

NFC-based computation offloading does not need to consider most of the problems that traditional offloading frameworks face. For example, the low–range communication of NFC eliminates the need for data encryption, which of course is an overhead that current frameworks have to deal with [37]. Moreover, our architecture allows the mobile devices to connect with the powerful offloadee entities automatically, since they will be in close NFC proximity, eliminating this way the need for long registration process as presented by Kosta et al. [7] and neglected in other works.

3 Design and Implementation

In this section, we describe the requirements and the steps followed to implement a functional computation offloading framework between smartphones and from smartphones to a cloudlet over the NFC communication channel. First, we list the limitations of the current NFC hardware and software interfaces, which make it difficult to build a fully functional NFC offloading framework. Then, we describe the steps we undertake to overcome such limitations and build the framework. In the rest of the paper, we will refer to the device asking for computation offloading as the *main device* or the *offloading device*, while we will refer to the device executing the offloaded computation as the *offloadee device* or simply the *offloadee*. In Table 2 we show the specifications of the devices we use for the experiments. As the table shows, they do not have any special features, and the developed protocol can function in any smartphone that supports NFC. These devices are typical examples of the available smartphones in the market that support NFC.

In the rest of the sections, we present four different implementations with increasing degrees of complexity that incrementally build the final version of our framework.

3.1 Implementation Based on the Default NFC Protocol

To facilitate the comprehension of our technical solution, in this section we present a brief description of the NFC technology and its limitations on Android devices.

The underlying implementation of the NFC in Android is based on *Nfc Data Exchange Format (NDEF)*, which is a lightweight, binary format that is mainly used for data transmission and storage. As of today, the data transmission can only be unidirectional. This, of course, does not allow for a successful computation offloading, given that the result of the offloaded task cannot be sent back to the offloading device. Moreover, to trigger the transmission of an NDEF message (*NdefMessage*) the Android operating system requires the user to tap on the smartphone's screen.

To start testing the feasibility of NFC offloading, we first implemented a functional prototype that requires the user

Name	CPU	Memory	OS
Xiaomi Mi 3	Quad-core 2.3 GHz Krait 400	2 GB	Android 4.4.4
Samsung Galaxy Note 3	Quad-core 2.3 GHz Krait 400	3 GB	Android 4.4
Samsung Galaxy Note 2	Quad-core 1.6 GHz Cortex-A9	2 GB	Android 4.4.2
LG G Pro 2	Quad-core 2.26 GHz Krait 400	3 GB	Android 4.4.2
Cloudlet: Dell Latitude E7270	Intel Core i5-6300U CPU@2.40GHz 2.50GHz	8 GB	Windows 7 Professional

Cloudlet Interface	Interface	Power (mA)
ACR122U USB	USB	200

TABLE 2: Characteristics of the devices used in our experiments. We employed 4 mobile devices, one cloudlet, and a NFC reader that allows the cloudlet to establish NFC connections with the smartphones.

intervention in several steps, as depicted in Figure 1. Specifically, the user must tap once on the main device's screen to send the task for remote computation to the offloadee device. Once the remote computation is finished, the user must tap on the offloadee's screen to send back the result to the main device. The need for constant user intervention makes this strategy unusable in practice. In particular, even if the requirement of taping the main device—which usually will be the user's device—could be tolerated, the requirement of the second tap on the offloadee device is not realistic. The user would need to continually monitor the execution on the offloadee device until it is finished before tapping for the second time to send the result back to the main device. For this reason, we investigate other solutions that allow more flexibility and more transparent implementation.

On our path towards the final solution, we investigated the connection between the Android API framework and the NFC driver. We wanted to find a relationship and a way to access and edit the responsible source code, in order to be able to bypass the second tap and to enable one-tap offloading. From our observations, we concluded that the Android NFC stack is composed of five main components:

Android core framework. It provides an API to the functionalities of the NFC system service.

Android NFC system service. It implements the high-level functionality based on the low-level interface library, which will be explained later. The first two components are written in Java.

Java Native Interface library. It bridges the high-level Java code and the low-level interface library that is written in the C language.

Low-level interface library (*libnfc-nci*). It provides a set of high-level functions to interact with the NFC controller.

NFC interface device driver. It pushes down the NFC frames to the NFC controller.

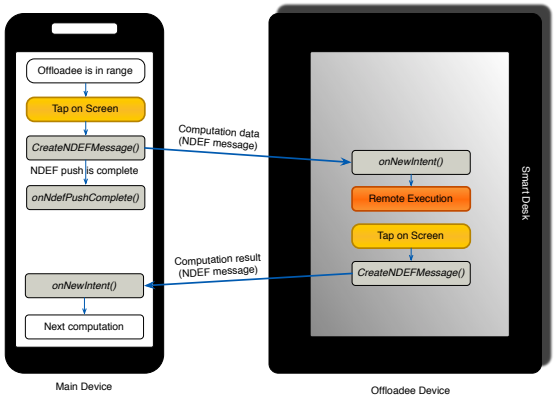


Fig. 1: Two-tap Protocol. For every NDEF message exchanged a tap in the screen of the sending device is required.

We discovered that the implementation of tapping before sending a *NdefMessage* object is located in the Android NFC system service. Unfortunately, we could only find the header file of the code and are still not able to access and edit its C file. Hence, we had to look for another alternative. Moreover, we also wanted to find a solution that would not require modifications to the Android operating system, which of course would make the adoption of our solution extremely difficult.

3.2 Utilising Host-based Card Emulation

We improve the previous implementation by removing the need for the second tap on the offloadee device. Tapping only once at the beginning and allowing the offloadee device to send back the result without the requirement of a second tap automatically creates a much more convenient user experience. To implement such functionality, we use the Android NFC Host-based Card Emulation (HCE) service, which allows any NFC-enabled smartphone to emulate an NFC card so that it can be read directly by an NFC card reader. In our case, the offloadee device emulates the NFC card, while the main device emulates the NFC card reader. The card reader communicates with the emulated card by exchanging application-level packets called Application Protocol Data Units (APDUs) and through Application ID (AID), which are used as application selectors. In some cases, more than one AID is required per application.

When the environment is set up, the card reader application creates a class that implements the *NfcAdapter.ReaderCallback* interface and initializes a *CardReaderCallback* object in the designated Activity class. This Activity class enables the reader mode by using an *NfcAdapter* object method called *enableReaderMode()*. The emulated card application, on the other hand, creates a service class that extends *HostApduService*. Once the service is extended, the emulated card mode is enabled automatically. When the card reader discovers a tag or the emulated card, the *onTagDiscovered()* function in the *CardReaderCallback* class is automatically called, which creates an APDU command to be sent to the emulated card to read the desired data. The command consists of a header and an AID and it is sent using the *IsoDep* object method called *transceive()*. Once

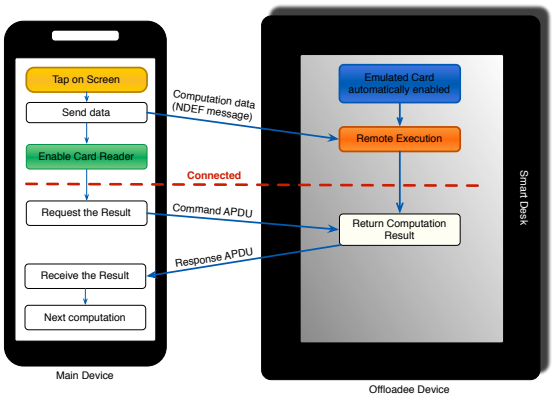


Fig. 2: HCE Protocol. Using Host Card Emulation we can bypass the need for tapping in the offloadee and send the result of the offloaded task as soon as it is ready.

the emulated card receives the command, it executes the *processCommandApdu()* function in the *CardEmulationService* class. If the AID specified in the command is the same as the AID of the application, the function can immediately return the desired data concatenated with a few bytes designating the status word, which specifies that it is the message in response to the initial APDU command. Upon receiving the response, the card reader checks the status word bytes to ensure that it is the desired message. The packet is corrupt if the status word is not the default `OK` status. If it is not corrupt, the application may continue the next computation. The HCE Implementation is depicted in Figure 2.

The most significant advantage of using the HCE implementation compared to the basic *NdefMessage* version is that it allows the main device and the offloadee device to communicate without tapping. However, it is difficult to implement it for an offloading framework, since the current Android NFC API allows only one role for each device, either as an NFC reader or as an emulated card. The solution we adopt in this implementation is first to utilise the basic *NdefMessage* transfer method to send computation data before utilising the HCE service. When the user taps on the screen, the main device sends the computation data in NDEF format. Once the transfer is complete, the main device is transformed into a card reader. The offloadee, on the other hand, is transformed in an emulated card , once receives and executes the computation. The main device will then receive the computation result(s) automatically by reading the emulated card on the offloadee device. The limitations of this implementation are twofold: *(i)* the user still needs to tap the screen on the main device, and *(ii)* the offloading process can be realised only once, since the main device becomes a card reader and is not able to send NDEF messages anymore. In order to send a second message, the main device has to deactivate the card reader in order to send a second NDEF message, which requires a new tap on the screen. Thus, if the user wants to offload another computation, or if a single message is not enough to send the required data to the offloadee, she needs to tap once again, which again raises the problem of the low automation and high user involvement.

3.3 Towards No-tap, Multiple Transfer Offloading

In this section we describe how we achieved multiple data exchange between the two devices by enabling both the card reader function and the card emulation function alternately on each smartphone. This approach requires both devices to constantly switch roles until all the computation offloading is completed, meaning that the implementation is quite challenging. We explore and implement two different strategies, namely: ***(i)*** the reader mode **disabling-enabling** method, and ***(ii)*** the reader mode **enabling-disabling** method. Compared to the previous solution, these methods present the following advantages:

No tapping is required. Both methods only utilise the HCE service, and therefore, data reading works automatically without any tapping.

No modifications of Android OS are required. User intervention can be avoided (no tapping) without the need to modify any source file on the Android OS.

Multiple data transfers are possible. Data can be exchanged between the two devices in a bidirectional manner, so the communication does not stop after just one single offloaded task.

Only one identical application is required. While other methods require different applications to be installed on the main and the offloadee devices—i.e. client-server components, this method only requires that the same application to be installed on both devices.

3.3.1 *The Reader Mode **Disabling-Enabling***

As summarised in Figure 3, the main idea of this method is to disable the reader mode on one device before enabling the reader mode on the other device (as shown in the grey dash-rounded-rectangle). The implementation relies on the emulated card service method, namely *onDeactivated()*, which will be called only when the connection to the card reader is lost. As mentioned before, the *CardEmulationService* allows the emulated card mode to be automatically enabled. In order to switch the role to the card reader mode, we found that the device only needs to call the *enableReaderMode()* function. Similarly, calling *disableReaderMode()* would switch the role back to the emulated card mode.

In the beginning, the main device acts as an emulated card, while the offloadee device acts as a card reader. When the offloadee device finishes executing the offloadable task, it immediately switches roles and becomes an emulated card by disabling the card reader mode. Once the card reader mode is disabled, the connection link to the emulated card on the main device breaks down. This triggers the *onDeactivated()* to be called, which then alerts the main device to switch role and become a card reader so that it can read the computation result from the offloadee device. There are two challenges in the implementation of this strategy: enabling role switching and solving the hardware delay problem.

The first one is the trickiest. Even though we only need to call *enableReaderMode()* and *disableReaderMode()*, those functions can only be called from an Activity or a FragmentActivity class. The solution we adopt is to create a central Activity class, namely *CentralActivity*. Concerning the role switching from the card reader to the emulated card, we apply the same procedure utilised by Android's *CardReader*

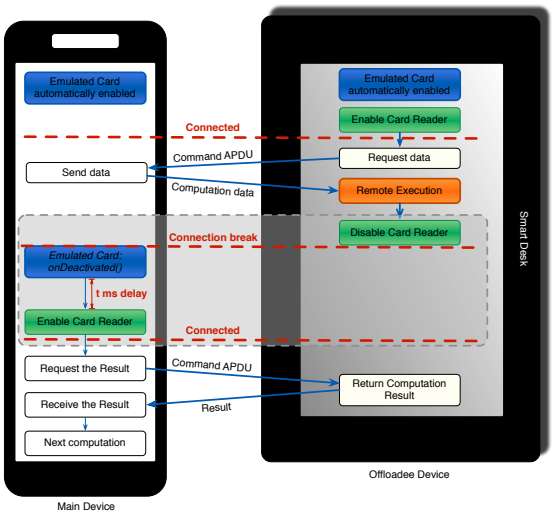


Fig. 3: The Reader Mode **Disabling-Enabling** Protocol.

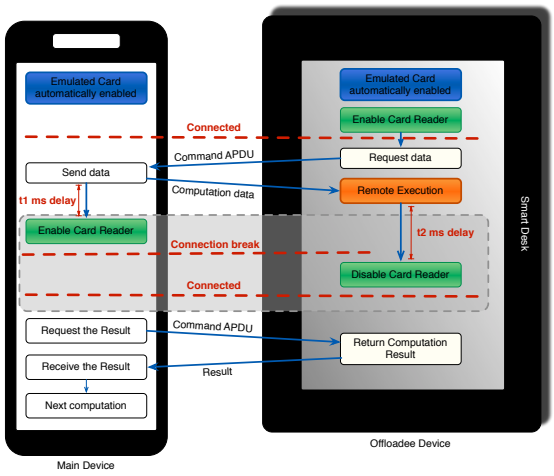


Fig. 4: The Reader Mode **Enabling-Disabling** Protocol.

sample application[1]. Using this procedure, the lines of code listed below are added into the card reader class.

```
private WeakReference<MessageCallback> messageCallBack;

public interface MessageCallback(){
    public void onMessageReceived();
}

public CardReader (MessageCallback msg){
this.messageCallBack=new WeakReference<MessageCallback>(msg);
}
```

Then, the *CentralActivity* class is modified to implement *CardReader.MessageCallback* and a new override method *onMessageReceived()* is added. We finally put *disableReaderMode()* within the overridden method. By applying this procedure, every time the card reader finishes interpreting the received message, it only needs to call *messageCallback.get().onMessageReceived()* to disable the reader mode. In order to switch role from the emulated card to the card reader, once *onDeactivated()* is called, a new intent is created to start *CentralActivity*. When started, this activity executes *enableReaderMode()* within the *onNewIntent()* method.

Regarding the second challenge, during the testing phase, we noticed that if we directly enable the reader mode once the *onDeactivated()* is called, the new connection will not be created, and the card reader will not be able to read the emulated card. We then discovered that the hardware needs some small amount of time before it can be ready to enable the reader mode. The process of selecting the proper amount of required delay is described in Section 4.1, where we discuss the fact that this delay causes an overall decrease in the bandwidth of the NFC data transmission.

3.3.2 The Reader Mode ***Enabling-Disabling***

This implementation only differs from the previous implementation in the order we disable and enable the reader mode on each device. By making such a change in the implementation protocol, we discovered that if we manage to enable the reader mode on one device before disabling the reader mode on the other, no delay is required; the connection immediately starts once the disabling reader mode occurs. The summary of this implementation is presented in Figure 4.

We need to enable the card reader on the main device while the connection is running. One solution is to create a new thread that initialises a new intent for starting the central activity. One important note, however, is to ensure that the intent starts after *processCommandApdu()* has returned the APDU response. Therefore, we set a delay on this new thread to start after t_1 ms. On the offloadee device, we have to ensure that the reader mode is disabled after the card reader on the main device has been enabled. Again, we set a delay on the disabling card reader of t_2 ms. We perform extensive experiments to find the appropriate values for t_1 and t_2. These findings are discussed in Section 4.2.

4 Measurements

In this section, we present the experiments we performed to measure the performances of the reader mode *disabling-enabling* and *enabling-disabling* protocols regarding latency and bandwidth. We evaluated the proposed protocols on two Xiaomi Mi 3 phones, whose specifications are shown in Table 2. The basic experiment setup is composed of these two devices, where one is used as the *main device* and the other as the *offloadee*.

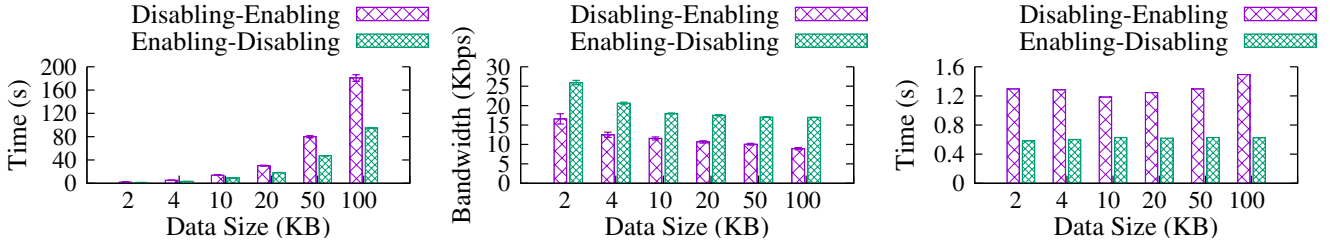
4.1 The Reader Mode Disabling–Enabling Protocol

As shown in Figure 3, the most important parameter of the *disabling–enabling* protocol is the delay t. Small values of t could enable low latency and high bandwidth but could increase the chances of transmission failure, since the hardware may not be able to switch modes in such a short amount of

t (ms)	Success Rate
680	5%
690	40%
700	82%
710	82%

TABLE 3: The success rate of the **disabling-enabling** protocol for variable values of t.

1. https://github.com/googlesamples/android-CardReader



(a) Latency (b) Bandwidth (c) Role Switching Delay

Fig. 5: Latency, Bandwidth, and Role Switching delay when sending data of different size using the *Disabling-Enabling* protocol with delay $t = 700$ ms and the *Enabling-Disabling* protocol with delays $t_1 = 310$ ms and $t_2 = 100$ ms.

time. We experiment using different values of t while sending and receiving messages of 2 KB 50 times (round–trips). Notice that we perform this round–trip experiment of small data, instead of single transmission of more extensive data, due to the limitations of the NFC packets, which should be smaller than 1 KB, according to Android guidelines. However, our experiments show that these packets can contain up to 2 KB of data. So, if an application wants to transmit more than 2 KB it should perform more than one round–trip. This process is handled automatically by our offloading framework, as we explain in detail in Section 5.1. We consider the experiment successful if and only if **all** 50 round trips were correctly performed. We repeat the experiment 20 times for each value of t and count the number of successful experiments, which divided by 20 gives the success rate: percentage of experiments that accomplished 50 round–trips. The results of these experiments are presented in Table 3, from which we select the smallest value of t such that the success rate is at least 80%, corresponding to $t = 700$ ms.

4.2 The Reader Mode Enabling–Disabling Protocol

Figure 4 shows that the *enabling–disabling* protocol is characterized by **two** delays: t_1, which is needed to enable the card reader, and t_2, which is needed to disable the card reader mode. One indicator for finding t_1 is to look for the lowest possible time delay which ensures the role switching to occur after the device has sent an APDU response. In searching for the optimal t_2, on the other hand, it is necessary that the reader mode is disabled after the reader mode on the other device is enabled. When at least one of these values is too small, the round–trip data transmission will stop with the error message: *"Error communicating with card: android. nfc. TagLostException: Tag was lost"*. The value of t_1 is initially foung by setting t_2 equal to 1000 ms so that it will not hinder the round–trip transmission. Once t_1 is selected, t_2 can be determined similarly. We follow the same process as in the previous section, sending round–trip messages of 2 KB 50 times, to measure the success rate of the experiments.

t_1 (ms)	Success Rate
250	0%
260	0%
270	30%
280	55%
290	60%
300	65%
310	95%

TABLE 4: The success rate of the **enabling-disabling** protocol for delay values $t_2 = 1000$ ms and variable t_1.

The results are presented in Table 4 and Table 5. From the first experiment we fixed $t_1 = 310$ ms and from the second we fixed $t_2 = 100$ ms. Figures 5a and 5b show the latency and the bandwidth results of the two protocols when sending data of different sizes. The delay values used are $t = 700$ ms for the *disabling–enabling* and $t_1 = 310$ ms, $t_2 = 100$ ms for the *enabling–disabling*. The *enabling–disabling* protocol presents a better performance, with latency being around 50% smaller and bandwidth being about 1.6 times higher compared to the *disabling–enabling*.

t_2 (ms)	Success Rate
50	0%
70	0%
90	0%
100	85%

TABLE 5: The success rate of the **enabling-disabling** protocol for $t_1 = 310$ ms and variable t_2.

4.3 Comparison

Given that the only difference between these two methods is the role switching process, we hypothesize that the reader mode *enabling–disabling* role switching is faster than the reader mode *disabling–enabling*. Each method requires introducing some time delays: the reader mode disabling-enabling method introduced a time delay t, while the reader mode enabling-disabling method introduced time delays t_1 and t_2. To better understand and calculate the duration of the role–switching process, we show in Figure 6 our NFC protocol of sending the data in a round–trip. We define T_{APDU} as the time needed to send an APDU command from the card reader to the emulated card and sending the APDU response back to the card reader. We define $T_{switching}$ as the time needed for both devices to switch roles. Finally, we define $T_{round-trip}$ as the total time it takes for one device to send the request, for devices to switch roles, and for the device to get the response back. As we can see, the formula to calculate the time for one round–trip can be expressed by the following equation:

$$T^{(1)}_{round-trip} = 2 \cdot T_{APDU} + T_{switching}.$$

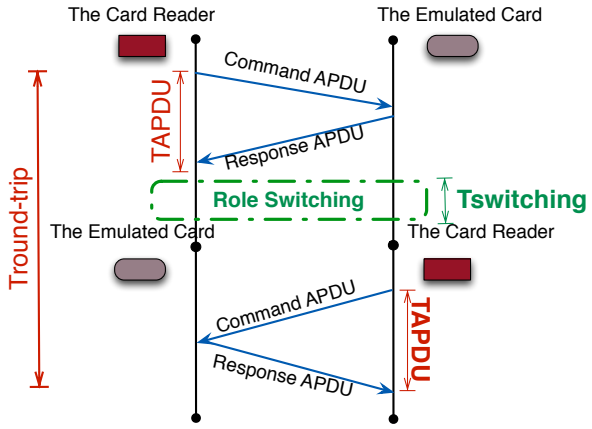


Fig. 6: Definition of T_{APDU}, $T_{switching}$, and $T_{round-trip}$.

The formula for two round–trip transmissions, assuming $T_{switchingAvg}$ is the average of all $T_{switching}$, is:

$$\begin{aligned} T^{(2)}_{round-trip} &= 2 \cdot T^{(1)}_{round-trip} + T_{switchingAvg} \\ &= 4 \cdot T_{APDU} + 3 \cdot T_{switchingAvg}. \end{aligned}$$

Iterating the formula by incrementing the number of round–trips n, we obtain:

$$T^{(n)}_{round-trip} = 2n \cdot T_{APDU} + (2n-1) \cdot T_{switchingAvg}$$

$$\text{Therefore: } T_{switchingAvg} = \frac{T^{(n)}_{round-trip} - 2n \cdot T_{APDU}}{2n-1}.$$

Given that the T_{APDU} for a message of 2 KB is 329 ms, according to experimental results, we can use the previous formula to calculate the average value of the switching time for both protocols. In Figure 5c we show the calculated values of $T_{switchingAvg}$ when sending data of a different size. Based on these results, it is apparent that the average switching time of the reader mode *enabling–disabling* protocol is less than half that of the reader mode *disabling–enabling* protocol. This finding explains the time duration difference between the two methods. As a result, we chose the reader mode enabling-disabling method in our HCE-based offloading framework implementation.

5 Final Offloading Framework

After extensive testing and evaluation, presented in Section 4, we conclude that the **enabling-disabling** strategy presents the *lowest data transmission delay* and *highest bandwidth* of all the proposed protocols. Hence, we build the offloading framework library on top of this communication protocol. In Section 5.1 we provide a generic API to be used by application developers willing to offload parts of their applications. Next, in Section 5.2 we analyse the performance of the proposed NFC offloading framework, while in Section 5.3 we list the advantages and its limitations, followed by a description of applications that can benefit from such a framework in Section 5.4.

5.1 Supported API

To enable bidirectional transmission of a large quantity of data, we design a *MessageStorage* class which stores two two-dimensional byte arrays, namely *messageToSend* — the message to be sent by the emulated card as an APDU response, and *messageReceived* — the message received by the card reader from the emulated card. As mentioned in the previous sections, this class implements a data transmission protocol based on the *enable–disable* protocol, which allows developers to transparently send and receive a large quantity of data that would be conversely impossible to achieve with the default NFC protocol. The class exposes the following methods:

setMessageToSend(byte[] message, int index) sets the $messageToSend[index]$ to the message value. It is called by the emulated card to prepare the APDU response.

getMessageToSend(int index) returns the value of $messageToSend[index]$. The emulated card calls it upon receiving an APDU command. The emulated card then sets it as the APDU response and sends it to the reader.

setMessageReceived(byte[] message, int index) sets the $messageReceived[index]$ to the message parameter value. Once the card reader receives an APDU response from the emulated card, it immediately stores the value by calling this method.

getMessageReceived(int index) returns the value of $messageReceived[index]$. It is called by the card reader to retrieve the result received by the emulated card.

On the emulated card, the APDU response is constructed by storing the desired message into the message byte array and calling the *setMessageToSend(message, 0)* method. If the message is bigger than 2 KB, we divide it into n arrays of size 2 KB or smaller. Then, each of these arrays is stored sequentially inside the class object. In order to read those messages, we manually add more application ids (AIDs) in the $aid\_list.xml$. When the emulated card receives the command APDU containing the AID, it reads the last two digits to get the index and returns *getMessageToSend(index)*. Once the card reader receives the message, it immediately calls *setMessageReceived(received_message, index)*. For further processing, it can get the received message easily by calling *getMessageReceived(index)*. Note that the main device and the offloadee device have their own *MessageStorage* class, so each of them has its own *messageToSend* and *messageReceived* attributes.

The *cloudlet* (e.g. PC) uses a similar *HCE* mode to enable the sending and receiving capabilities; it uses the API class *IsoDepTamaCommunicator* to create the API to communicate with the mobile device via the NFC protocol. In more detail, in card emulation with a secure element, the NFC emulated card is usually provided by the wireless carrier SIMs, and the data is routed through the secure element, as shown in Figure 7. In this work, we exploit the *HCE*, where the NFC card is emulated by the Android device, and there is no secure element. The transmitted data are routed to the host CPU where Android applications are running directly, as opposed to the secure element mode. The NFC reader

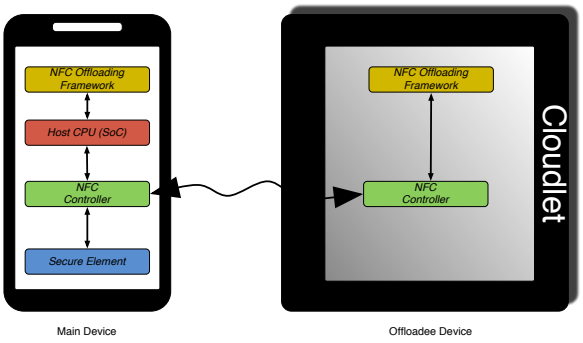


Fig. 7: Interacting components on the cloudlet setting.

ACR122 shown in Table 2 offers an API that we exploit to implement the following methods:

createConnection() establishes a connection between the card emulated in the Android device and the NFC ACR122;

sendData(byte[] b) sends data from computer to the Android device via NFC;

receiveByte() receives data sent from the Android device to the computer (NFC ACR122);

divideAndSendData(byte[] sentByte) splits the data into smaller chunks (i.e., byte array) to send via the NFC interface.

However, the SDK provided by the production company has a limitation on the size of byte array for sending at a time. For each transfer, the available byte array size for sending the data is only 260 B. Due to this limitation, we could only split the whole byte array into smaller groups of byte array with a size of 250 B.

5.2 Performance Analysis

The goal of computation offloading frameworks is to minimize the energy consumption and/or the execution time of demanding applications. Considering an application a that consumes E_a^L joules in T_a^L seconds if it is executed locally and $E_a^R(o)$ joules and $T_a^R(o)$ seconds if it is executed remotely based on the offloading decisions $o \in \mathcal{O}$, an optimal action is required to solve the following problem:

$$\textit{minimize}_o \quad \lambda(E_a^L - E_a^R(o)) + (1-\lambda)(T_a^L - T_a^R(o)) \quad (1)$$

$$\text{subject to:} \quad o \in \mathcal{O}, \lambda \in (0,1) \quad (2)$$

In case the of offloading via NFC, $\mathcal{O}$ contains all the sets of methods that can be executed on the helping device. These offloadable methods are determined by the application developers, and the offloading decisions can be made either statically or dynamically by solving the problem above. The tuning parameter λ determines whether the offloading should be focused on minimizing the energy or minimizing the execution time. $E_a^R(o)$ depends on the energy consumption of the offloadee during the remote execution and the transmitted data between the offloader and the offloadee. $T_a^R(o)$ depends on the processing capabilities of the offloadee and the bandwidth between the two devices. It is worth mentioning that existing mechanisms that have been developed for existing computation offloading frameworks, such as ThinkAir [7], can be easily applied to this work.

5.3 Advantages and Limitations

The advantages of our proposal can be categorised in the following four main points:

Fully automatic: Our framework removes the requirement of tapping. We implement the new NFC protocol using the NFC/HCE service, removing the need for user intervention and making it possible to run applications without affecting the user experience.

Portability: Application developers do not need to implement specific application versions for different device roles, i.e. *offloader* and *offloadee*. They only need to install the same application on both devices and specify the role of each device through the framework's settings. When installed on a cloudlet device, the framework will always be set to serve as offloadee.

Application execution time improvement: When a device offloads the heavy tasks of an application to a more powerful offloadee, the overall execution time of the application is reduced.

Device energy reduction: The energy consumption of the device that offloads the heavy tasks is reduced, since it is the offloadee that takes care of the computation and because the NFC data transmission consumes very little energy.

However, we are aware that our framework also presents some limitations, which are inherited by the existing underlying technologies: ***(1)*** limited bandwidth and ***(2)*** small APDU packet size. The main drawback that comes from these limitations is that the current implementation of the framework is not suitable for data-intensive applications, which need to transfer a large quantity of data during the offloading process. The APDU packet size is limited to only 2488 bytes. The limited bandwidth, combined with the relatively good channel conditions of the NFC, motivate the decision for not implementing a transmission control mechanism to handle packet losses. However, such a mechanism is easily implementable using the supported API.

Computation offloading frameworks that use other network interfaces, such as Wi-Fi, Wi-Fi direct, cellular, and Bluetooth incorporate a module that is responsible for detecting available offloadees and select the most suitable one every time there exists a task that needs to be offloaded. The selection of the most suitable offloadee is complex, depends on various parameters, and can differ per task offloading. Although the low coverage of NFC deters the need for such a module, the modular design of our framework allows its integration.

5.4 Characteristics of Suitable Applications

Considering the positive and negative aspects of our NFC protocol and our NFC offloading framework, the best application candidates suitable for NFC offloading should have the following characteristics: ***i)*** small input size, ***ii)*** small output size, and ***iii)*** high computational needs. Based on these characteristics, the proposed framework is appropriate for computationally intensive applications that do not

require high data transfers, such as ***i)*** encryption, ***ii)*** mobile payments, ***iii)*** cryptocurrencies, and ***iv)*** mathematical computations, among others.

6 Experiments

To evaluate the performance of our proposal we utilised two of the three puzzles that are proposed by Google in its Google Optimization Tools[2]. Specifically, we implemented the N Queens mathematical puzzle [38] and the Rivest-Shamir-Adleman (RSA) encryption algorithm [39]. Also, we implemented a face detection application, a file transfer application, and a link download application. In the rest of this section, we initially introduce the set up of the experiments. Then, we present a more detailed description of the applications and discuss their performance results with regards to ***i)*** execution duration and ***ii)*** energy needs. In Figure 8, we show a screenshot of the Android application that we used for running all the testing scenarios. The results are obtained by repeating each experiment 50 times and averaging. In the plots presented below we show the average and the standard deviation of the results.

Set Up: For the experiments of this section, we used a Xiaomi Mi-3 device as an offloader and a Samsung Galaxy Note 3 as an offloadee for the set of the experiments where we offloaded tasks between mobile devices. We selected the Xiaomi Mi-3 as an offloader because of its inferior specifications. For the experiments where we offloaded tasks from mobile devices to the cloudlet device, we used a Samsung Galaxy Note 2 and an LG G Pro 2. The details of the employed devices are presented in Table 2.

6.1 Examined Applications

1) N Queens: is a classic puzzle, which requires placing N queens on an $N \times N$ chess board so that no queen can attack another one. This is a typical application that is generally adopted for benchmarking, because of its high computational requirements. In our implementation, we find all possible solutions of the puzzle for a given N by using a backtracking algorithm, which has $\mathcal{O}(N!)$ complexity. During the execution of the N Queens problem, the main device, which is initially the emulated card, stores the inputted value N in the *messageToSend* two-dimensional byte array in [application_number | N] format. When the offloadee device, which is initially the card reader, is within range, the main device sends the value of *messageToSend* as the response APDU. Once the offloadee device receives the details of the sample application, it immediately executes the computation based on the received N and stores the result in its *messageToSend* variable. After both devices have switched roles, the main device reads the result from the offloadee device.

2) RSA: is an asymmetric cryptographic algorithm comprised of three main parts: *1) Key generation:* the key generation process aims to generate public and private keys from two large prime numbers. Each prime number is at least 2048 digits, which is considered to be secure [39]; *2) Encryption:* a given plaintext is encrypted using the generated public key from the previous process, and *3) Decryption:* the private key is used to decrypt the encrypted text. In our application, we offload the key generation and encryption processes. The decryption is performed on the main device after the offloading is finished, to ensure that the transferred data are not corrupted. We use the *java.security* API[3] with 2048 bits as the key length in the key generation process. Given a plain text as the input, shorter than 2048 bits, the computation produces a set of private and public keys and the encrypted message. On the main device, the application prompts the user for a plain text file. Once the user presses the *Start* button, the application reads the file and gets the plain text in bytes. The plain text is then stored in *messageToSend*. When the offloadee device is within range, the main device sends the message stored in *messageToSend*. Upon receiving the message, the offloadee immediately starts the RSA process by generating the keys and encrypting the text. It then stores the public key, the private key, and the decrypted text in *messageToSend*. Since the total size of the result is bigger than 2 KB, it is divided in two separate byte arrays. The first byte array stores the concatenated decrypted text and the public key, because the decrypted text is always 512 B (based on the key length), while the public key is always less than 1500 B. The second byte array stores the private key. After both devices switch roles, the main device reads the whole result by sending two separate APDU commands and stores all the received responses in *messageReceived*.

3) Face Detection: is a computationally demanding task that is also associated with the transmission of a picture between the offloading device and the offloadee. First, we run the face detection algorithm on the mobile device only.

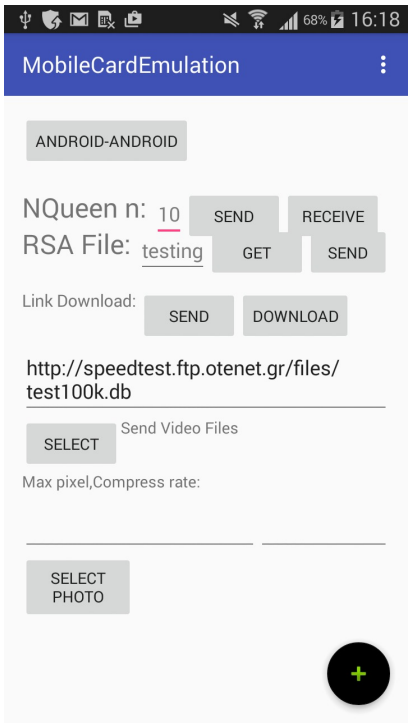
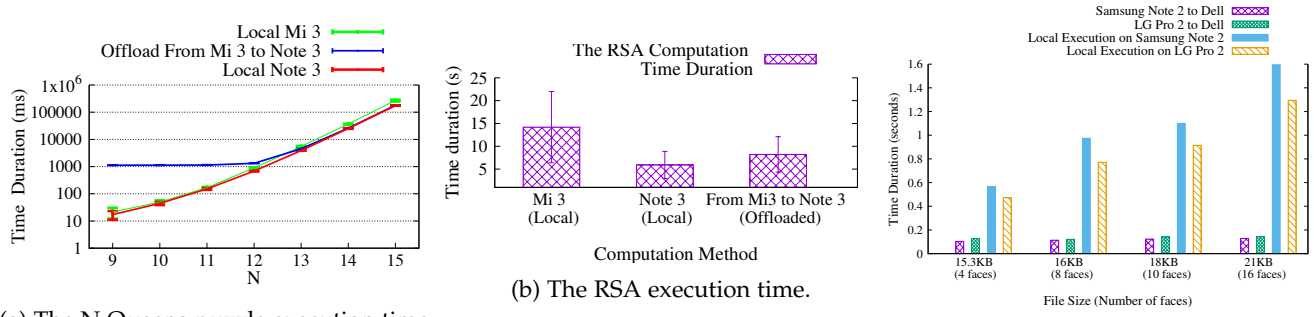


Fig. 8: Screenshot of the NFC offloading testing application.

2. https://developers.google.com/optimization/puzzles

3. https://developer.android.com/reference/java/security/package-summary.html



(a) The N Queens puzzle execution time.

(b) The RSA execution time.

(c) Face detection execution time

Fig. 9: The execution time of the N Queens puzzle, the RSA encryption algorithm and face detection.

Then, we offload the photos to the cloudlet and run the identical face detection algorithm on the computer. We perform several experiments, by varying the number of photos, the number of faces in each photo, and the size of each file. Precisely, in our experiments we use five photos, where two photoes have the same number of faces but different file size, and the other photoes have a different number of faces.

4) Data transfer: Mobile device specifications (i.e., RAM and CPU) are still the main bottleneck of computer vision tasks such as object detection and object classification [40]. Computers perform up to 1000 times faster than mobile devices for the feature extraction method in scenarios where the computer includes a GPU, and around 100 times faster for scenarios where the computer runs the tasks using only the CPU [40]. Many commercial computer vision Virtual Reality (VR) and Augmented Reality (AR) applications rely on the cloud to provide real-time experience to the users. The images transferred from the device to the cloud in these commercial applications are usually around 10 KB, otherwise they would suffer from high latency, which would obstruct the real-time experience latency requirements of 20 ms to 7 ms[4]. This requirement fits perfectly with the low bandwidth limitations of NFC, meaning that task offloading for real-time applications can be considered feasible.

As such, we have designed two experiments that deal with measuring the feasibility of data transfer in the context of computer vision applications:

Link Download is a network demanding task that downloads a file in the cloudlet and forwards it to the offloading device. In our experiments, we downloaded files with a size of 100 KB, 1 MB, and 10 MB.

Data Transfer is a task for sending data from the offloading device to the cloudlet. In our experiments we use three videos of 174 KB, 451 KB, and 870 KB.

Our framework can assist more sophisticated applications and we hope that it will help with the implementation of future killer applications in the highly active area of NFC.

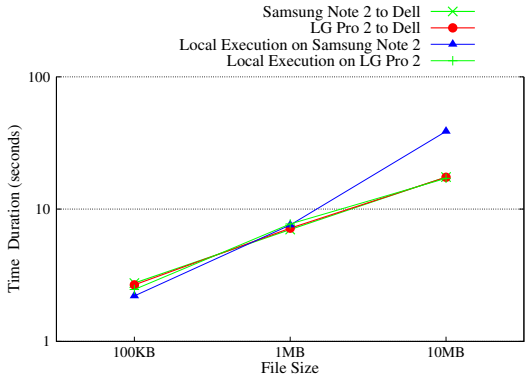
6.2 Execution Time

In the case of N Queens, we measure the execution time of the task when varying the number of queens $N = \{9, 10, 11, 12, 13, 14, 15\}$ on the Xiaomi Mi 3 and the Samsung Galaxy Note 3 devices. First, we measure the time for the case of the local execution in both devices. Then, for the remote execution we use the Xiaomi Mi 3 as the main device and the Samsung Galaxy Note 3 as the offloadee device, since based on the results of the local execution, presented in Figure 9a, the Samsung Galaxy Note 3 performs better. The measurements of the offload cases include the transmission time, the processing time, and the time to receive the result. The results of all experiments are shown in Figure 9a, where we present the average of 50 repetitions with the error bars depicting the standard deviation. For small values of N, the offloaded application has worse performance than the local ones regardless of the device. This is due to the communication overhead. However, for high values of N the execution on the Xiaomi Mi 3 is much slower than when offloading it.

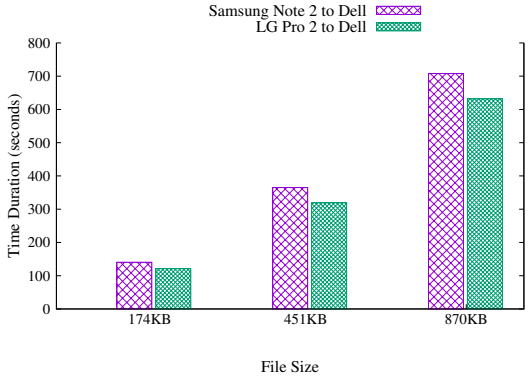
For the case of the RSA application, we measure the time duration of the local execution (on both devices), and the offloaded execution where, again, the Xiaomi Mi 3 is the main device and the Samsung Galaxy Note 3 is the offloadee device. The results, shown in Figure 9b, present the average of 50 repetitions with the error bars depicting the standard deviation. As the results show, the Xiaomi Mi 3 is almost 2.5 times slower than the Samsung Galaxy Note 3. However, if the application is offloaded, the execution time becomes much smaller on both devices.

Finally, Figure 9c presents the results of the face detection application when executed on two smartphones locally and when offloaded from these smartphones to a cloudlet PC. The cloudlet executes the offloaded tasks using only the CPU. We vary the picture size from 12 KB (4 faces in the picture) to 21 KB (16 faces in the picture). In these results, we do not consider the latency introduced by the data transmission, as we want to show the difference in processing capabilities between the mobile device and the cloudlet. As the results show, the difference is quite impressive. We can see that face detection in the cloudlet is not influenced significantly by the number of faces and remains almost constant at around 100 ms. Meanwhile the face detection increases almost linearly on the mobile devices. When the number of faces to detect in the picture is small, i.e. 4 or 8 faces, offloading yields 5 to 10 times faster execution time than the local execution on the phones, while when the number of faces increases further, i.e. 10 or 16 faces, offloading results in 10 to 16 times faster execution.

4. http://blogs.valvesoftware.com/abrash/latency-the-sine-qua-non-of-ar-and-vr/



(a) Download link scenario.



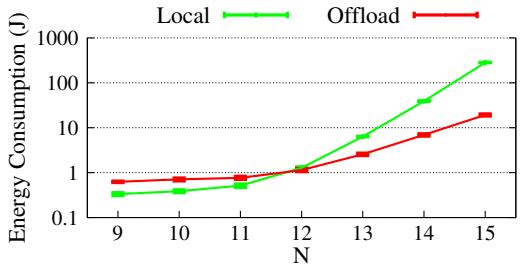
(b) Data transferring for different sizes.

Fig. 10: Transmission time between mobile devices and the cloudlet.

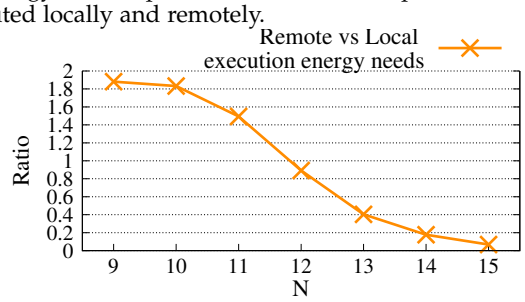
6.3 Transmission Time

In Figure 10a, we show the time needed for downloading a large amount of data in the mobile devices versus downloading the data through the cloudlet (the PC in this experiment). For this experiment, we stored files of 100KB, 1MB, and 10MB on Dropbox and used the "create link" functionality to download them. Although wireless protocols will reach speeds close to 1 Gbps shortly, currently there is a big difference when we compare the transmission differences between the PC (Ethernet connection) and the mobile device (WiFi). As the results show, downloading large files is faster when performed via offloading on the cloudlet. This can be motivated by the faster Ethernet connection on the PC compared to the mobile device's WiFi. Notice that after downloading of the files in the cloudlet (PC), we do not send them to the smartphone.

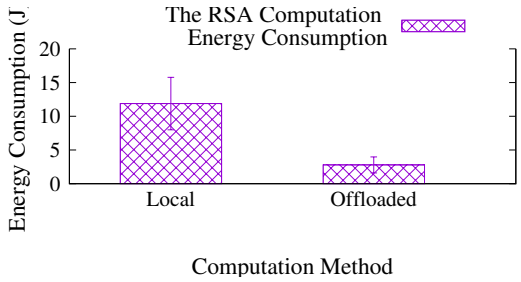
Figure 10b presents the delay overhead caused by the limited bandwidth of NFC. As in other offloading frameworks, also in our NFC-based system, the amount of data to transmit is a crucial factor because it can degrade the whole performance of an application if the time to send the data is significantly higher than the time needed to process them. The previous experiments show the benefits of NFC offloading to cloudlets for constrained devices. The processing capabilities of edge computing significantly overpass the specifications of current and near future mobile devices.



(a) Energy consumption of the N Queens puzzle solver when executed locally and remotely.



(b) Energy consumption when the N Queens puzzle solver is offloaded divided by the energy consumption when it is executed locally.



(c) The RSA energy consumption.

Fig. 11: Energy consumption of the N Queens puzzle and the RSA encryption algorithm.

6.4 Energy Consumption

To accurately measure the energy consumption on the devices during the experiments, we used the highly adopted Monsoon Power Monitor[5], which samples the power utilisation of the device with 5 KHz frequency. We use the Samsung Galaxy Note 3 for the power measurement since we need to remove the battery of the device in order to make it compatible with the Monsoon Power Monitor. Unfortunately, the battery of the Xiaomi Mi 3 is not removable, which physically limits the feasibility of the experiment on this device.

For the N Queens application, we measure the energy consumption for $N = \{9, 10, 11, 12, 13, 14, 15\}$ when running the task locally on the phone and when offloading it. For the latter setup, we use the Samsung Galaxy Note 3 as the main device and the Xiaomi Mi 3 as the offloadee device, since we are interested in the energy consumption of the main device. The results are shown in Figure 11a in a logarithmic scale. As expected, the energy consumption increases with the increase of N, but it increases slower when the computation is offloaded. Both curves depict the

5. https://www.msoon.com/LabEquipment/PowerMonitor/

average of 50 repetitions of each execution and the error-bars show the standard deviation. For small values of N (i.e. $N < 12$) the offloading is not very beneficial concerning the energy, but for $N \geq 12$ the benefit increases, and for the case of $N = 15$, the main device consumes 15 times less energy thanks to offloading. The gain on energy consumption is shown in Figure 11b, where we present the ratio of the energy needs when offloading over the energy needs when running the task locally.

We perform the same experiments for the RSA application, measuring the energy consumption on the Samsung Galaxy Note 3 of both the local computation and offloaded computation. Similar to the N Queens experiment, we use the Samsung Galaxy Note 3 as the main device and the Xiaomi Mi 3 as the offloadee device. We present the results of 50 measurements in Figure 11c, showing the average and the standard deviation. When the task is offloaded the main device consumes around 5 times less energy than when running the task locally. By looking at the results of these experiments, we argue that there is a significant benefit of using our framework to offload heavy computations via NFC.

7 Conclusions

In this paper, we proposed, designed, and implemented the first NFC offloading framework for Android devices. First, we proposed a new NFC communication protocol that circumvents the limitations of the default Android NFC protocol. Our protocol eliminates the requirement of user intervention, working automatically without requiring users to tap on the device's screen for data transfer. Furthermore, our protocol enables bidirectional communication between two devices, which paved the way towards building the NFC offloading framework.

We initially proposed delving into the Android NFC Stack and planned to modify and compile the Android source code directly. However, we wanted to find a solution that could be easily adopted by existing users without the need to modify the operating system of the device. Utilising the HCE service, although it is not designed for such interactions between mobile devices (it is mainly used in mobile payments and not for continuous peer-to-peer communications), we managed to remove the tapping requirement. With this service, not only could we provide a no-tapping offloading method, but we could also provide multiple data transmissions between two smartphones. Finally, we implemented the first known, to the best of our knowledge, NFC-based computation offloading framework between two smart devices.

Building on top of the newly designed NFC protocol, we presented the HCE-based computation offloading framework that requires no tapping and enables task offloading thanks to the multiple transfers between two NFC devices. We implemented four applications that use the framework to offload heavy computations from one *main device* to an *offloadee device*. We showed that when the offloadee device is more powerful, the execution time of the offloaded task is improved. The experiments show that the latency of the offloaded computation is almost equal to the latency of running the task locally on the offloadee device. Finally, we showed that the main device could reduce its energy consumption when offloading the computations, thanks to the low–energy consumption of the NFC interface.

8 Future Work

We observed several limitations in our framework, mostly due to the underlying technologies that NFC is built upon. The main problem we faced was the low bandwidth of the data transmission, with values around $15 - 25$ Kbps, which is caused by the existing hardware that allows only one message per connection to be transmitted. These limitations make the framework unsuitable for several types of applications, in particular, those that need to transfer some data during the offloading process. However, we believe that new NFC chips that can support multiple messages per connection will increase the bandwidth significantly and will broaden the applicability of NFC.

We investigated different techniques to increase the bandwidth and enable support a broader range of applications. Our next steps on this direction are to find techniques to increase the bandwidth and support a broader range of applications by complementing the NFC framework with parallel connections between the devices by using Bluetooth and/or WiFi–direct. Furthermore, we plan to extend our framework and make it more heterogeneous by supporting other operating systems and devices, such as tablets, laptops, and desktops with external NFC readers. Finally, we will enrich the current API to expose more functionalities to the developers and we will continue to implement more applications that make use of the framework.

References

[1] J. Newman, "Peak Battery: Why Smartphone Battery Life Still Stinks, and Will for Years," techland.time.com/2013/04/01/peak-battery-why-smartphone-battery-life-still-stinks-and-will-for-years/.

[2] S. D. Ltd, "FlashBattery for smartphones," http://www.store-dot.com/#!smartphones/zoom/c1w5t/c1u51, 2015.

[3] D. Borghino, "Nanodot-based smartphone battery that recharges in 30 seconds," http://www.gizmag.com/nanodot-smartphone-battery-30-second-recharge/31467/, 2014.

[4] R. Balan, J. Flinn, M. Satyanarayanan, S. Sinnamohideen, and H.-I. Yang, "The case for cyber foraging," in *Proceedings of the 10th Workshop on ACM SIGOPS European Workshop*, ser. EW 10, 2002, pp. 87–92.

[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," ser. MobiSys, 2010, pp. 49–62.

[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, 2011, pp. 301–314.

[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 *Proc. of IEEE INFOCOM*, 2012.

[8] C. Shi, V. Lakafosis, M. H. Ammar, and E. W. Zegura, "Serendipity: Enabling remote computing among intermittently connected mobile devices," ser. MobiHoc, 2012, pp. 145–154.

[9] D. Chatzopoulos, M. Ahmadi, S. Kosta, and P. Hui, "Have you asked your neighbors? a hidden market approach for device-to-device offloading," in *IEEE WoWMoM*, June 2016, pp. 1–9.

[10] R. Want, "An introduction to RFID technology," *Pervasive Computing, IEEE*, vol. 5, no. 1, pp. 25–33, 2006.

[11] V. Coskun, K. Ok, and B. Ozdenizci, *Professional NFC application development for android*. John Wiley & Sons, 2013.

[12] B. Hopkins, "Faster Data Transfer With Bluetooth and Contactless Communication," http://www.oracle.com/technetwork/articles/javame/nfc-bluetooth-142337.html, 2009.

[13] Y.-S. Chang, C.-L. Chang, Y.-S. Hung, and C.-T. Tsai, "NCASH: NFC phone-enabled personalized context awareness smart-home environment," *Cybern. Syst.*, vol. 41, no. 2, pp. 123–145, Feb. 2010.

[14] P. Smith, "Comparing Low-Power Wireless Technologies," http://www.digikey.com/en/articles/techzone/2011/aug/comparing-low-power-wireless-technologies, 2011.

[15] B. Ozdenizci, M. Aydin, V. Coskun, and K. Ok, "NFC research framework: a literature review and future research directions," in *The 14th International Business Information Management Association (IBIMA) Conference. Istanbul, Turkey*, 2010.

[16] E. Haselsteiner and K. Breitfuß, "Security in near field communication (NFC)," in *Workshop on RFID security*, 2006, pp. 12–14.

[17] H. Eun, H. Lee, and H. Oh, "Conditional privacy preserving security protocol for NFC applications," *IEEE Transactions on Consumer Electronics*, vol. 59, no. 1, pp. 153–160, February 2013.

[18] F. Dang, P. Zhou, Z. Li, E. Zhai, A. Mohaisen, Q. Wen, and M. Li, "Large-scale invisible attack on AFC systems with NFC-equipped smartphones," in *IEEE INFOCOM*, May 2017, pp. 1–9.

[19] F. Dang, P. Zhou, Z. Li, and Y. Liu, "NFC-enabled attack on cyber physical systems: A practical case study," in *IEEE INFOCOM WKSHPS*, May 2017, pp. 289–294.

[20] S. CRAWFORD, "How Microsoft Surface Tabletop Works," http://computer.howstuffworks.com/microsoft-surface2.htm, 2011.

[21] K. Sucipto, D. Chatzopoulos, S. Kosta, and P. Hui, "Keep your nice friends close, but your rich friends closer - computation offloading using NFC," in *IEEE INFOCOM*, May 2017, pp. 1836–1844.

[22] S. Bouzefrane, A. F. B. Mostefa, F. Houacine, and H. Cagnon, "Cloudlets authentication in nfc-based mobile computing," in *MobileCloud*. IEEE, 2014, pp. 267–272.

[23] T. Verbelen, P. Simoens, F. De Turck, and B. Dhoedt, "Cloudlets: Bringing the cloud to the mobile user," in *Proceedings of the third ACM workshop on Mobile cloud computing and services*. ACM, 2012, pp. 29–36.

[24] M. Satyanarayanan, Z. Chen, K. Ha, W. Hu, W. Richter, and P. Pillai, "Cloudlets: at the leading edge of mobile-cloud convergence," in *MobiCASE*. IEEE, 2014, pp. 1–9.

[25] D. SUTIJA, "NFC is the underdog tech set to explode in the next five years," thenextweb.com/contributors/2018/04/07/nfc-underdog-tech-set-explode-next-five-years/, 2018.

[26] L. Atzori, A. Iera, and G. Morabito, "The internet of things: A survey," *Computer networks*, vol. 54, no. 15, pp. 2787–2805, 2010.

[27] A. Alzahrani, A. Alqhtani, H. Elmiligi, F. Gebali, and M. Yasein, "NFC security analysis and vulnerabilities in healthcare applications," in *IEEE PACRIM*, vol. 302, 2013, pp. 27–29.

[28] E. Haselsteiner and K. Breitfuß, "Security in near field communication (NFC)," in *Workshop on RFID security*, 2006, pp. 12–14.

[29] D. Chatzopoulos, M. Ahmadi, S. Kosta, and P. Hui, "Openrp: a reputation middleware for opportunistic crowd computing," *IEEE Communications Magazine*, vol. 54, no. 7, pp. 115–121, July 2016.

[30] ——, "Flopcoin: A cryptocurrency for computation offloading," *IEEE Transactions on Mobile Computing*, vol. 17, no. 5, pp. 1062–1075, May 2018.

[31] N. Fernando, S. W. Loke, and W. Rahayu, "Honeybee: A programming framework for mobile crowd computing," in *Mobile and Ubiquitous Systems: Computing, Networking, and Services*. Springer Berlin Heidelberg, 2012, pp. 224–236.

[32] R. Montella, C. Ferraro, S. Kosta, V. Pelliccia, and G. Giunta, *Enabling Android-Based Devices to High-End GPGPUs*. Cham: Springer International Publishing, 2016, pp. 118–125. [Online]. Available: https://doi.org/10.1007/978-3-319-49583-5_9

[33] I. Zhang, A. Szekeres, D. Van Aken, I. Ackerman, S. D. Gribble, A. Krishnamurthy, and H. M. Levy, "Customizable and extensible deployment for mobile/cloud applications," in *USENIX OSDI*, 2014, pp. 97–112.

[34] M. S. Gordon, D. K. Hong, P. M. Chen, J. Flinn, S. Mahlke, and Z. M. Mao, "Accelerating mobile applications through flip-flop replication," ser. MobiSys, 2015, pp. 137–150.

[35] S. Bouzefrane, A. F. B. Mostefa, F. Houacine, and H. Cagnon, "Cloudlets authentication in NFC-based mobile computing," in *IEEE MobileCloud*, April 2014, pp. 267–272.

[36] S. Simanta, G. A. Lewis, E. Morris, K. Ha, and M. Satyanarayanan, "A reference architecture for mobile code offload in hostile environments," in *Software Architecture (WICSA) and European Conference on Software Architecture (ECSA), 2012 Joint Working IEEE/IFIP Conference on*, Aug 2012, pp. 282–286.

[37] A. Reiter and T. Zefferer, "Paving the way for security in cloud-based mobile augmentation systems," in *IEEE MobileCloud*, March 2015, pp. 89–98.

[38] J. Somers, "The N Queens Problem: A Study in Optimization," , 2015.

[39] M. Rouse, "RSA algorithm (Rivest-Shamir-Adleman)," http://searchsecurity.techtarget.com/definition/RSA, Nov. 2014.

[40] P. Jain, J. Manweiler, and R. Roy Choudhury, "Overlay: Practical mobile augmented reality," in *ACM Mobisys*, 2015, pp. 331–344.

Dimitris Chatzopoulos received his PhD in Computer Science and Engineering from The Hong Kong University of Science and Technology and his Diploma and Msc in Computer Engineering and Communications from the Department of Electrical and Computer Engineering of University of Thessaly, Volos, Greece. His main research interests are in the areas of mobile computing, device–to–device ecosystems and cryptocurrencies.

Carlos Bermejo received his MSc. degree in Telecommunication Engineering in 2012 from Oviedo university, Spain. He is currently a PhD student at Hong Kong University of Science and Technology working at the Symlab research group. His main research interest are Internet-of-Things, mobile augmented reality, network security, human-computer-interaction, social networks, and device-to-device communication.

Sokol Kosta received his Bachelor's, Master's, and Ph.D. degrees in Computer Science (summa cum Laude) from Sapienza University of Rome, Italy, in 2006, 2009, and 2013, respectively. In 2013–2014 he was a postdoctoral researcher at Sapienza University and a visiting researcher at HKUST in 2015. He is currently Assistant Professor at Aalborg University Copenhagen. He has published in several top international conferences and journals including IEEE Infocom, IEEE Communications Magazine, IEEE Transactions on Mobile Computing. His research interests include networking, distributed systems, and mobile cloud computing.

Pan Hui (IEEE Fellow and ACM Distinguished Scientist) received his Ph.D degree from Computer Laboratory, University of Cambridge, and earned his MPhil and BEng both from the Department of Electrical and Electronic Engineering, University of Hong Kong. He is currently a faculty member of the Department of Computer Science and Engineering at the Hong Kong University of Science and Technology where he directs the HKUST-DT System and Media Lab. He also serves as a Distinguished Scientist of Telekom Innovation Laboratories (T-labs) Germany and an adjunct Professor of social computing and networking at Aalto University Finland.He is an associate editor for IEEE Transactions on Mobile Computing and IEEE Transactions on Cloud Computing.