Towards the Implementation of an Energy Saving App State Migration Technique (ASMT)

THESIS

Presented in Partial Fulfillment of the Requirements for the Degree Master of Science in the Graduate School of The Ohio State University

By

Nathaniel Joseph Morris, B.S.

Graduate Program in Electrical and Computer Engineering

The Ohio State University

2014

Master's Examination Committee:

Prof. Xiaorui Wang, Advisor

Prof. Füsun Özgüner

Abstract

In the recent years, advances in mobile computing have transformed society's perspective regarding mobile devices such as smartphones. Consequently, a large percentage of the world's population has been influenced by the smartphone's ability to perform operations that were typically limited to general purpose computers. Since, the current development of mobile computing exhibits attributes similar to general purpose computing; the smartphone provides a limited but convenient and mobile solution to general purpose computing that fits in the palm of a user's hand. In result, by the end of 2013, statistics revealed that 56% of American adults own either an Android or iOS smartphone. Due to the increasing popularity of smartphones, the research community has devoted more time in optimizing mobile computing. Naturally, with any mobile device, the battery capacity determines the duration of usage. In fact, a 2013 survey expressed that 71% of 3500 smartphone users in the US, UK, Germany and UAE, desired a long lasting battery.

As a solution to the above stated problem, this thesis presents the design and implementation of a prototype "ASMT". The ASMT is a light-weight solution that seamlessly migrates the states of active applications between a smartphone and an Android Virtual Machine (VM) located on a desktop for energy savings.

This technique extends Android's NDEF protocol as well as its NFC service to minimize the performance impact.

In conclusion, this thesis verified that the ASMT maintained a small performance footprint. Meanwhile, external work demonstrated that this technique on a smartphone produced a minimum of 12% energy savings and 64.4% on average.

Dedication

This document is dedicated to my family.

Acknowledgments

I would like to express my sincere gratitude to organizations as well as individuals who supported my academic endeavors at The Ohio State University. Most importantly, I appreciate all of the guidance from my advisor Dr. Xiaorui Wang over the past two years. Moreover, with the support from the Ohio Space Grant Consortium (OSGC), I had the opportunity to pursue graduate school without financial concerns. Of course, I cannot overlook Dr. Abayomi Ajayi-Majebi efforts in supporting my continuation in higher education and being an awesome mentor. Recently, I came in contact with the staff of the Minority Engineering Program (MEP) at The Ohio State University who cheered me on in finishing my Thesis and providing me with financial relief during the summer of 2014. In addition, I would like to thank my partner Bruce Beitman (Ph.D. Student) for guidance on Android Open Source Project (AOSP) and my Thesis. Lastly, I would like to acknowledge my father's unconditional support as a parent and advisor.

Vita

May 2007 .....Trotwood Madison High School

2012.....B.S. Manufacturing Engineering, Central State University

2012 to present .....Graduate Fellow, Department of Electrical & Computer Engineering, The Ohio State University

Fields of Study

Major Field: Electrical and Computer Engineering

Table of Contents

List of Tables

List of Figures

Chapter 1: Introduction

The software technique in this paper relies on app state migration between smartphone and desktop in order to conserve energy for Android smartphones. This technique allows users to interact with their smartphone through an Android virtualization on a desktop. Meanwhile, the smartphone reduces energy consumption by transitioning into a low power state. Resultantly, the user remains connected to their smartphone through desktop interaction and also reduces energy consumption that would otherwise not be conserved. This paper details the design and implementation of the ASMT as well as presents performance evaluation results. The first prototype discussed in this paper serves as the proof of concept while the second prototype described is the final prototype that satisfies system requirements and project goals.

1.1 Problem Statement

The modern smartphone consumes a significant amount of energy due to its technology density. Consumers expect their smartphone's battery life to sustain appropriate levels for operation throughout the day. Hence, longer operation times are necessary for practical usage and customer satisfaction. A software technique should be developed to increase the energy savings for Android smartphones. This software technique must possess the following attributes; low overhead, manual or automatic triggering, and have a gentle learning curve for software programmers.

1.2 Android Smartphone System Requirements

The system requirements for the Android smartphone and desktop are not rigid and majority of smartphones and desktops on the market can meet the minimum or exceed the following requirements discussed. The first requirement defines the operating system version while the second specifies the wireless communication protocol. The third and last requirement defines the specifications for the desktop.

The software technique developed in this paper requires the Android OS to include near field communication's (NFC) application programming interfaces (API) and services. The NFC framework provides a sufficient foundation for the proposed energy conservation technique. The Android operating system (OS) introduces Android Beam, a NFC feature allowing the convenient short-range exchange of data between smartphones, in Version 4.0 (Ice Cream Sandwich) of the Android OS. Therefore, version 4.0 or later of the Android OS establishes the first well defined requirement.

The smartphone's hardware requirement only specifies the type of wireless communication. Android smartphones typically include a combination of these wireless communication types: Bluetooth, cellular network, NFC, and WI-FI. The type of wireless communication directly impacts the effectiveness of the software technique. The wireless technologies' attributes reviewed and evaluated are communication range, bandwidth, and energy consumption. However, due to the small data packets transmitted/received during synchronization, this fact relaxes the bandwidth requirement. In other words, for

any type of wireless communication the bandwidths provide sufficient data rates. Bluetooth proves to be an attractive wireless technology. The maximum power permitted depends on the Bluetooth Class. There are Classes 1, 2, and 3 with corresponding maximum power 100, 2.5, and 1 milli-Watts (mW) respectively and communication ranges 100, 10, 1 meters (m) respectively [1]. In addition, the most common radio uses Bluetooth Class 2 [1]. Since majority of desktops are not equipped with Bluetooth and also limited to a 10 m range, this wireless technology appears not sufficient. Cellular network's energy consumption varies significantly and typically not equipped in desktops. The variability of the cellular network's energy consumption and the incompatibility with desktops eliminate this wireless technology from consideration. NFC, the next wireless communication type consumes 15 mili-Amps (mA) or less and has a communication range less than 0.2 m [2]. The communication range labels NFC impractical for the software technique detailed in the following chapters. The final wireless communication type, WI-FI, consumes the most energy out of the previous wireless communication types. WI-FI's maximum power of 1 Watt (W) permits an average communication range of 46 m [3]. The additional communication range better suits the synchronization between smartphone and desktop. Especially, since most desktops and all laptops are equipped with WI-FI. However, the significant power consumption forces one to reconsider Bluetooth as the optimal wireless communication type. After a fine granular comparison of the two wireless communication types, WI-FI's higher compatibility with desktops and laptops rules out Bluetooth. In conclusion, WI-FI establishes the second well defined requirement. The third requirement details the

desktop's software and hardware specifications. The desktop's software requirements consist of the OS and the x86 virtualizer. VirtualBox satisfies the x86 virtualizer requirement and inherently governs the OSs supported. VirtualBox supports many OSs, such as Windows XP and later, Linux distributions, Mac OS X, Solaris, and OpenSolaris [4]. In fact, Windows 7 Ultimate provides the greatest user familiarity and flexibility due to the percentage of desktops and laptops with it installed [5]. Hence, Windows 7 Ultimate establishes the OS requirement. The hardware requirements detail the minimum specifications for memory, disk space, and processor. Desktops or laptops must contain at least 1 GB of physical memory and 16GB or more of available hard disk space [6]. Any recent processor from Intel or AMD satisfies the processor requirement.

Chapter 2: Background

Chapter 2 provides information on topics relevant to the design and development of the software technique presented in this paper. Furthermore, topics such as smartphones, Android, power management techniques, migration and synchronization are discussed. Next, a literature review introduces other research similar to the software technique developed in this paper. Lastly, the motivation of this paper is described.

Smartphones continue to evolve by including more features equivalent to other popular electronic devices. For example, modern smartphones incorporate features from personal digital assistants (PDAs), media players, digital cameras, and GPS navigation units to touchscreen computing, web browsing, WI-FI, and third party apps [7]. Conversely, a basic mobile phone incorporates features only essential to mobile communication. Bare minimum features for such mobile devices typically contain features relevant for executing wireless calls and short message service (SMS) text messages [7]. A limitation on the number of features for a device reduces the total energy consumption. Meanwhile, mobile devices such as smartphones include more features and consequently consume more energy. Hence, the research in this paper focuses on an energy conservation technique that improves operation time.

In order for users to interact with a smartphone, the device must contain an OS. Depending on the manufacturer of the smartphone, the OS installed varies.For instance, Google's Android and Apple's iOS constitute the two popular OSs available on the market. In particular, the Android OS is open source and relies on a modified Linux kernel primarily for touchscreen mobile devices [8]. Consequently, an open source OS allows streamlined research with community support that proprietary OSs cannot offer. As a result, this research focuses on the Android OS. The Android OS provides a user interface based on direct manipulation using touch inputs such as tapping, swiping, pinching, and reverse pinching [8]. Touch inputs allow the user to navigate and manipulate objects on the screen. From a hardware perspective, the OS's most popular hardware platform is the 32-bit ARMv7 architecture [9]. The minimum hardware requirements for Android version 4.4 are a 200 Mega-Hertz (MHz) ARMv7 processor with 512 Mega-Bytes (MB) of Random Access Memory (RAM) [9]. Also, Android requires at least 32 MB of flash memory.

Migration and synchronization are techniques capable of data and application transfer. Specifically, personal computer (PC) migration transfers the entire user environment between two computers [10]. The user environment includes user and registry data and also applications. If the OSs on both computers differ, then the pc migration may fail due to incompatibilities [10]. Such incompatibilities include registry data, OS version, and the type of OS. However, under the circumstance that requires the complete transfer of the user environment to another computer with similar software and hardware specifications;

PC migration becomes a suitable solution. Conversely, synchronization simply maintains user's data integrity between two devices [10]. In other words, data synchronization keeps copies of data in coherence with both devices. Normally, one of the devices is a computer while the other device ranges from PDAs to smartphones.

2.1 Related Work

Recent research has explored potential energy savings for both smartphones and desktops. One of the most effective approaches to energy conservation for smartphones is the transfer of computational load to a cloud server [11, 12]; this process is commonly known as offloading. Computation offloading alleviates restrictions due to limited resources, such as battery life, network bandwidth, storage capacity, and processor performance [12]. Offloading techniques may be performed on workloads constituting the following: method procedures [13], tasks [14], applications [15], or virtual machines [16]. Workloads of these types are processed in the cloud server, such that the smartphone experiences energy savings. Current offloading techniques require intricate algorithms in order to make offloading decisions to improve performance or save energy. These decisions are based on the analysis of parameters including bandwidths, server speeds, available memory, server loads, and the data traffic between servers and mobile systems. Even though offloading shows promise, current offloading techniques focus only on computational energy for smartphones. Thus, the smartphone continues executing local tasks, while the wireless radios WIFI and 3G are receiving the results from the cloud server. In result, cloud offloading is an effective approach for

computational energy savings but does not consider other perspectives of energy savings. Such perspectives may involve the hardware's power states and also the local tasks.

Currently, engineering literature has not explored energy savings through application state migration between a smartphone and a desktop. The following will discuss work closely related to the research in this paper. Recent studies pertaining to energy savings focused on the energy consumption in power states or the energy consumption for particular scenarios. Carroll et al. express the significance of suspended and idle states for a smartphone [17]. Specifically, in the suspended state the application processor is idle, while the communication processor remains active to receive calls. Conversely, the idle state is fully awake with no active applications. In addition, Shye et al. reveal that the suspended state consumes less energy than the active state. Regardless of the discovery, the user spends majority of their time in the suspended state, consuming on average 49.3% of the total system energy [18]. This research focuses on minimizing the energy consumed in the suspended state and reducing time the user spends in the active state. Moreover, research has suggested transitioning idle desktops to a sleep mode while maintaining network presence through live migration of virtual machine (VM), solves wasted energy for idling desktops [19,20,21]. LiteGreen presents a system that uses VM live migration to save energy during idle periods. In particular, LiteGreen virtualizes the user's desktop environment as a VM and migrate it between the user's desktop and a VM server. LiteGreen's technique closely parallels the energy conservation technique in this paper with one major difference. LiteGreen requires a VM server to transfer VMs and

execute migrated VMs. On the contrary, the software technique in this paper synchronizes application states between the desktop and smartphone with insignificant overhead.

2.2 Motivation

In the recent years, advances in mobile computing have transformed society's perspective regarding mobile devices such as smartphones. By the end of 2013, 56% of American adults own either an Android or iOS smartphone [22]. Due to the increasing popularity of smartphones, the research community has devoted more time and effort optimizing mobile computing. Naturally, with any mobile device, the battery capacity determines the duration of usage. In fact, a 2013 survey expressed that 71% of 3500 smartphone users in the US, UK, Germany and UAE, desired a long lasting battery [23]. Hence, energy conservation and the extension of battery life are topics of interest.

Research concurrent to this thesis and performed by PhD student Bruce Beitman shows the average energy savings in multiple scenarios. The Galaxy Nexus smartphone's idle power was measured in three cases. These cases include each of these scenarios: only baseband, WI-FI and baseband, and 3G and baseband. The three cases were repeated to simulate the least amount (worst user case) and average amount (average user case) of energy savings. Baseband is the wireless radio responsible for receiving calls and SMS text messages. Beitman's results indicated that energy savings of 12.2% in the worst case and 28% on average can be achieved [24]. However, with complete smartphone migration, the energy used by the smartphone during user interaction is eliminated. As a

result, application state migration has a potential energy savings of 69% [24]. The research presented in this paper specifically focuses on the implementation of this app state migration technique.

Chapter 3: Software design

Chapter 3 discusses the software design of two distinct prototypes. The first prototype design primarily serves as a proof of concept. The second prototype design implements the app state migration software technique into the physical smartphone and desktop. The end of this chapter concludes software design and transitions into chapter 5, which discusses software implementation.

3.1 First Prototype

The first design focuses on proving if application state migration is possible. In other words, this prototype represents the proof of concept for application state migration. The proof of concept must achieve certain outcomes in order to declare it a success. A successful outcome consists of an effective application state migration between two VirtualBox Android instances. The migration protocol utilizes the Android Debug Bridge (ADB) command line tool in order to synchronize application states. After the completion of the first prototype, the results and insights are incorporated into the more practical second prototype.

3.2 Development Environment

Any software project has an environment that aids in productivity and development. The development environment for this research is not an exception. In general, the development environment consists of environments for programming, debugging, and

performance evaluation. The setup procedures and settings for each environment are described in detail.

The programming environment serves as the workspace for program design and debugging. Such environments may support one or more programming languages based on the particular integrated development environment (IDE). The internet contains a vast selection of IDEs that are available commercially or as open source. Naturally, for a small project, an open source IDE suffices. Choosing a suitable open source programming IDE proves not difficult, since the native language for Android's applications and operating system is JAVA. Consequently, the first prototype is programmed in JAVA. In the programming community, Eclipse IDE remains one of the most popular JAVA IDEs. Also, Eclipse seamlessly integrates Android application development into an intuitive environment suitable for this research. Therefore, Eclipse offers a flexible programming environment containing compatible features with Android development that complements the first prototype.

The first step in setting up the programming environment starts with the installation of Eclipse IDE. The next few steps configure Eclipse's environment in order to provide tools and interface features for Android application development. Eclipse provides an interface for adding third party plugins such as Android Development Tools (ADT). This plugin extends the current environment to permit quick set up of Android projects and offer convenient tools to build app UI, debug your app, and export signed app packages

for distribution. The final package necessary to complete the programming environment is the Android Software Development Kit (SDK). Android's SDK provides developers with the API libraries and platform tools necessary to build, test, and debug applications for Android. The platform tools consist of the debug loggers, Android Debug Bridge (ADB) command line, and the ARM emulator as well as other tools ranging from performance analysis to stress test. Android's developer website offers an ADT bundle that includes everything one may need for application development. In conclusion, the setup procedure for Eclipse may be somewhat involved. However, Android's developer website provides the necessary installation and setup procedures for a user to have an operational programming environment suitable for Android application development. For more details regarding Eclipse installation and setup, refer to the Android developer website.

Debugging software significantly reduces development time by assisting the programmer with identifying errors throughout their code. Fortunately, Eclipse helps with identifying typical compiler and syntax errors. However, Eclipse fails in the case when the code compiles successfully, but the programmer is trying to debug their code logic. A logging system that conveys the states of interest to the programmer remains the most popular technique for runtime debugging. The typical technique for runtime debugging relies on the deployment of the program onto a physical device. The programmer may not have access to a physical device every time debugging is necessary. Hence, one solution depends on the deployment of the program on a virtual instance of the OS. Android's

SDK precisely provides such a solution that emulates the ARM processor on x86-64 based personal computers (PCs). In addition, Google's Android provides a distribution that supports X86-64 architectures. Therefore, Android can be installed on x86-64 machines or be virtualized on a Windows, Mac, or Linux PC using virtualization software without the drawbacks associated with emulation. Naturally, emulators suffer performance degradation due to the additional logic necessary to execute code on non-native execution environments. In terms of performance and convenience, virtualizing Android on a PC offers the ideal environment for runtime debugging without a physical device. The deployment of a programmer's application on a particular instance of Android is the first step towards runtime debugging. The logging system for Android provides a high level interaction between programmer and application that is sufficient for debugging. The Android OS includes a logging system called "logcat". Logcat filters and views logs collected by multiple circular buffers. The programmer writes to the circular buffers by invoking the "Log" function at points of interest throughout their code. Consequently, logcat offers the means for the programmer to debug application on the Android OS regardless if the platform is an emulator, virtualizer, or physical device.

In regards to performance, the first prototype does not consider performance impact but reveals the level of application state migration as its goal. However, the second prototype includes performance as one of its goals and evaluates the performance on a physical smartphone. Therefore, this section briefly describes the types of Android platforms and performance metrics considered. The performance of software on a mobile device must

have the means to be evaluated in order to measure the impact on system resources. As mentioned in the previous sections, Android can run on multiple platforms: a physical device (ARM architecture), emulator, or virtualizer. Obviously, measuring performance on an emulator results in erroneous data due to emulation inefficiencies, unless relative performance is attainable. Since, measuring relative performance requires a baseline evaluation and consequently more implementation; an optimal solution for performance evaluation is discussed in detail within the second prototype section. Specifically, the performance metrics of interest are execution time, memory allocation, and energy consumption. In section 3.2, the definition and implementation of each performance metric is explained in detail.

3.3 Design Concept

The first prototype must achieve a certain level of migration in order to further develop and implement an improved technique. Consequently, this section describes the rubric used to qualitatively measure the level of migration as well as discuss the design algorithm for the first prototype. For any details on the migration software's code, refer to Appendices A and B at the end of this paper.

The rubric is a simple user-centric evaluation tool that determines if the first prototype achieves its defined goals as depicted in Table 1. In essence, the user observes the type of data from each application to determine how well the available data describes the current state of an application. For instance, if the data available for the applications are characterized as only persistent data, then significant application state migration is not

plausible. Conversely, conditional data governed by the current state of the application, provides a greater level of application state migration. Depending on the required level of migration, it may be desirable to control the level of migration by implementing specific protocol for application states during development. The latter allows the level of migration to be controlled based on the software developers providing persistent data that is highly correlated to application states. Hence, using this basic evaluation tool to determine the frequency of types of data from a sampling pool provides useful statistics in determining the best way to provide application states. Consequently, the first prototype can be evaluated appropriately to obtain valuable results for the creation of the second and final prototype. In conclusion, the purpose of the first prototype is to reveal the types of data associated with applications and their effects on migration.

Data Type	**Location**	**Potential Level of Migration**	**Action**
Persistent	SD Card	Low	Developer Implementation
Conditional	SD Card / RAM	Medium - High	No Action / Developer Implementation
Developer Created	Defined by Developer	Controlled Level	No Action

Table 1. The evaluation tool (rubric) for the first prototype

The remaining of this section details the design of the first prototype from a high level of abstraction to the specific workings, such as program logic and protocol. Figure 1 illustrates the highest level of abstraction of the system. In particular, this prototype is designed and implemented on a Windows X64 OS desktop in order to expedite the

prototyping process. In other words, all experimentation will occur on a Windows machine and Android X86 VM. The migration software follows a server-client model in terms of communication and command request. The current system implements the migration server on a Windows machine, whereas the Android smartphone and client resides on the Android VM1 and Android VM2 respectively. On the server side during initialization, the migration software executes the configuration routine in order to specify critical parameters before entering the main program. After initial setup, the server enters the main program and listens for commands over Transmission Control Protocol/ Internet Protocol (TCP/IP) network from the client on the Android VM. The server executes all commands received from the client, which in turn begins the migration process. Once the client finishes requesting commands, the server remains active until another migration begins. For instance, the instantiation of another migration from the client repeats the same process as previously described from the beginning of the server's main program. In regards to the client, it also requires a one-time configuration before entering the main program that triggers the application state migration. Unlike the server, the client terminates all operations at the end of migration. Consequently, each migration instance in this system requires separate triggering. In general, the network communication for both the server and client implements a basic hand-shake protocol to signify various states such as successful transmission, busy, and ready.

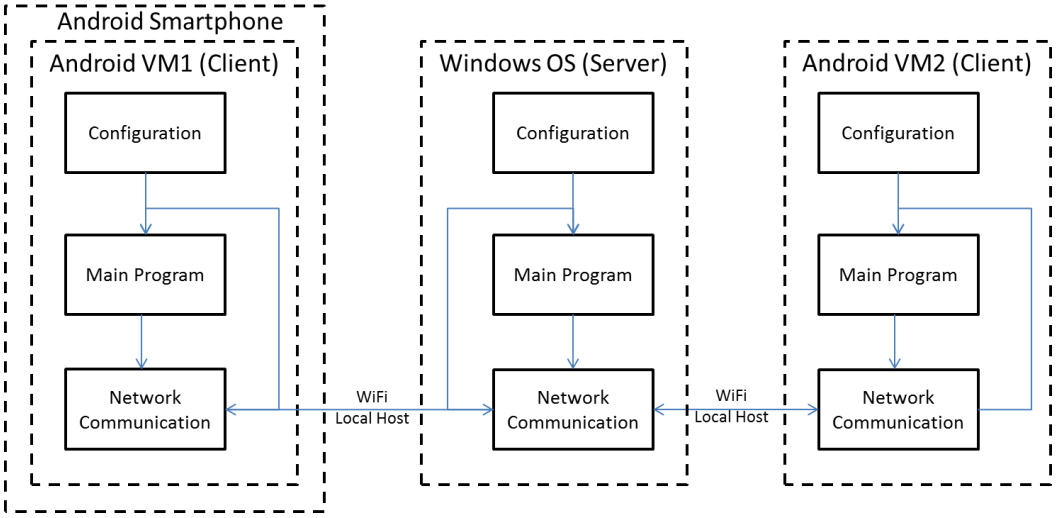

Figure 1. The system diagram of the ASMT (Proto 1)

This next section describes the design of the first prototype at a lower level of abstraction. The blocks in Figure 1 are dissected into six individual flowcharts in order to discuss the migration software for the server and client in detail; each block in Figure 1 is finely parsed to reveal critical blocks of logic. The following figures pertaining to the server are discussed first, while the figures for the client close the discussion. Since, the Android VM1 and Android VM2 have identical structure this paper will only cover the flowcharts for Android VM1.

Using a top down approach for each block in Figure 1, the server's configuration is the first block discussed in this section. The server's configuration in Figure 2 provides essential parameters and invokes the network setup for the server's main program. Consequently, the initialization provides the means for the server to communicate with the client and vice versa. Specifically, these parameters consist of an IP address for both

of the Android VMs and a port number. The network setup creates the socket and binds it to a specific port number. Hence, the server listens to the socket for the client to make a connection request [25]. The IP Addresses serve as device identifiers in order to route data packets to the correct VM during network data exchange. Once the user starts the server, a configuration window appears asking for the IP addresses for the VMs and the port number. In order to accept the parameters, the user presses the "OK Button". However, before the parameters are used for the purpose of data routing and socket creation, the parameters must pass a format check. As depicted in Figure 2, if the format check fails for any one of the parameters, the configuration window appears again asking for the re-submission of the information. Conversely, successful format verification for the IP addresses and port number allows for the continuation to the next sequential block, the server's socket. At this point in the Figure 2, the following sequential blocks detail the process required to physically prepare communications over the network; the process creates and binds a socket to the specified port number to allow the acceptance of incoming connection request. After successfully binding the socket to the port number, the server waits for a connection request from the client as illustrated in Figure 2. Once the server secures a connection, it transmits a status byte with the decimal value of *2* to the client. Table 2 specifies the command and status bytes associated with the server.

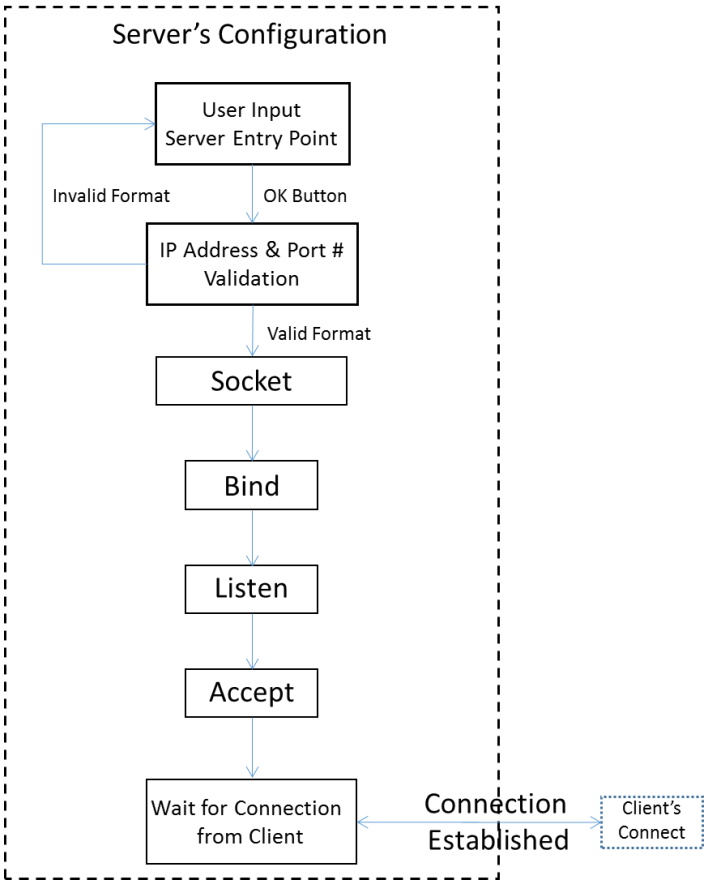

Figure 2. The lowest level of abstraction for the server's configuration logic

According to Figure 1, the next sequential block to discuss in detail is the server's main program. In particular, the server's main program only determines the next logical state based on status and command bytes. Furthermore, the main program does not have any direct involvement with the transfer of data over the network. However, the latter block in Figure 1 directly manages the data transmitted and received from the network, and further details will be addressed later in this paper. In general, the server's main program handles the interpretation of the client's command and status bytes in order to issue migration tasks between both of the VMs and the termination of the server respectively.

The server performs predetermined tasks based on a byte formatted command from the client. As illustrated in Figure 3, a command from the client with a decimal value of *1* triggers the migration for any application not present on the target VM as well as the current states for three Android applications: Health Tracker, Translate, and Block Puzzle. The migration process utilizes Android Debug Bridge (ADB) command line tool to copy database files and install apk files (applications) onto the VM being migrated to. In addition, a status byte from the client with a decimal value of *4* terminates the server's main program loop and the server transitions into a standby state waiting for future connection request. After, the server completes all task associated with application state migration, the server transmits a status byte with a decimal value *3* to the client. These commands and status bytes serve the purpose of a simplified handshake protocol in order to guarantee thread and platform synchronization.

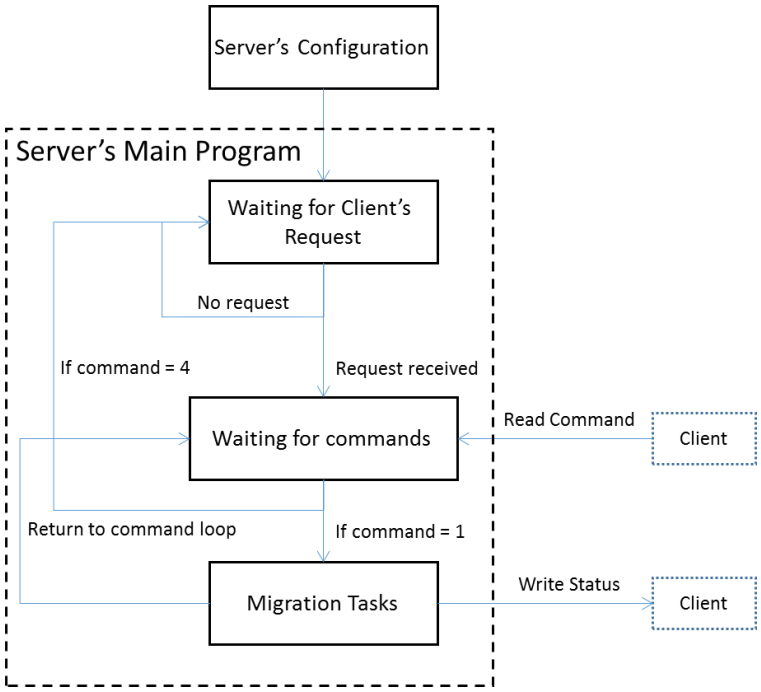

Figure 3. The lowest level of abstraction for the server's main program logic

Type of Byte	Byte-Decimal	Direction of Data Flow	Function
Command	1	Received	Trigger Migration
Status	2	Transmitted	Connection Established
Status	3	Transmitted	Server's Tasks Completed
Status	4	Received	Transition Server to Standby

Table 2. The functions of each type of byte associated with the server client

The final sequential block in Figure 1, the server's network communication block, reveals the details involved in communicating between the server and client. The network communication's logic occurs at the beginning of the software for initialization and at the end of the software during communication. Hence, the initialization of the network communication proceeds by binding the socket to the given port number as mentioned in the explanation for the server's configuration section. In detail, the server's configuration

creates a socket with *ServerSocket (port #)* method, binds it to a specific port, and accepts incoming connection request with the *accept()* method, which blocks until a connection occurs. In addition, the initialization process includes the creation of an *InputStream* and *OutputStream* Input/Output (I/O) objects. These objects allow for reading byte based data (InputStream) and writing byte based data (OutputStream) from and to the server's socket. The previous statements are depicted in Figure 4, where the logical flow of the network communication can be visualized. When a successful connection transpires between the server and client, the server waits for incoming commands or status updates or actively sends status updates via the bound socket. The server's network communication sends byte encoded data utilizing the *write()* method and receives byte encoded data with the *read()* method.

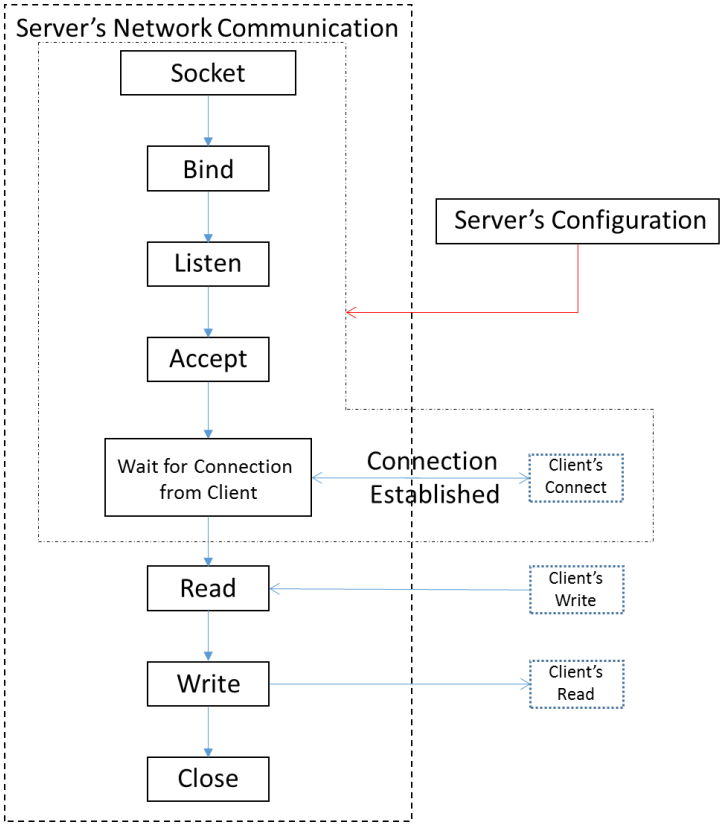

Figure 4. The lowest level of abstraction for the server's network communication logic

In regards to the client, the content will be discussed in the order it appears in either block titled "client" in Figure 1: client's configuration, client's main program, and the client's network communication. Overall, the client's migration software parallels the inner workings of the server's. Therefore, throughout this discussion certain sections' details for the client are similar enough to server to be ignored. However, an in depth explanation can be found in the server section. One of the major differences between the server and client's software is the targeted platform. In other words, the server's software is compiled for Windows X64 OS and the client's software for the Android OS. The

other differences between the server and client are minuscule enough to conclude their logic to be nearly identical.

The client's configuration begins when the user invokes the "Migration App" from the Android VM screen, as depicted in Figure 5. Once the client's migration software is actively running in the foreground, the migration requires two parameters before proceeding. Both of these parameters define the properties for the client's socket. Specifically, the client's configuration requires the corresponding IP address and port number for the server's socket. After the user presses the "Live Migration Button", both the IP address and port number are passed to logic that checks the format. If either parameter, IP address or port number fails the format check, the client's software reverts to the beginning and asks for resubmission of the parameters. Conversely, successful validation of both parameters' format allows the client's software to continue its initialization process. The initialization process entails the creation of a socket utilizing the IP address and port number provided by the user as well as the establishment of a network connection with the server. For the client's configuration, the network connection occurs immediately after the creation of the client's socket, unless the server happens not to be active. If the server does not respond, the client's *connect()* method results in a timeout and starts the configuration process from the beginning. Otherwise, the client establishes a connection to the server and the flow of bidirectional data begins.

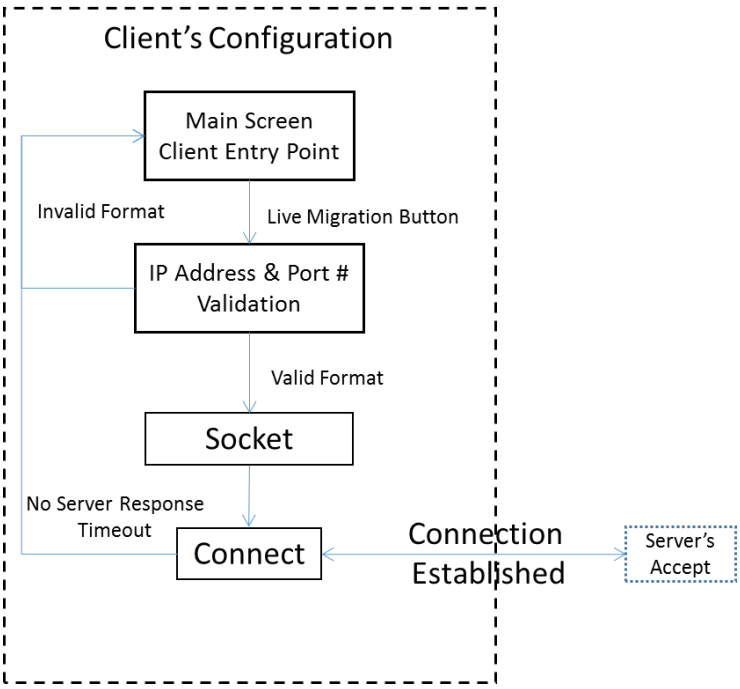

Figure 5. The lowest level of abstraction for the client's configuration logic

The client's main program primarily functions as a command center that determines when the migration of applications transpires, as seen in Figure 6. For example, the client's main program sends a single byte encoded command to the server in order to invoke the migration process. This one command triggers the server to completely migrate all applications not present on the targeted platform as well as the applications states for three applications: Health Tracker, Translate, and Block Puzzle. After the client and server establish a network connection, the client's main program waits for the server to send the status byte with decimal value of *2* before proceeding to the migration logic. Once the client receives a status byte with a decimal value of *2*, the client's main program transmits the byte command *1* to the server in order to invoke the migration process. At this point, the client's main program transitions into a standby state waiting for the server

to send the status byte *3*, which signifies that the server cleared its busy flag. The migration is complete when the client's main program receives a status byte of *3*. Consequently, the client's software restarts from the beginning, waiting for the user to trigger another migration with the same or different IP address and port number. For more detail on each command and status byte, Table 2 lists the details such as function for each one.

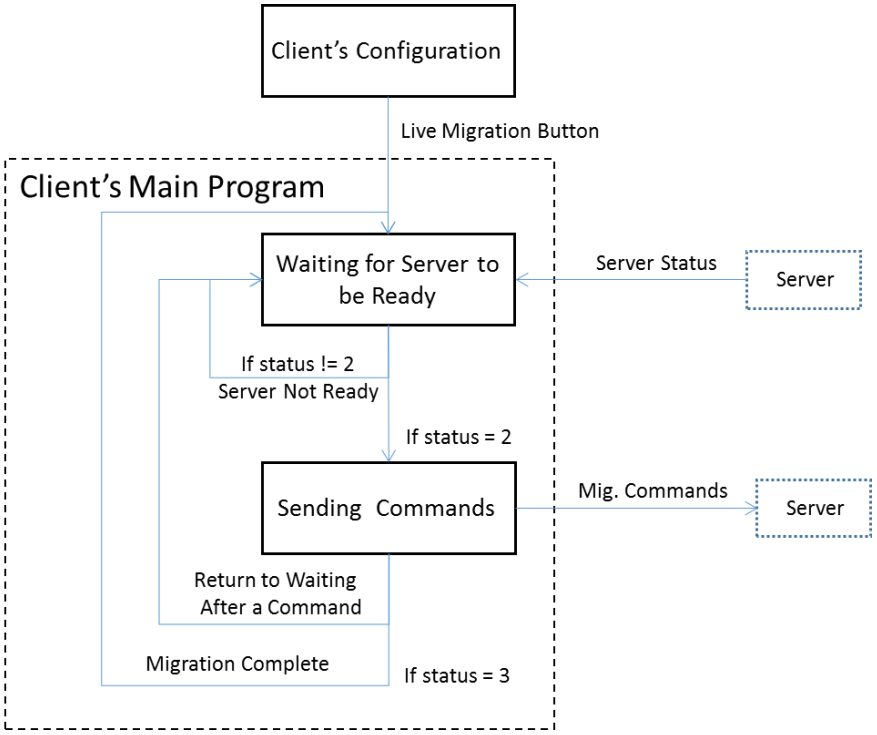

Figure 6. The lowest level of abstraction for the client's main program logic

In terms of complexity, the client's network communication logic requires less explanation in order to understand its functionality. Especially, since the client's configuration only creates a socket with the correct parameters and connects to the

server's socket. Walking through the flow of logic in Figure 7, the client's network communication begins with establishing a connection between the client and server. Moreover, if the server does not respond to the client's connection request, then the client blocks any further execution until a connection occurs. Therefore the initialization process includes the creation of the client's socket with the *Socket (port #)* method, binds it to a specific port, and connects to the server's socket with the *connect()* method, which blocks until a connection occurs. In addition, the initialization process includes the creation of an *InputStream* and *OutputStream* Input/Output (I/O) objects. These objects allow for reading byte based data (*InputStream*) and writing byte based data (*OutputStream*) from and to the client's socket. When a successful connection transpires between the client and server, the client waits or actively transmits status updates via the bound socket. The client's network communication sends byte encoded data utilizing the "write ()" method and receives byte encoded data with the "read ()" method.

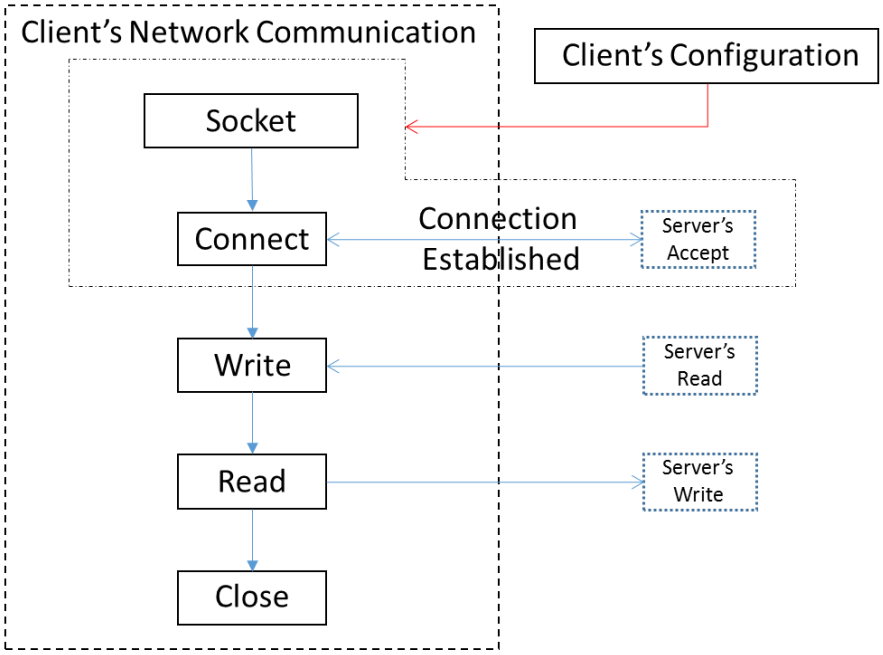

Figure 7. The lowest level of abstraction for the client's network communication logic

The previous discussions explained the design of the first prototype and the functionality of major blocks of logic associated with this migration technique. Therefore, the design of the server and client's main components were expounded upon; that is the configuration, main program, and network connection. Any detail left out can be supplemented by the Appendices A and B found at the end of this paper.

3.4 Results

The first prototype serves as a proof of concept for the implementation (second prototype) of the migration technique on the Android OS. Consequently, a proof of concept typically will not have quantifiable results due to the lack of preparation for the experimental design. However, most prototypes contain qualitative data resulting from human observation and/or the rubric defined earlier in this paper. Thus, the user-centric

rubric is used to acquire data relevant to the potential level of migration based on different data types. Based on observations, most applications lack the data relevant to application states in their database files and cannot contribute effectively to the migration.

3.5 Implications for Second Prototype

Based on the observations, the second prototype will produce the highest level of migration when the software developers provide the application states explicitly as dynamic persistent data. In this case, migrating application states between a smartphone and desktop for energy savings can provide consistent behavior as well as practicality.

Chapter 4: Second Prototype

The second prototype builds upon the shortcomings and implements observations realized from the first prototype. In other words, the second prototype presents a noticeably improved migration technique that is well constructed. This prototype performs migration of cached application states between a physical Android smartphone and the VM located on desktop with Windows OS. However, the analyses in regard to the second prototype focus on the migration between two VMs on the same desktop. Cached applications are applications that were recently active in the foreground and typically remain in the background. The state of an application encompasses the current window open, task being performed by the user, text entered, and etc. In comparison to the first prototype, the second one does not rely on Android's ADB command line tool. In fact, the second prototype implements the migration technique in Android's OS. Consequently, the second prototype has two major blocks in the system diagram, since the server exists on both platforms involved with the migration, (see Figure 8). As a result, there is no need for a third system diagram block for housing the server's logic. In the end, this paper will discuss the future endeavors in regards to the expansion of this technique and any challenges encountered.

4.1 Development Environment

Since the development environment for both prototypes have many commonalities, this section only focuses on the differences within the programming and performance evaluation environments. The following will discuss the details involved with the setup procedures and functionalities for both environments. Unlike the section 3.2, the performance evaluation environment consists of real experiments performed on an Android VM, which provides execution delay for major blocks of logic and memory allocation.

As mentioned in section 3.1.1, the programming environment provides the workspace and tools to effectively develop software. In terms of Android application development, the Eclipse IDE is sufficient for the second prototype; which the first prototype uses this environment as well. However, the compilation of Android's OS requires other tools in order to successfully build a custom OS for the VM or smartphone. The compilation of Android's OS may occur on a Windows or Linux machine. For this paper, an Ubuntu machine possesses the most community support in regards to Android and is deemed as the native environment for compiling the Android OS. Hence, the modifications for the energy conservation migration technique take place on Ubuntu. Before beginning development for Android, the local work environment requires Android's source code and the initialization of environment variables as well as the installation of software tools. The installation of packages such as Java, Python, Make, and etc. are necessary to correctly initialize the local environment on Ubuntu. Instructions can be found on

Android's Android Open Source Project (AOSP) website under the section labeled "Source" explaining how to download Android's source code and initialize the local environment [26]. The same is true for building Android's OS for a targeted platform and installing that custom OS on that platform.

The performance evaluation environment makes it possible to measure overhead and memory allocation induced by the implemented technique. There are various methods that can be used to obtain relevant performance data. If taking the experimental route, timestamps are inserted before and after the block of code in order to measure execution time. In regards to memory allocation, the system's memory usage is recorded before the execution of the software, as a baseline. Then, the memory usage is monitored during the execution of software to calculate memory consumption over time. Conversely, a modeling approach relies on the development of a system model that simulates the behaviors of the variables of interest. As a result, the system model may be exactly or closely the same as the lower levels of abstraction in the flowcharts. Furthermore, each block of distinctive logic in the model will behave as a function with multiple inputs and multiple outputs. In terms of the system inputs, these inputs are randomly generated to provide the system with the number of applications and the size of their migration packets. The final outputs from the system model describe the performance impacts on the Android system in regards to implemented software; the performance metrics considered are execution delay and memory allocation during migration.

4.2 Design Concept

In this section, the discussion pertains to the inner workings of the second prototype's design. Moreover, this paper elaborates on the system overview (highest level of abstraction) and follows up with detailed explanations (lowest level of abstraction) for the individual blocks illustrated in Figure 8. The second prototype performs migration between two instances of Android in order to conserve energy on the instance requesting a migration and vice versa. The current platforms may be structured in two ways. The first and primary structure migrates between Android on a smartphone and Android X86 on a VM. The second structure migrates between two Android X86 VMs in order to reduce the time involved with software development and debugging. Subsequently, the system with two VMs will be used for majority of the experiments and implementation, throughout this paper. In particular, this migration process begins at the "application layer" utilizing an Android application for manual triggering or the modified "NFC service" for automatic triggering through hardware detection of a NFC tag or device (see Figure 8). Before the migration process, the client (migration initiator) retrieves the IP address and port number from a text file stored in memory in order to direct network traffic to the server (targeted Android VM or device). At the moment the migration occurs, the Android OS gathers cached application states as well as the currently active application state in the framework. Next the application states are packaged using NFC Data Exchange Format (NDEF) protocol in order to prepare them for wireless transmission to the targeted platform, as depicted in Figure 8. After the wireless transfer of the application states to the Android VM on the desktop or device, the Android OS

processes the incoming data and distributes it appropriately to update that platform's application states. For any details on the ASMT's code, refer to Android Open Source Project (AOSP) repository for original files and Appendix C for the Diff file representing the modifications made to individual files of Android's OS.

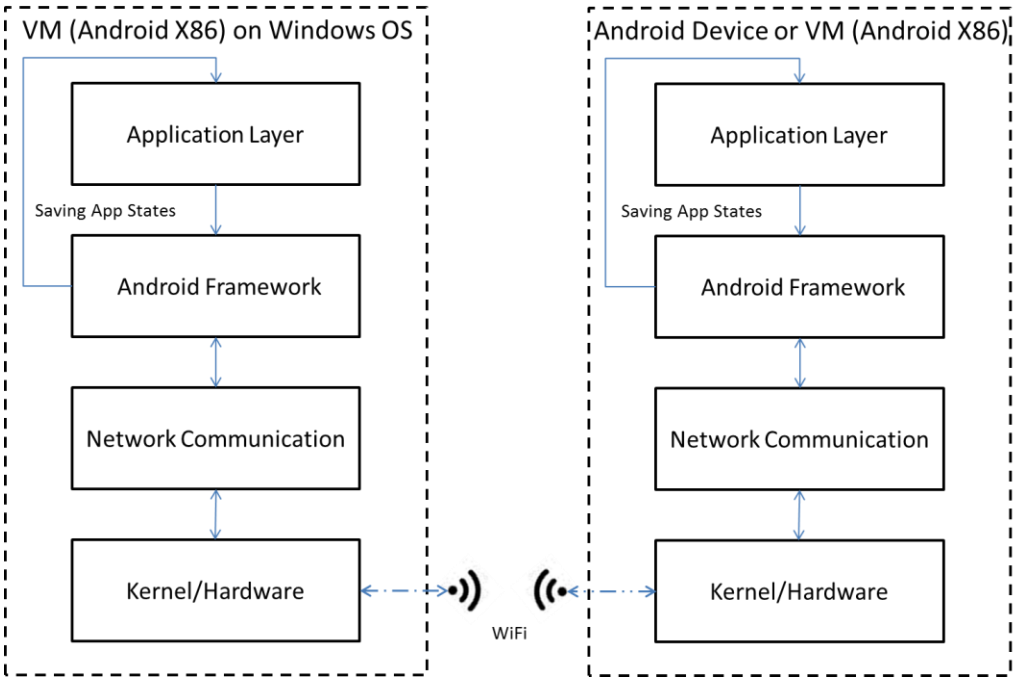

Figure 8. The system diagram of the ASMT (Proto 2)

The remaining of this section explains the specifics involved with both manual and automatic triggering of migration for the second prototype. Before explaining the in-depth design of this migration technique, this paper will provide background information on Android's activity lifecycle and the definition of an application. The activities in the system are managed in a way that behaves as a First In Last Out (FILO) stack (see Figure 9). When a new activity is started, the activity manager places that activity on the top of

the stack. For each new activity, the previous activity on the top of the stack is shifted down by one place. Thus, the previous activity will not move to the foreground until the new activity exits. An activity has the following four states as depicted in Figure 10: active or running, paused, stopped, and destroyed. In the running state, the activity is at the top of the stack or in the foreground of the screen. The paused state occurs when the activity loses focus but is still visible. A paused activity is completely alive, meaning that it maintains all state and member information. At any time, a paused activity can be killed by the system in extreme low memory situations. In regards to the stopped state, an activity is stopped when it is completely obscured by another activity. The activity still retains all of its state and member information, but the window is hidden from the user and it will often be killed by the system when memory is needed elsewhere. In either case, paused or stopped state, the system can destroy the activity by asking it to finish or simply killing its process. Hence, the activity must be completely restarted and restore to its previous state.

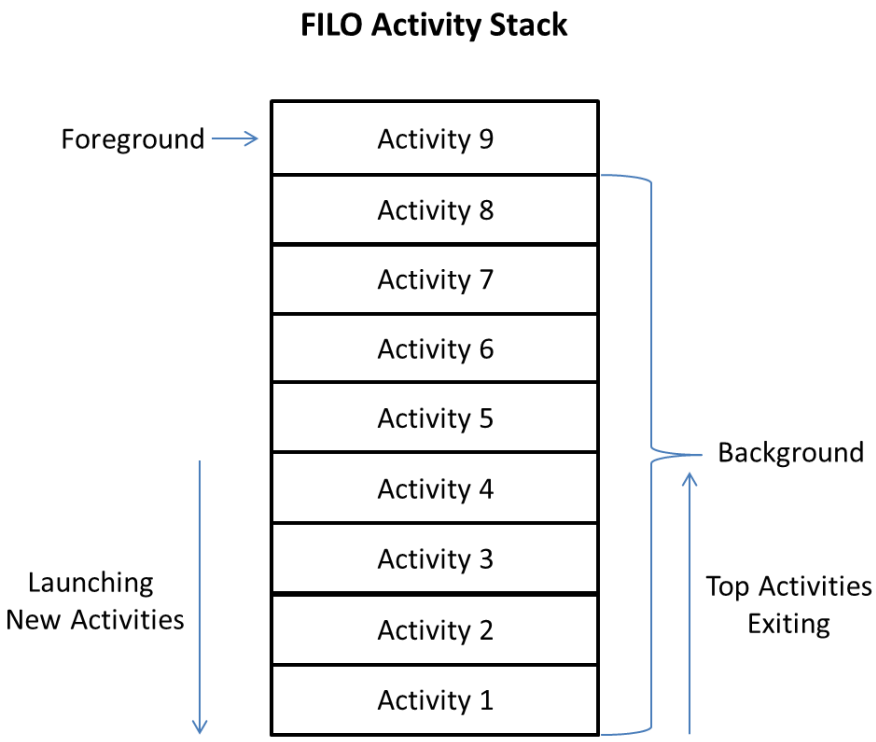

Figure 9. Android's FILO activity stack

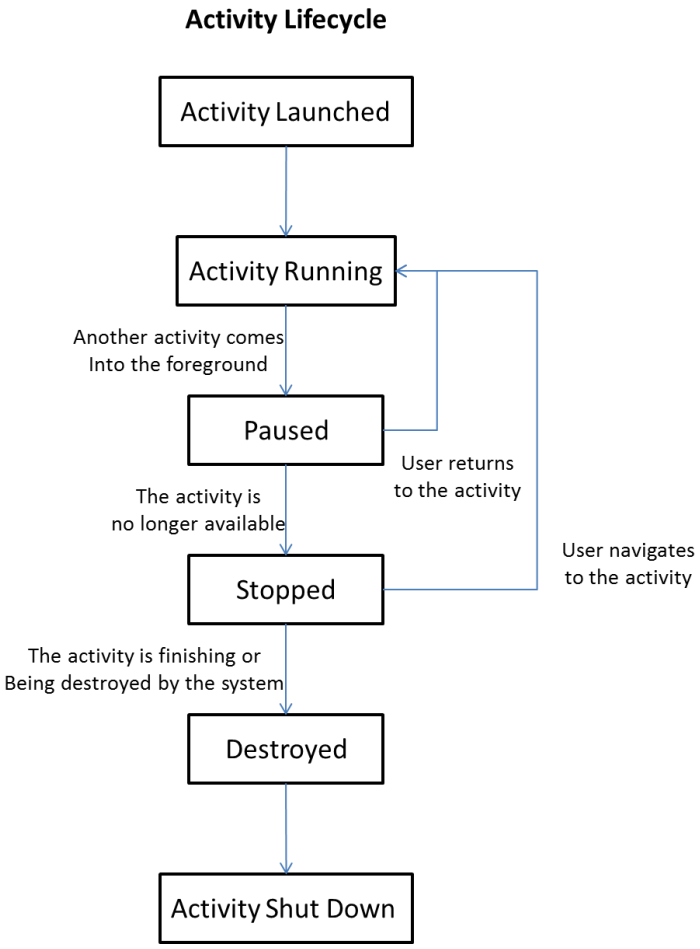

Figure 10. Android's activity lifecycle

Prior to application state migration, the system continuously acquires the callbacks to the states of cached and current applications in order to store them in memory as an array. The exact logic employed and when this process transpires will be discussed from this point on. Based on the background information given, applications may reside in various stages of the activity lifecycle governed by the user's actions. Specifically, applications are only considered cached when they are either in the paused or stopped state. Typically, applications in the stopped state are immediately destroyed by the Garbage Collector

(GC) in order to reallocate resources to other active processes. The most logical method in obtaining cached and current application states relies on the activity within an application to pass through the paused state. At the moment an activity enters the paused state, the application migration technique packages information defining that activity's callback, name, and static state and stores them into memory as seen in Figure 11.

Another concept of Android's current design that relates to this migration technique is Beaming. Android Beam allows the exchange of simple data between two Android devices. The application that wants to beam data must be in the foreground and the device receiving data must not be locked [27]. In order to enable Android Beam for an application, one of the two methods must be implemented; these methods are *setNdefPushMessage*, which accepts an NdefMessage to beam automatically when two devices are in proximity and *setNdefPushMessageCallback*, which accepts a callback that contains *createNdefMessage*. When a device is in range to beam data, *createNdefMessage* is called to create the NDEF message only when necessary [27]. As a whole, when the Beaming process takes place, Android's framework uses a callback to ask for data from the application and packs this data into an NDEF message. NDEF is a light-weight binary format used to encapsulate data. The transmission and storage standard is specified by the NFC Forum, but remains transport agnostic to the system [27]. The NDEF standard defines messages and records. An NDEF Record contains typed data or a custom application payload, whereas an NDEF Message is a container for one or more NDEF Records [27]. The migration technique modifies a range of NFC's

objects in order to perform application state migration. These objects are critical to the design of the ASMT presented in this paper. The following information describes the purpose of each modified object:

- **NFC Activity Manager** – manages NFC's APIs that are hooked to the lifecycle of an activity. The modifications here consist of sending the application's callback, static state message, and task name to the NfcService object, when the activity transitions to the paused state.
- **NFC Service** – initializes hardware and considered the heart of NFC's operations. This object's modification range from disabling hardware checks for the VM instances to the addition of a HashMap that houses applications' callbacks and package names.
- **P2P Link Manager** – manages the sending and receiving of NDEF messages over LLCP link. Modifications are made to pass the HashMap of application callbacks, be able to distinguish between manual and automatic trigger, and the creation of state messages.
- **Native NFC Manager** – is the interface between the NFC controller's firmware and NFC service. There are no modifications made to this object. It is an object critical to the automatic trigger.
- **NDEF Push Client** - configures the client's socket and handles the outgoing data resulting from Android Beam. The major modification to this object is the addition of WiFi communication.

- **NDEF Push Server** – configures the server's socket and handles the incoming data resulting from Android Beam. The major modification to this object is the addition of WiFi communication.
- **NFC Dispatcher** – handles the decoding of NDEF messages and constructs the intents to start the correct application with the data encapsulated in the message. The modifications made to this object simply extend its ability to process an array of state messages.

In terms of this migration technique, the framework layer serves the purpose of providing APIs to software developers as well as collecting application states. The collection of application states take place when Android's activity manager declares an activity to be in the paused state, and as a result, the callback *onPause* is invoked in the *NfcActivityManager*. Within the *onPause* method, the invocation of *updateNfcService* and *setForegroundNdefPush* methods pass the following information to the *NfcService* via Android's Inter-Process Communication (IPC): the static application state (a state that remains the same), the activity's callback to its object, and the task's name. The callback and task name are added to a HashMap in order to manage multiple entries. A HashMap is an interface that stores key/value pairs. HashMap also accepts null values for both the key and value. The main difference between a Map and HashMap is that the HashMap provides optimum performance level by assigning explicit capacity and load factor [28]. However, the capacity expands when needed. If the static application state is not null, then it is added to an array of states ready for migration. IPC provides the mechanism for

facilitating communications and data sharing between applications [29]. Typically, the types of applications that use IPC are categorized as clients and servers. Additionally, the passing of the activity's callback provides a way to keep the weak reference to the application's callback alive without risking null pointer references. In order to provide synchronization between the *NfcService* and the current activity's threads in the paused state, an intrinsic lock was implemented to enforce exclusive access to the object's state and establish happens-before relationships that are essential to visibility [30]. Once, the migration process begins automatically or manually, the "P2pLinkManager" request dynamic application states (states that vary based on circumstance) by executing the *createNMessage* method using the HashMap of application callbacks; which, returns a NDEF message per application callback to the "P2pLinkManager" that contains the content associated with that application's state as shown in Figure 11. The "P2pLinkManager" receives the messages and stores them in an array list containing application state messages.

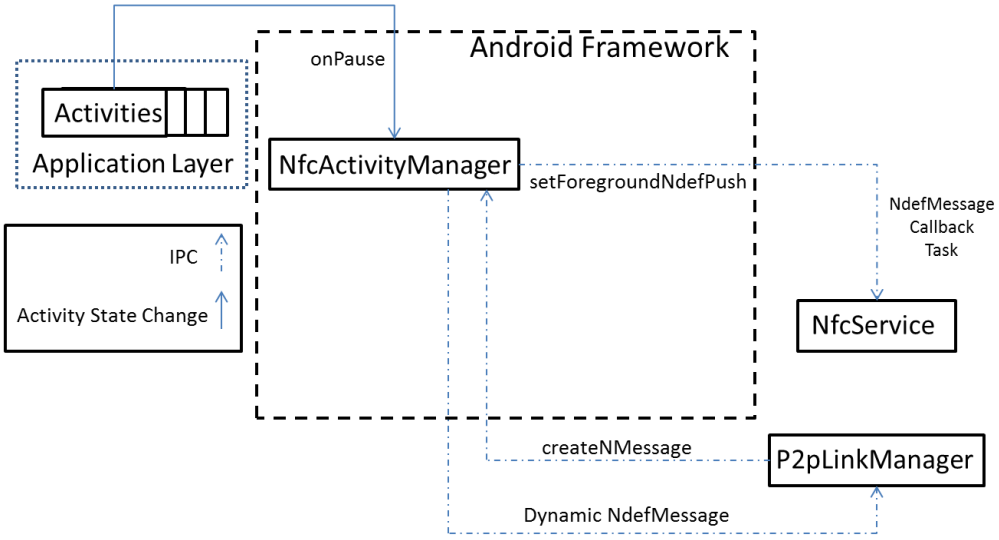

Figure 11. The framework layer for the ASMT

Aside from the logic responsible for gathering application states, the majority of the software implementation resides in the application layer. In general, the application layer manages the following aspects of the ASMT and will be discussed in the given order: storing of application states, operation of manual and automatic triggering, transferring of data over the network using a client and server, and dispatching of states to the correct application. This discussion will start with details on how the application states are stored as an array and the corresponding replacement policy. When an application losses focus, Android's framework gathers three pieces of information about an application: static state, callback, and task's name. These pieces of information are communicated through IPC to the *NfcService* by the method, *setForegroundNdefPush*. Inside the *NfcService*, the *setForegroundNdefPush* method adds application callbacks to a HashMap with the tasks' names as the index (see Figure 12). The replacement policy's logic resides in the *setForegroundNdefPush* method, which actively maintains unique entries and updates the

existing entries when necessary. In order to avoid multiple entries for the same application, the replacement policy iterates through the tasks' names in the HashMap to add a new instance or replace the instance identical to the one being added. During each invocation of *setForegroundNdefPush*, the "P2pLinkManager" receives an updated version of the HashMap as well as the current static application state. Specifically, the "P2pLinkManager" is updated by the execution of the *setNdefToSend* method. In the end, the *setNdefToSend* method produces two types of arrays: state messages and a HashMap containing application callbacks and their corresponding names.

The following information will describe the logic responsible for automatic and manual triggering of the ASMT. The initiator for each type of trigger varies. Automatic triggering begins when the migrator detects a NFC tag or smartphone. Meanwhile, manual triggering occurs when the migrator invokes *onLlcpLinkActivated* method. Thus, the two initiators involved in this migration technique are the hardware detector and method invoker (user). The automatic trigger process starts when the migrator's NFC antenna detects the NFC tag or smartphone's electromagnetic field. From this point, the native code (Java Native Interface (JNI)) written in C/C++ recognizes that a tag or device has been detected and notifies the *NativeNfcManager* by executing the *notifyLlcpLinkActivation* method. Thereafter, the invocation of the *notifyLlcpLinkActivation* leads to the invocation of the method, *onLlcpLinkActivated* in *NfcService*. These two methods simply raise a flag that a LLCP link has been established between two P2P devices or Android instances. At this current stage, the manual and

automatic trigger follows the same logic. The manual differs from the automatic trigger's logic by bypassing the need of hardware detection of a NFC tag or device. Instead, the manual trigger directly invokes the *onLlcpLinkActivated* method when the user presses the button displayed in the *NfcTrigger* activity, as seen in Figure 12. The following information will explain the remaining stages of the trigger process without distinction of the manual or automatic triggers' perspectives, since the both are identical from this point on. Inside the *NfcService*, the *onLlcpLinkActivated* method passes a message to the handler indicating that application state migration has been initiated. The message handler executes the *llcpActivated* method, which in turn invokes the *onLlcpActivated* method in the *P2pLinkManager*. The main purpose of the *P2pLinkManager* is to prepare applications' state messages for communication over the network. Therefore, the *prepareMessageToSend* method identifies what type of migration has transpired; For instance, was the migration automatically or manually triggered. The identification process simply checks the variable *manualtrigger* for a *1* if the migration was manually invoked or *0* for automatically invoked. Immediately, the method *prepareMessageToSend* creates the state messages with *createNMessage* by iterating through the HashMap of callbacks. Each state message created is stored in an array of type NdefMessage. However, if any one of the application callbacks ends up being null, then a default state is created for that specific app. A default state of an application only consists of an intent that opens that application. Finally, the array of application states is sent to *NdefPushClient* with the invocation of the "push" method. The *NdefPushClient* prepares the data to be transferred over the network to the server (see Figure 12). The

exact details on how the application states are transferred over the network using the client-server model will be discussed in the next section.

The client and server for the ASMT are encapsulated in two classes: *NdefPushClient* and *NdefPushServer*. The *NdefPushClient* configures and initializes the client similar to the first prototype's client in section 3.1.2. In essence, the configuration of client consists of obtaining the server's IP address from a text file labeled "server_ip.txt", and afterwards the initialization follows. Thereafter, the internet protocol address is defined using the information from the text file as well as the socket address by the following object instantiations: *InetAddress* and *SocketAddress* respectively. In particular, the server's port was arbitrarily selected as *3333*, with the only condition that this value resides outside the reserve port number range. Next, the client establishes a connection with the server via a socket using the *connect()* method. Prior to sending the array of state messages, the messages are converted to NDEF's push protocol and parsed into a byte array. In conclusion, the creation of an output stream provides the pathway to transmit byte converted application states to the server.

The *NdefPushServer* configures and initializes the server similar to the first prototype's server in section 3.1.2. The main difference between the first and second prototypes' server logic is that the second uses separate threads for the incoming connection request and the handling of incoming connections. The server uses the port number *3333* to initialize the server. The server creates a socket with the *ServerSocket* instantiation and

waits for incoming connection request with the “accept” method. This method blocks the thread until it establishes a connection with the client. As a consequence, this logic is executed on a separate thread in order to avoid blocking the User Interface (UI) thread. After achieving a connection, the incoming connection as well as data processing starts on a new thread. An input stream provides the pathway for receiving byte converted application states and storing them in a buffer. The buffer is reconstructed with NDEF’s push protocol and obtains the application state messages with the *getImmediate* method. The final step in the server sends the state messages to the “P2pLinkManager” by the invocation of *onMessageReceived*. The last portion of this discussion ends with the details on the way state messages are dispatched to the correct applications.

The dispatching of the state messages to the Android OS effectively updates the applications common between both Android instances, either an Android smartphone or VM. The “P2pLinkManager” executes the *onReceiveComplete* that uses the message handler to notify the system about the receive status of the state messages, as illustrated in Figure 12. If the state messages are received properly, the status message, “MSG_RECEIVE_COMPLETE” is sent to the handler for processing. In response, the message handler updates the *NfcService* with the *sendMockNdefTag* method, which passes the array of state messages to the NFC service’s handler. NFC service issues the dispatch of each state message as it iterates through the array. The dispatch process in *NfcDispatcher* operates on each state message by building a unique intent specifically for that application’s state message. Subsequently, the unique intent starts the corresponding

activity, while passing the application's state message to the appropriate activity in time for updating. In conclusion, the dispatching of state messages brings closure to the application state migration process and prepares the system for another migration.

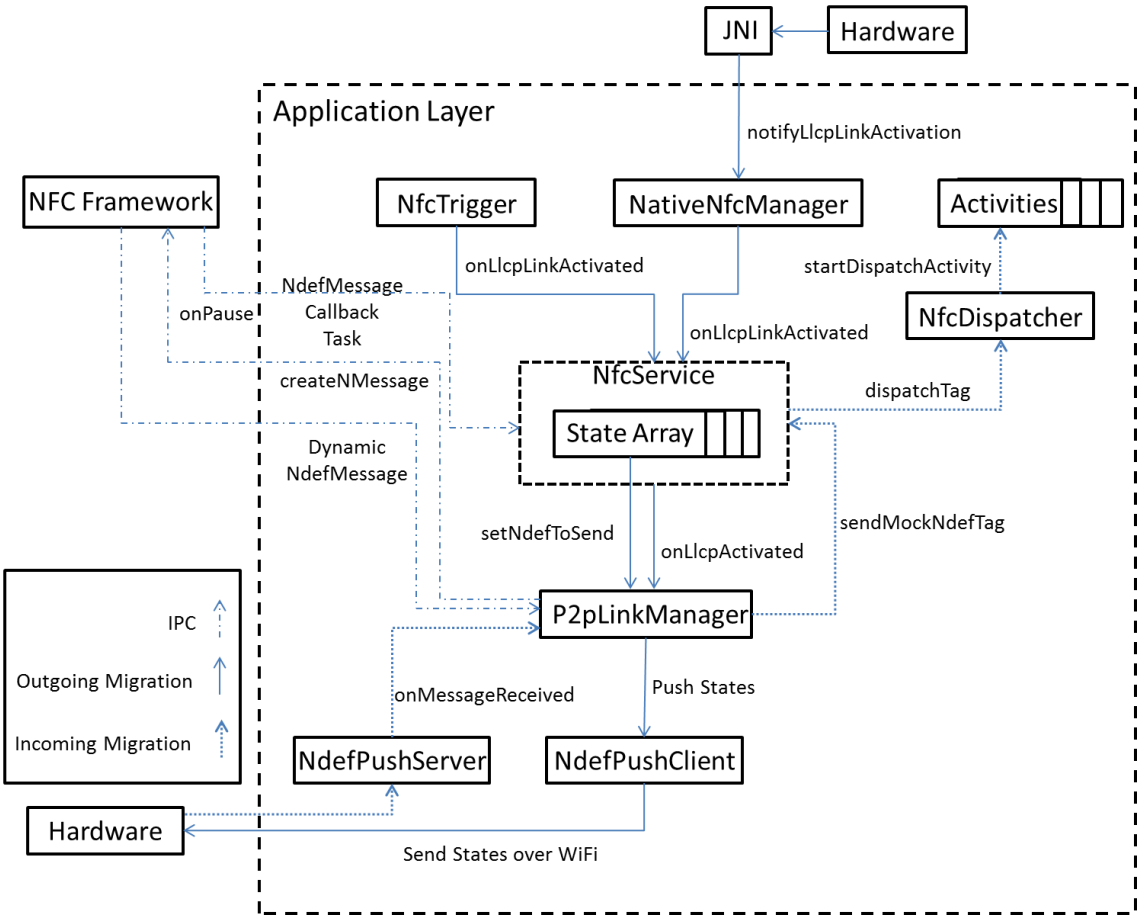

Figure 12. The application layer for the ASMT

4.3 Android Browser Example

The remaining of this section will introduce how an app can create and transmit a state message for synchronizing two Android instances regardless of platform. The ASMT extends the functionality of the current APIs in regards to NFC and NDEF. The functionalities added to the framework and NFC provide the means to create state messages as well as handling multiple messages. The two APIs used to enable state migration for an application are: *setNdefPushMessageCallback,* which accepts a callback that contains a *createNdefMessage*, and *createNdefMessage*, which allows the creation of a message with data that might vary based on the content currently visible to the user. The method *createNdefMessage* is called when a device is in range of the Migrator's antenna or when migration is manually invoked. An application can have multiple state messages, which are serialized by using an array of NdefRecords packed into an NdefMessage. An NdefRecord defines a record with a type, id, and payload field; the type field details the typing for the payload, the id field uses a meta-data identifier, and the payload field is the actual payload. Furthermore, a NDEF message is a container for one or more NDEF records [27]. Processing an NdefMessage requires the main activity of the application to filter for application state intents. In general, applications can filter for multiple types of intents and adding an additional type does not affect the current infrastructure. The last step involves the differentiation of unpackaged application state messages and appropriately updating aspects of the application.

For an example, this paper will describe the extension added to Android's native web browser to enable state migration for this particular application. Originally, Android's browser already implements the two methods necessary for Android's Beam feature (lines 56 and 81 of Figure 13). Therefore, the ASMT is implemented in such a way that enables state migration for any application employing these two methods, *setNdefPushMessageCallback* and *createNdefMessage*. Moreover, to correctly use the *setNdefPushMessageCallback*, the callback passed to this method must be an object implementing *NfcAdapter.CreateNdefMessageCallabck* (line 37 of Figure 13) and include the definition of *createNdefMessage* (lines 80-122 of Figure 13). As a result, Android's browser requires little modification to be compatible with this technique. The browser sets the NDEF callback in order to grant Android's framework the permission to invoke *createNdefMessage*. The logic residing in the overridden *createNdefMessage* method, determines the type of information packaged in the state message. Currently, the browser migrates one Uniform Resource Locator (URL) tab to the recipient. The extension increases functionality by including multiple tabs in the state message, which completely encapsulates the browser's current state (lines 92-119 of Figure 13). Resultantly, the extended Android browser captures the current application state completely. For any details on the browser modification, refer to Appendix D at the end of this paper.

```
37  public class NfcHandler implements NfcAdapter.CreateNdefMessageCallback {
38      static final int GET_PRIVATE_BROWSING_STATE_MSG = 100;
39      static final String TAG = "BrowserNfcHandler";
40      final Controller mController;
41
42      Tab mCurrentTab;
43      boolean mIsPrivate;
44      CountDownLatch mPrivateBrowsingSignal;
45
46      public static void register(Activity activity, Controller controller) {
47          NfcAdapter adapter = NfcAdapter.getDefaultAdapter(activity.getApplicationContext());
48          if (adapter == null) {
49              return;  // NFC not available on this device
50          }
51          NfcHandler handler = null;
52          if (controller != null) {
53              handler = new NfcHandler(controller);
54          }
55
56          adapter.setNdefPushMessageCallback(handler, activity);
57      }
```

```
80      @Override
81      public NdefMessage createNdefMessage(NfcEvent event) {
82          mCurrentTab = mController.getCurrentTab();
83          List<Tab> currentTabs = mController.getTabs();
84          NdefRecord[] recordarray;
85          Log.i(TAG, "Number of Tabs open =" + Integer.toString(currentTabs.size()));
86          if(currentTabs.size() > 0) {
87             recordarray = new NdefRecord[currentTabs.size()];
88          }
89          else {
90              return null;
91          }
92          for (int i = 0; i < currentTabs.size(); i++) {
93              if ((currentTabs.get(i) != null) && (currentTabs.get(i).getWebView() != null)) {
94                  // We can only read the WebView state on the UI thread, so post
95                  // a message and wait.
96                  mPrivateBrowsingSignal = new CountDownLatch(1);
97                  mHandler.sendMessage(mHandler.obtainMessage(GET_PRIVATE_BROWSING_STATE_MSG));
98                  try {
99                      Log.i(TAG, "thread waiting for 1 countdown");
100                     mPrivateBrowsingSignal.await();
101                 } catch (InterruptedException e) {
102                     return null;
103                 }
104             }
105
106             if ((currentTabs.get(i) == null) || mIsPrivate) {
107                 return null;
108             }
109             Log.i(TAG, "Passed the hang up");
110             String currentUrl = currentTabs.get(i).getUrl();
111             if (currentUrl != null) {
112                 recordarray[i] = NdefRecord.createUri(currentUrl);
113                 //NdefMessage msg = new NdefMessage(new NdefRecord[] { record });
114                 //return msg;
115             } else {
116                 Log.i(TAG, "Browser returned null");
117                 return null;
118             }
119         }
120         Log.i(TAG, "Constructing NdefMessage for all Tabs");
121         NdefMessage msg = new NdefMessage(recordarray);
122         return msg;
```

Figure 13. Android's browser code depicting the necessary addition to enable the ASMT

4.4 Results

In order to measure the performance impact of the ASMT, the implementation was investigated with three experiments. The first part analyzed the effects on execution time in regards to different data sizes that are transferred across Android's binder protocol. The second part looked at how the execution time varied based on typical data sets for the migration of multiple application states. The final experiment measured the CPU utilization during migrations with multiple application states. Each experiment was designed to measure performance impact based on the worst case scenario. For the experiments, the following assumptions are made based on observations and external research [24]. Hence, the setup details and conditions for each experiment follow:

- Experiment 1: The first experiment measures execution time in nanoseconds across IPC calls with varying messages sizes: 100B, 1KB, 16KB, 32KB, and 64KB.
- Experiment 2: The second experiment measures execution time in nanoseconds for the whole system with exactly 10 state messages and the following package sizes: 1000B, 10KB, 50KB, and 100KB.
- Experiment 3: The third experiment measures CPU utilization for the whole system with exactly 10 state messages and the following package sizes: 1000B, 10KB, 50KB, and 100KB.

The results from the first experiment do not come as a surprise. As depicted in Figure 14, the execution time increases exponentially with the size of the message. The execution time for the worst case scenario clocks in around "0.00612" seconds.

Moreover, Figure 14 exposes the limitation of the inter-process communication, since the execution time to process larger messages will hinder a multi-message system invoking the same call. Most importantly, for the common case the inter-process communication performs well with little performance degradation.

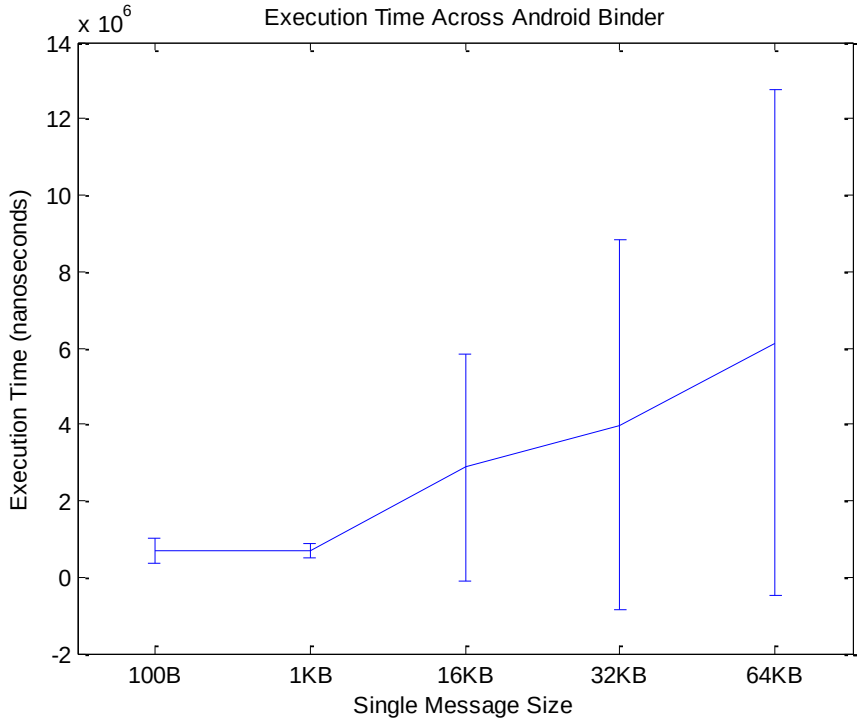

Figure 14. Execution time for IPC across different message sizes

The second experiment revealed the limitation of the system and Android's IPC. In the case when multiple application states are migrated, ASMT cannot dispatch each state message sequentially in a loop. In fact, the best solution to this problem is to insert a *thread.sleep()* in between each dispatch. The shortest pause between each iteration is 1 second. Any pauses shorter than 1 second does not allow the system to completely start

each application and update its state. Consequently, Figure 15 verifies the slow execution times of a multi-state migration. Theoretically, the fastest execution times for a 10 application state migration would be 10 seconds. However, the execution time for the worst case scenario is “12.1036” seconds with a total size of 100KB.

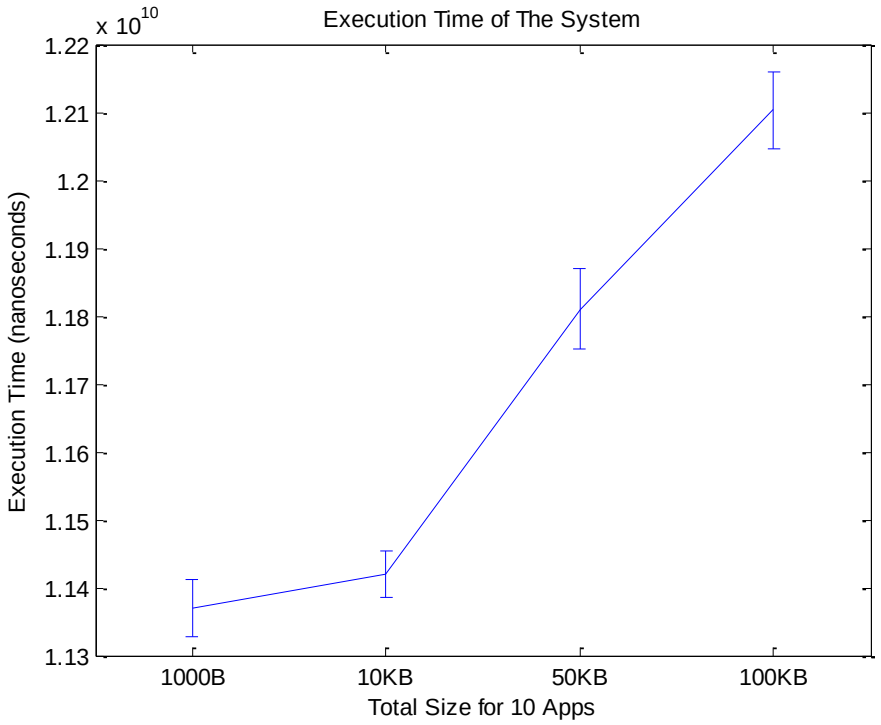

Figure 15. Execution time for ASMT with a fixed number of apps and varying total size

The results from the third experiment proved that the implementation of ASMT has insignificant overhead. In figure 16, the CPU utilization did not change much from a small to large migration package. In the worst case scenario, the CPU utilization reached a maximum of 8.2%. In conclusion, the modification made to Android’s OS had little impact on the system as a whole.

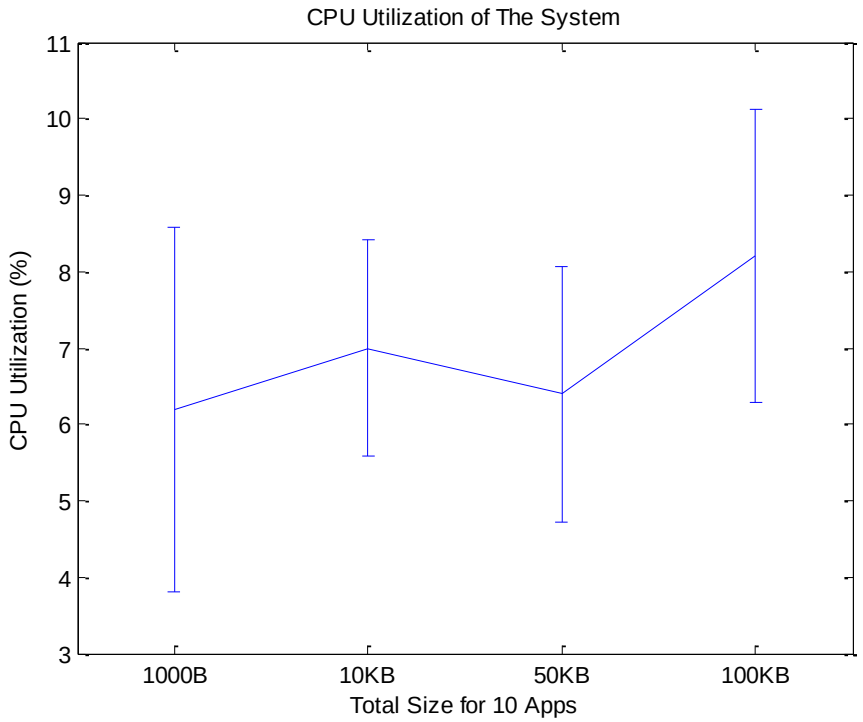

Figure 16. CPU utilization of ASMT with a fixed number of apps and varying total size

Chapter 5: Discussion

In conclusion, this project was successful at meeting its goal of providing a working prototype ASMT, but it came well short of offering a final implementation for a commercial product. This project proved to be challenging in terms of developing software that is light-weight and practical. For instance, this technique has proven to be quite effective in conserving energy with state message's sizes less than 10KB. However, for applications that produce significant amount of dynamic data (e.g. 3D games), this technique would bottleneck during inter-process transactions.

5.1 Challenges

Over the course of developing this ASMT, there were multiple noteworthy challenges that had to be resolved. Firstly, the configuration of the Android VM required additional coding to work seamlessly on a desktop. Originally, the VM does not possess a virtual sdcard, and with the time given, the best solution to this problem was to create an sdcard folder on the existing internal space provided to Android. In addition, modifications were made to the Android's "webkit" in order to allow the browser to display internet webpages. The second challenge was figuring out the modifications necessary to invoke Android Beam manually without breaking its original functionality. In the end, the best solution was to create an application (an activity, NfcTrigger) that would be face of NFC service. The last problem presented itself during testing multiple application and different

state message sizes. The issue ended up being due to the limitation of Android's IPC. Android's IPC can only transfer a maximum of 1MB of data for a given process. After trying to verify this claim, it happens to be that the most a single IPC call can transfer is 64KB. This reduction in data size varies based on the process of interest, since other IPC data transfers are occurring at the same time. However, when experimenting with different state message sizes, most applications only needed message sizes 10KB or less [24].

5.2 Future Work

A number of areas of this project can be developed further for future endeavors. The performance evaluation of this technique can be significantly improved by reducing the time necessary to obtain performance data. This can be done by building a discrete model of the system that approximates the real system's behavior. As a result, recompiling of the Android OS will not be necessary to change testing implementation or parameters. The time it takes to compile major changes to the Android OS varies significantly from 5 to 35 mins, which is unacceptable for extensive experimentation. Furthermore, an ASMT such as the one presented in this paper, should be implemented in the Android OS as a unique service with corresponding APIs. The additional framework may increase overall system overhead, but the advantage of managing IPC calls as separate transactions will allow for larger state messages. Applying the same logic to Android's NFC framework would require too many modifications and ultimately change its intended functionality.

List of References

[1] B. SIG (2014) Bluetooth Basics. [Online]. Available: http://www.bluetooth.com/

[2] Android (2014) Near Field Communication. [Online]. Available: http://developer.android.com/guide/topics/connectivity/nfc/index.html

[3] B. Mitchell (2014) What is the Typical Range of a Wi-Fi Network?. [Online]. Available: http://compnetworking.about.com/cs/wirelessproducts/f/wifirange.htm

[4] Oracle (2014) Introduction to VirtualBox. [Online]. Available: https://www.virtualbox.org/

[5] E. Bott (2013) Latest OS share data shows Windows still dominating in PCs. [Online]. Available: http://www.zdnet.com/

[6] Microsoft (2014) Windows 7 system requirements. [Online]. Available: http://windows.microsoft.com/en-us/windows7/products/system-requirements

[7] L. Cassavoy (2014) What Makes a Smartphone Smart?. [Online]. Available: http://cellphones.about.com/od/smartphonebasics/a/what_is_smart.htm

[8] Android (2014) Android, the world's most popular mobile platform. [Online]. Available: http://developer.android.com/about/index.html

[9] Android (2013) Android 4.0.4 Compatibility Definition. [Online]. Available: http://source.android.com/compatibility/android-cdd.pdf

[10]wiseGEEK (2014) What is a Computer Migration?. [Online]. Available: http://www.wisegeek.com/what-is-a-computer-migration.htm

[11] A. Saarinen, M. Siekkinen, Y. Xiao, J. K. Nurminen, and M. Kemppainen (2012) Can offloading save energy for popular apps?. MobiArch, pp 1-7

[12] K. Kumar, J. Liu, Y.-H. Lu, and B. Bhargava (2012) A survey of computation offloading for mobile systems. MobileNetwork, pp 2-8

[13] Rim H, Kim S, Kim Y, Han H (2006) Transparent method offloading for slim execution. In: International symposium on wireless pervasive computing, pp 1–6

[14] Yang K, Ou S, Chen H-H (2008) On effective offloading services for resource-constrained mobile devices running heavier mobile internet applications. IEEE communications magazine 46(1):56–63

[15] Xian C, Lu Y-H, Li Z (2007) Adaptive computation offloading for energy conservation on battery-powered systems. In: International conference on parallel and distributed systems, pp 1–8

[16] Chun BG, Maniatis P (2009) Augmented smartphone applications through clone cloud execution. In: Conference on hot topics in operating systems, USENIX Association, pp 8–12

[17] A. Carroll and G. Heiser (2010) An analysis of power consumption in a smartphone. USENIX Association, pp 1-4

[18] A. Shye, B. Scholbrock, and G. Memik (2009) Into the wild: Studying real user activity patterns to guide power optimizations for mobile architectures. In: Annual IEEE/ACM International Symposium, pp 6-10

[19] J. Reich, M. Goraczko, A. Kansal, and J. Padhye (2007) Sleepless in seattle no longer. Usenix Association, pp 11-14

[20] N. Bila, E. de Lara, K. Joshi, H. A. Lagar-Cavilla, M. Hiltunen, and M. Satyanarayanan (2012) Jettison: Efficient idle desktop consolidation with partial vm migration. In: International Conference on Mobile Systems, Applications, and Service, MobiSys, pp 8-11

[21] T. Das, P. Padala, V. N. Padmanabhan, R. Ramjee, and K. G. Shin (2010) Litegreen: Saving energy in networked desktops using virtualization. In: Annual Technical Conference, USENIX Association, pp 9-14

[22] A. Smith (2013) Smartphone Ownership 2013. [Online]. Available: http://www.pewinternet.org

[23] C. Phones (2013) New Research Reveals Mobile Users Want Phones to Have a Longer Than Average Battery Life. [Online]. Available: http://www.catphones.com/news

[24] B. Beitman and X. Wang (2014) StateMig: Dynamic App State Migration between Smartphone and Desktop for Energy Conservation, pp7-11

[25] Java (2014) What is a Socket?. [Online]. Available:
http://docs.oracle.com/javase/tutorial/networking/sockets/definition.html

[26] Android (2014) Downloading and Building. [Online]. Available:
https://source.android.com/source/building.html

[27] Android (2014) Beaming NDEF Messages to Other Devices. [Online]. Available:
http://developer.android.com/guide/topics/connectivity/nfc/nfc.html#p2p

[28] S. Nageswara (2011) Map vs HashMap. [Online]. Available:
http://way2java.com/collections/map/map-vs-hashmap/

[29] Microsoft (2011) Interprocess Communications. [Online]. Available:
http://msdn.microsoft.com/en-us/library/windows/desktop/

[30] Java (2014) Intrinsic Locks and Synchronization. [Online]. Available:
http://docs.oracle.com/javase/tutorial/essential/concurrency/locksync.html

Appendix A: First Prototype's Server Code

```
/*--------------------------------------------------------------------------
 * First Prototype's Server Code for Migration Technique
 * Author: Nathaniel J. Morris
 * Date: 07/20/2013
 *--------------------------------------------------------------------------*/

import java.io.File;
import java.io.FilenameFilter;
import java.io.IOException;
import java.io.InputStream;
import java.io.OutputStream;
import java.net.ServerSocket;
import java.net.Socket;
import java.util.regex.Matcher;
import java.util.regex.Pattern;

import javax.swing.JOptionPane;
import javax.swing.JTextField;


public class Server {
private static Pattern pattern;
private static Pattern pattern1;
private static Matcher matcher;
private static Matcher matcher1;
// This string defines the format that the ip address should be in.
private static final String IPADDRESS_PATTERN =
                    "^([01]?\\d\\d?|2[0-4]\\d|25[0-5])\\." +
                    "([01]?\\d\\d?|2[0-4]\\d|25[0-5])\\." +
                    "([01]?\\d\\d?|2[0-4]\\d|25[0-5])\\." +
                    "([01]?\\d\\d?|2[0-4]\\d|25[0-5])$";
// This string defines the format that the port number should be in.
private static final String PORT_PATTERN = "^(\\d*)$";
// The intermediate directory for data storage during the migration.
static File srcFolder = new File("C:\\data\\app");
private static String S_ID_T = null;
private static String D_ID_T = null;


      public static void main(String[] args) {
```

```
            try{

/*-------------------------------------------------------------------------
 * Specifying the text fields for the input of the server's parameters
 *-------------------------------------------------------------------------*/
            JTextField field1 = new JTextField();
            JTextField field2 = new JTextField();
            JTextField field3 = new JTextField();
            Object[] message = {
                  "Port #:", field1,
                  "Source ADB Device-ID:", field2,
                  "Destination ADB Device-ID",field3,
            };
//-------------------------------------------------------------------------


/*-------------------------------------------------------------------------
 * The variables that store the IP addresses for the source and destination
 * as well as the port number.
 *-------------------------------------------------------------------------*/
            String port=null;
            String S_ID=null;
            String D_ID=null;
//-------------------------------------------------------------------------

/*-------------------------------------------------------------------------
 * Creating the GUI window for the user to submit server parameters
 *-------------------------------------------------------------------------*/
            int option = JOptionPane.CANCEL_OPTION;
      // Assigning REGEX patters to the variables: pattern and pattern1.
            IPAddressValidator();
            while(option != JOptionPane.OK_OPTION){
                  option = JOptionPane.showConfirmDialog(null, message,
"Server Configuration", JOptionPane.OK_CANCEL_OPTION);
            // Checking the format of both the IP address and port number.
                  if (option == JOptionPane.OK_OPTION &&
validateport(field1.getText()) && validateip(field2.getText()) &&
validateip(field3.getText())){
                      port = field1.getText();
                      S_ID = field2.getText();
                      D_ID = field3.getText();

                  }
                  else{
                        JOptionPane.showMessageDialog(null, "INVALID
FORMAT:IP ADDRESS AND/OR PORT #", "Error Message", JOptionPane.ERROR_MESSAGE);
                        option = JOptionPane.CANCEL_OPTION;
                  }
                  }
//-------------------------------------------------------------------------
            boolean done = false;
            while(!done){
```

```
/*---------------------------------------------------------------------------
 * Initialization of the server's socket, input/output streams, and the
 * integer that holds the commands and status data
 *---------------------------------------------------------------------------*/
            int type = 0;
            ServerSocket ss = new ServerSocket(Integer.parseInt(port));
            Socket clientSocket = ss.accept();

            InputStream ctrl = clientSocket.getInputStream();
            OutputStream ctrlout = clientSocket.getOutputStream();
//---------------------------------------------------------------------------
      //The server notifying the client that the connection was successful
            ctrlout.write(2);


/*---------------------------------------------------------------------------
 * The main program's logic, which terminates when done is true and type = 4.
 * This logic determines the current status of the server and when to migrate
 *---------------------------------------------------------------------------*/
       while(!done && type != 4){

            type = ctrl.read();
            // When this if is true, the migration process begins
            if(type == 1){
            try{
/*---------------------------------------------------------------------------
 * Pulling all apps from the VM that invoked the migration and install them on
 * targeted VM
 * Pulling apps' database files (app states) and copying them to the targeted
 * VM
 *---------------------------------------------------------------------------*/
            Process p = Runtime.getRuntime().exec("cmd /c start/wait cmd.exe
/c \"adb start-server && adb connect "+S_ID+" && adb -s "+S_ID+":5555 root &&
adb -s "+S_ID+":5555 pull /data/app C:\\data\\app && adb -s "+S_ID+":5555 pull
/mnt/sdcard C:\\mnt\\sdcard && adb -s "+S_ID+":5555 pull
/data/data/com.benoved.phr_lite C:\\data\\data\\com.benoved.phr_lite && adb -s
"+S_ID+":5555 pull /data/data/biz.mtoy.blockpuzzle
C:\\data\\data\\biz.mtoy.blockpuzzle && adb -s "+S_ID+":5555 pull
/data/data/com.google.android.apps.translate
C:\\data\\data\\com.google.android.apps.translate\"");
//---------------------------------------------------------------------------

            p.waitFor();
            if(srcFolder.isDirectory()){
                  String[] flist = srcFolder.list(new FilenameFilter() {
                        @Override
                      public boolean accept(File srcFolder, String name) {
                          return name.endsWith(".apk");
                      }
                  });
```

```
                    String base = "cmd /c start/wait cmd.exe /c \"adb start-
server && adb connect "+D_ID+" && adb -s "+D_ID+":5555 root && adb connect
"+D_ID+" && adb -s "+D_ID+":5555 push C:\\mnt\\sdcard /mnt/sdcard";
                    String baseadd = " && adb -s "+D_ID+":5555 install -r ";
                    for (int index = 0; flist.length>index; index++) {
                          base = base+baseadd+"C:\\data\\app\\"+flist[index];
                    }

                    base = base+" && adb -s "+D_ID+":5555 push
C:\\data\\data\\com.benoved.phr_lite /data/data/com.benoved.phr_lite && adb -s
"+D_ID+":5555 push C:\\data\\data\\biz.mtoy.blockpuzzle
/data/data/biz.mtoy.blockpuzzle && adb -s "+D_ID+":5555 push
C:\\data\\data\\com.google.android.apps.translate
/data/data/com.google.android.apps.translate";
        // Copying all the content from the sdcard of one VM to the other VM
                    Process p1=Runtime.getRuntime().exec(base);
                    p1.waitFor();
              }
        // Sending status byte that the server completed the migration tasks
              ctrlout.write(3);
              S_ID_T = S_ID;
              D_ID_T = D_ID;
              S_ID = D_ID_T;
              D_ID = S_ID_T;
              }
              catch (InterruptedException e) {
                          System.out.println(e);
                          System.exit(1);
                    }
              System.out.println(type);
              }
        }

        ctrl.close();
        ctrlout.close();
        ss.close();
              }
//-------------------------------------------------------------------------
              }
              catch (IOException e) {
                    System.out.println(e);
                    System.exit(1);
              }

      }


/*-------------------------------------------------------------------------
 * Methods for validation of IP addresses and port number
 *-------------------------------------------------------------------------*/
      private static boolean validateip(final String ip){
              matcher = pattern.matcher(ip);
```

```
            return matcher.matches();
        }
    private static boolean validateport(final String port){
          matcher1 = pattern1.matcher(port);
          return matcher1.matches();
    }

    public static void IPAddressValidator(){
            pattern = Pattern.compile(IPADDRESS_PATTERN);
            pattern1 = Pattern.compile(PORT_PATTERN);
      }

//-------------------------------------------------------------------------
}
```

Appendix B: First Prototype's Client Code

```
/*---------------------------------------------------------------------------
 * First Prototype's Client Code for Migration Technique
 * Author: Nathaniel J. Morris
 * Date: 07/25/2013
 *---------------------------------------------------------------------------*/

package com.example.migration;

import java.io.IOException;
import java.io.InputStream;
import java.io.OutputStream;
import java.net.InetAddress;
import java.net.InetSocketAddress;
import java.net.NetworkInterface;
import java.net.Socket;
import java.util.Collections;
import java.util.regex.Matcher;
import java.util.regex.Pattern;

import android.app.Activity;
import android.os.AsyncTask;
import android.os.Bundle;
import android.view.View;
import android.widget.EditText;
import android.widget.TextView;
import android.widget.Toast;


public class MainActivity extends Activity {
private TextView statusText;
private EditText ipText;
private EditText portText;
private int port;
private int inprogress = 3;
private int counter = 0;
private Pattern pattern,pattern1;
private Matcher matcher,matcher1;
//This string defines the format that the ip address should be in.
private static final String IPADDRESS_PATTERN =
                    "^([01]?\\d\\d?|2[0-4]\\d|25[0-5])\\." +
                    "([01]?\\d\\d?|2[0-4]\\d|25[0-5])\\." +
```

```
                    "([01]?\\d\\d?|2[0-4]\\d|25[0-5])\\." +
                    "([01]?\\d\\d?|2[0-4]\\d|25[0-5])$";
//This string defines the format that the port number should be in.
private static final String PORT_PATTERN = "^(\\d*)$";

      @Override
      public void onCreate(Bundle savedInstanceState){
            super.onCreate(savedInstanceState);
            setContentView(R.layout.activity_main);
            TextView local = (TextView)findViewById(R.id.textView3);
            // Displaying local IP address to the user
            local.setText(getIP());

      }

      public void appclient(View view){
/*---------------------------------------------------------------------------
 * Retrieving the IP address and port number in order to connect to server.

 *---------------------------------------------------------------------------*/
            ipText = (EditText)findViewById(R.id.editText1);
          portText = (EditText)findViewById(R.id.editText2);
//---------------------------------------------------------------------------
          statusText = (TextView)findViewById(R.id.textView1);

            if(inprogress == 3 && view.getId() == R.id.button1 &&
ipText.getText().length() != 0 && portText.getText().length() != 0){


                String ipaddress = ipText.getText().toString();
                String portnum = portText.getText().toString();
                IPAddressValidator();
                if(validateip(ipaddress) == false | validateport(portnum) ==
false){
                   statusText.setText("IP Address or Port Not Formatted
Correctly");

                }
                else{
                   // The creation of the client's socket
                   port = Integer.parseInt(portnum);
                   new socketcreation().execute(ipaddress);
                }


            }
            else if(ipText.getText().length() == 0 |
portText.getText().length() == 0) {

                   statusText.setText("No IP Address or Port # Entered");
            }
            else if(inprogress == 0){
```

```
                    statusText.setText("Scheduled Task Still in Progress");

             }


       }


        private class socketcreation extends AsyncTask<String, String, String>
{
            @Override
            protected String doInBackground(String... ip) {

              try{
                     inprogress = 0;
                     if(counter == 0){

                                   publishProgress("Establishing Socket
Connection");
                                   Thread.sleep(1000);
                                   counter = counter + 1;
                              }
/*---------------------------------------------------------------------------
 * Initialization of the server's socket, input/output streams, and the
 * integer that holds the commands and status data
 *-------------------------------------------------------------------------*/
                        Socket clientSocket = new Socket();
                        clientSocket.connect(new
InetSocketAddress(ip[0],port),5000);
                        OutputStream ctrlout = clientSocket.getOutputStream();
                        InputStream ctrl = clientSocket.getInputStream();
                        int status = 0;
//---------------------------------------------------------------------------

/*---------------------------------------------------------------------------
 * Waiting for the server to notify client that connection is established
 *-------------------------------------------------------------------------*/
                             while(status != 2){
                             status=ctrl.read();
                             if(status == 2){
                                   publishProgress("Connection Estalished");
                                   Thread.sleep(1000);
                             }
                     // Sending command to trigger the migration process
                             ctrlout.write(1);
                             }
//---------------------------------------------------------------------------
                             while(inprogress != 3){
                                   inprogress = ctrl.read();
                             }
                            counter = 0;
                     // Updating status that the client has complete its tasks
```

```
                        ctrlout.write(4);
                        ctrl.close();
                        ctrlout.close();
                        clientSocket.close();
                    }
                    catch (IOException e) {
                       String error = e.toString();
                       inprogress = 3;
                       return error;
                       } catch (InterruptedException e) {
                              String error = e.toString();
                       return error;
                       }

             return "Completed";

           }
           @Override
           protected void onProgressUpdate(String... progress) {
              statusText.setText(progress[0]);
           }
           @Override
           protected void onPostExecute(String result) {
              statusText.setText(result);
           }
       }

/*---------------------------------------------------------------------------
 * Methods for retrieving the local IP address
 *---------------------------------------------------------------------------*/
      private String getIP()
      {
             try
             {
               for (NetworkInterface intf :
Collections.list(NetworkInterface.getNetworkInterfaces()))
               {
                   for (InetAddress addr :
Collections.list(intf.getInetAddresses()))
                   {
                       if (!addr.isLoopbackAddress()&&
!addr.isLinkLocalAddress() && addr.isSiteLocalAddress())
                           return addr.getHostAddress().toString();
                   }
               }
               throw new RuntimeException("No network connections found.");
             }
             catch (Exception ex)
             {
                  Toast.makeText(getApplicationContext(),"Error getting IP
address: " + ex.getLocalizedMessage(),Toast.LENGTH_SHORT).show();
                   return "Unknown";
```

```
            }
        }
//---------------------------------------------------------------------------

/*---------------------------------------------------------------------------
 * Methods for validation of IP addresses and port number
 *---------------------------------------------------------------------------*/
        private boolean validateip(final String ip){
                matcher = pattern.matcher(ip);
                return matcher.matches();
            }
        private boolean validateport(final String port){
              matcher1 = pattern1.matcher(port);
              return matcher1.matches();
        }

        public void IPAddressValidator(){
                pattern = Pattern.compile(IPADDRESS_PATTERN);
                pattern1 = Pattern.compile(PORT_PATTERN);
          }
//---------------------------------------------------------------------------

}
```

Appendix C: Second Prototype's Diff Output

```
/*--------------------------------------------------------------------------
 * Second Prototype's Diff Output of Android's Framework
 *--------------------------------------------------------------------------*/


diff -bur
/home/tman144566/Thesis/base/core/java/android/nfc/INdefPushCallback.ai
dl
/home/tman144566/WORKING_DIRECTORY/frameworks/base/core/java/android/nf
c/INdefPushCallback.aidl
---
/home/tman144566/Thesis/base/core/java/android/nfc/INdefPushCallback.ai
dl   2014-07-09 08:29:24.401390969 -0400
+++
/home/tman144566/WORKING_DIRECTORY/frameworks/base/core/java/android/nf
c/INdefPushCallback.aidl 2013-09-07 02:47:26.286201000 -0400
@@ -24,5 +24,6 @@
 interface INdefPushCallback
 {
     NdefMessage createMessage();
+    NdefMessage createNMessage();
     void onNdefPushComplete();
 }
diff -bur
/home/tman144566/Thesis/base/core/java/android/nfc/INfcAdapter.aidl
/home/tman144566/WORKING_DIRECTORY/frameworks/base/core/java/android/nf
c/INfcAdapter.aidl
--- /home/tman144566/Thesis/base/core/java/android/nfc/INfcAdapter.aidl
2014-07-09 08:29:24.401390969 -0400
+++
/home/tman144566/WORKING_DIRECTORY/frameworks/base/core/java/android/nf
c/INfcAdapter.aidl   2013-09-27 19:08:18.541840957 -0400
@@ -25,6 +25,7 @@
 import android.nfc.INdefPushCallback;
 import android.nfc.INfcAdapterExtras;
 import android.nfc.INfcTag;
+import java.util.List;

 /**
  * @hide
@@ -43,5 +44,6 @@
```

```
     void setForegroundDispatch(in PendingIntent intent,
             in IntentFilter[] filters, in TechListParcel techLists);
-    void setForegroundNdefPush(in NdefMessage msg, in
INdefPushCallback callback);
+    void setForegroundNdefPush(in NdefMessage msg, in
INdefPushCallback callback, in String task);
+    void passNdefMessage();
 }
diff -bur
/home/tman144566/Thesis/base/core/java/android/nfc/NfcActivityManager.j
ava
/home/tman144566/WORKING_DIRECTORY/frameworks/base/core/java/android/nf
c/NfcActivityManager.java
---
/home/tman144566/Thesis/base/core/java/android/nfc/NfcActivityManager.j
ava  2014-07-09 08:29:24.401390969 -0400
+++
/home/tman144566/WORKING_DIRECTORY/frameworks/base/core/java/android/nf
c/NfcActivityManager.java    2013-09-29 04:33:32.003447708 -0400
@@ -21,6 +21,22 @@
 import android.util.Log;

 import java.util.WeakHashMap;
+import java.math.BigInteger;
+
+import android.content.Intent;
+import android.content.Context;
+import android.app.Fragment;
+import android.content.ComponentName;
+import android.app.ActivityManager;
+import android.app.ActivityManager.RunningServiceInfo;
+import java.lang.Thread;
+import java.lang.InterruptedException;
+import java.util.concurrent.CountDownLatch;
+import java.util.concurrent.TimeUnit;
+import android.nfc.NdefMessage;
+import java.lang.NullPointerException;
+import android.app.ActivityManager.RunningTaskInfo;
+import java.util.List;

 /**
  * Manages NFC API's that are coupled to the life-cycle of an
Activity.
@@ -35,11 +51,14 @@
  */
 public final class NfcActivityManager extends INdefPushCallback.Stub {
     static final String TAG = NfcAdapter.TAG;
-    static final Boolean DBG = false;
+    static final Boolean DBG = true;

     final NfcAdapter mAdapter;
     final WeakHashMap<Activity, NfcActivityState> mNfcState;  //
contents protected by this
```

```
     final NfcEvent mDefaultEvent;  // can re-use one NfcEvent because
it just contains adapter
+    private CountDownLatch latch;
+    private ActivityManager manager;
+    private String task = "nReady";

     /**
      * NFC state associated with an {@link Activity}
@@ -71,6 +90,7 @@
         NfcActivityState state = mNfcState.get(activity);
         if (DBG) Log.d(TAG, "onResume() for " + activity + " " +
state);
         if (state != null) {
+            Log.i(TAG, "UpdateNfcService in Resume");
             state.resumed = true;
             updateNfcService(state);
         }
@@ -80,10 +100,57 @@
      * onPause hook from fragment attached to activity
      */
     public synchronized void onPause(Activity activity) {
+        //what actually worked sorta
+        //latch = new CountDownLatch(1);
         NfcActivityState state = mNfcState.get(activity);
+        //manager = (ActivityManager)
activity.getSystemService(activity.ACTIVITY_SERVICE);
+        //task =
manager.getRunningTasks(1).get(1).baseActivity.getPackageName();
+        task = activity.getPackageName();
+        Log.i(TAG, "First UpdateNfcService in Pause");
+        state.resumed = true;
+        updateNfcService(state);
+        //Intent intent = new Intent();
+        //intent.setClassName("com.android.nfc",
"com.android.nfc.NfcNService");
+        //Log.i(TAG, "Trying to Start my Intent Service");
+        //activity.startService(intent);
+        //Log.i(TAG, "It Started the service or crashed");
+        //Log.i(TAG, "Waiting for IntentService to Retrieve data");
+        //try {
+        //        latch.await(1, TimeUnit.SECONDS);
+        //   }
+        //    catch(InterruptedException e) {
+        //        Log.i(TAG, "InterruptedException");
+        //    }
+        /*
+        try {
+                Thread.sleep(1000);
+            }
+            catch(InterruptedException e) {
+                Log.i(TAG, "InterruptedException");
+            }
+        */
+        /*
```

```
+        //best thing to do is write messages to individual files and
read them when triggering nfc
+        Integer i = 0;
+        //maybe implement aidl to just invoke the correct methods from
NFC application/service
+        while(isMyServiceRunning(activity)) {
+            //waiting for IntentService to finish
+            i += 10;
+            try {
+                Thread.sleep(i);
+            }
+            catch(InterruptedException e) {
+                Log.i(TAG, "InterruptedException");
+            }
+        }
+        */
+        Log.i(TAG, "IntentService Finished");
+        //updateNNfcService(); invoking methods directly or indirectly
+        state = mNfcState.get(activity);
         if (DBG) Log.d(TAG, "onPause() for " + activity + " " +
state);
         if (state != null) {
-            state.resumed = false;
+            Log.i(TAG, "Second UpdateNfcService in Pause");
+            state.resumed = true; // false
             updateNfcService(state);
         }
     }
@@ -92,17 +159,25 @@
      * onDestroy hook from fragment attached to activity
      */
     public void onDestroy(Activity activity) {
-        mNfcState.remove(activity);
+        Log.i(TAG, "Activity is being destroyed");
+        //mNfcState.remove(activity);
     }

     public synchronized void setNdefPushMessage(Activity activity,
NdefMessage message) {
         NfcActivityState state = getOrCreateState(activity, message !=
null);
+        byte[] bytes = message.toByteArray();
+               BigInteger bi = new BigInteger(bytes);
+               String s1 = bi.toString(16);
+               String s = new String(bytes);
+               Log.i(TAG, "NDEF RECORD :" + s);
+               Log.i(TAG, "NDEF RECORD :" + s1);
+        Log.i(TAG, "setNdefMessage");
         if (state == null || state.ndefMessage == message) {
             return;  // nothing more to do;
         }
         state.ndefMessage = message;
         if (message == null) {
-            maybeRemoveState(activity, state);
```

```
+             //maybeRemoveState(activity, state);
          }
          if (state.resumed) {
              updateNfcService(state);
@@ -112,15 +187,29 @@
      public synchronized void setNdefPushMessageCallback(Activity
activity,
              NfcAdapter.CreateNdefMessageCallback callback) {
          NfcActivityState state = getOrCreateState(activity, callback
!= null);
+         Log.i(TAG, "Callback is always triggered when app is open!");
          if (state == null || state.ndefMessageCallback == callback) {
+         Log.i(TAG, "set ndef callback not updating");
              return;  // nothing more to do;
          }
+         //added
+         if(state.ndefMessageCallback != null) {
+             callback = state.ndefMessageCallback;
+         }
+         //added
          state.ndefMessageCallback = callback;
          if (callback == null) {
-             maybeRemoveState(activity, state);
+             //maybeRemoveState(activity, state);
          }
          if (state.resumed) {
+             Log.i(TAG, "set ndef callback updating"); //this works
when triggering service
              updateNfcService(state);
+             /*Intent intent = new Intent();
+             intent.setClassName("com.android.nfc",
"com.android.nfc.NfcNService");
+             Log.i(TAG, "Trying to Start my Intent Service");
+             activity.startService(intent);
+             Log.i(TAG, "It Started the service or crashed");
+             */
          }
      }

@@ -132,7 +221,7 @@
          }
          state.onNdefPushCompleteCallback = callback;
          if (callback == null) {
-             maybeRemoveState(activity, state);
+             //maybeRemoveState(activity, state);
          }
          if (state.resumed) {
              updateNfcService(state);
@@ -162,6 +251,7 @@
      synchronized void maybeRemoveState(Activity activity,
NfcActivityState state) {
          if (state.ndefMessage == null && state.ndefMessageCallback ==
null &&
                  state.onNdefPushCompleteCallback == null) {
```

```
+            Log.i(TAG, "maybeRemoveState method removing callback from
MAP");
             NfcFragment.remove(activity);
             mNfcState.remove(activity);
         }
@@ -173,14 +263,77 @@
     synchronized void updateNfcService(NfcActivityState state) {
         boolean serviceCallbackNeeded = state.ndefMessageCallback !=
null ||
                 state.onNdefPushCompleteCallback != null;
+                Log.i(TAG, "updateNfcService");
+                //state.resumed = true; //nate-remove
+                if(task == null) {
+                    task = "nReady";
+                }
+                Log.i(TAG, "ndefMessage: " + state.ndefMessage + "
this instance: " + this + " task: " + task);

         try {
             NfcAdapter.sService.setForegroundNdefPush(state.resumed ?
state.ndefMessage : null,
-                    state.resumed && serviceCallbackNeeded ? this :
null);
+                    state.resumed && serviceCallbackNeeded ? this :
null, task);
+
+    /*NdefMessage test = null;
+                if(state.ndefMessageCallback != null) {
+                //test =
state.ndefMessageCallback.createNdefMessage(mDefaultEvent);
+                //test = createMessage();
+
+                Log.i(TAG, "ndef message callback is not null");
+                }
+                else {
+                Log.i(TAG, "ndef message callback is null");
+                }
+                if(test != null) {
+         byte[] bytes = test.toByteArray();
+               BigInteger bi = new BigInteger(bytes);
+               String s1 = bi.toString(16);
+               String s = new String(bytes);
+               Log.i(TAG, "NDEF RECORD :" + s);
+               Log.i(TAG, "NDEF RECORD :" + s1);
+               }
+               else {
+               Log.i(TAG, "Ndef Message is null");
+               }
+               if(state.resumed) {
+               Log.i(TAG, "State is Resumed");
+               }
+               */
+        } catch (RemoteException e) {
+            mAdapter.attemptDeadServiceRecovery(e);
```

```
+        }
+    }
+
+    /*
+    void updateNNfcService() {
+        Log.i(TAG, "updateNNfcService");
+        try{
+            NfcAdapter.sService.passNdefMessage();
+        }
+        catch (RemoteException e) {
+            Log.i(TAG, "RemoteException");
+            mAdapter.attemptDeadServiceRecovery(e);
+        }
+    }
+    */
+
+    /*
+    void updateNNfcService(NfcActivityState state) {
+        boolean serviceCallbackNeeded = state.ndefMessageCallback !=
null ||
+                state.onNdefPushCompleteCallback != null;
+                Log.i(TAG, "updateNNfcService");
+
+        try {
+            NdefMessage test = null;
+            test = this.createMessage();
+            NfcAdapter.sService.passNdefMessage(state.resumed ?
state.ndefMessage : null,
+                    state.resumed && serviceCallbackNeeded ? test :
null);
         } catch (RemoteException e) {
+            Log.i(TAG, "RemoteException");
             mAdapter.attemptDeadServiceRecovery(e);
         }
     }
+    */

     /**
      * Callback from NFC service
@@ -188,20 +341,80 @@
     @Override
     public NdefMessage createMessage() {
         NfcAdapter.CreateNdefMessageCallback callback = null;
+        Log.i(TAG, "Callback from NFC Service");
         synchronized (NfcActivityManager.this) {
             for (NfcActivityState state : mNfcState.values()) {
                 if (state.resumed) {
                     callback = state.ndefMessageCallback;
+                    if(state.ndefMessageCallback == null) {
+                    Log.i(TAG, "ndefMessageCallback = null");
+                    }
+                    else {
+                    Log.i(TAG, "ndefMessageCallback != null");
                 }
```

```
+                    Log.i(TAG, "Iteration resumed=true");
+                }
+                else {
+                Log.i(TAG, "Iteration resumed=false");
             }
         }

+            Log.i(TAG, "Threads are synchronized but for loop is 0");
+        }
+
         // drop lock before making callback
+        try {
+            if (callback != null) {
+                NdefMessage msg =
callback.createNdefMessage(mDefaultEvent);
+                return msg;
+            }
+        }
+        catch(NullPointerException e) {
+            Log.i(TAG, "NullPointerException Due To The
createNdefMessage");
+            return null;
+        }
+        Log.i(TAG, "createmessage returning null");
+        return null;
+    }
+
+
+    @Override
+    public NdefMessage createNMessage() {
+        NfcAdapter.CreateNdefMessageCallback callback = null;
+        Log.i(TAG, "Callback from NFC Service");
+
+            for (NfcActivityState state : mNfcState.values()) {
+                if (state.resumed) {
+                    callback = state.ndefMessageCallback;
+                    if(state.ndefMessageCallback == null) {
+                    Log.i(TAG, "ndefMessageCallback = null");
+                    }
+                    else {
+                    Log.i(TAG, "ndefMessageCallback != null");
+                    }
+                    Log.i(TAG, "Iteration resumed=true");
+                }
+                else {
+                Log.i(TAG, "Iteration resumed=false");
+                }
+            }
+        try {
         if (callback != null) {
-            return callback.createNdefMessage(mDefaultEvent);
+                //latch.countDown();
+                NdefMessage msg =
callback.createNdefMessage(mDefaultEvent);
```

```
+                 return msg;
+             }
          }
+         catch(NullPointerException e) {
+             Log.i(TAG, "NullPointerException Due To The
createNdefMessage");
+             //latch.countDown();
          return null;
      }
+         Log.i(TAG, "createmessage returning null");
+         //latch.countDown();
+         return null;
+     }
+

      /**
       * Callback from NFC service
@@ -222,5 +435,17 @@
              callback.onNdefPushComplete(mDefaultEvent);
          }
      }
+     /*
+     private boolean isMyServiceRunning(Activity activity) {
+         ActivityManager manager = (ActivityManager)
activity.getSystemService(activity.ACTIVITY_SERVICE);
+         for(RunningServiceInfo service :
manager.getRunningServices(Integer.MAX_VALUE)) {
+             Log.i(TAG, "classname=" + service.service.getClassName());
+
if("com.android.nfc.NfcNService".equals(service.service.getClassName())
) {
+                 return true;
+             }
+         }
+         return false;

+     }
+     */
 }
diff -bur
/home/tman144566/Thesis/base/core/java/android/nfc/NfcAdapter.java
/home/tman144566/WORKING_DIRECTORY/frameworks/base/core/java/android/nf
c/NfcAdapter.java
--- /home/tman144566/Thesis/base/core/java/android/nfc/NfcAdapter.java
2014-07-09 08:29:24.401390969 -0400
+++
/home/tman144566/WORKING_DIRECTORY/frameworks/base/core/java/android/nf
c/NfcAdapter.java    2013-08-20 05:05:15.168702000 -0400
@@ -301,7 +301,7 @@
      public static synchronized NfcAdapter getNfcAdapter(Context
context) {
          if (!sIsInitialized) {
              /* is this device meant to have NFC */
-             if (!hasNfcFeature()) {
```

```
+            if (false) {//nate-!hasNfcFeature()
                 Log.v(TAG, "this device does not have NFC support");
                 throw new UnsupportedOperationException();
             }
@@ -565,6 +565,7 @@
      */
     public void setNdefPushMessage(NdefMessage message, Activity
activity,
             Activity ... activities) {
+            Log.i(TAG, "NfcAdapter-setNdef");
         if (activity == null) {
             throw new NullPointerException("activity cannot be null");
         }
@@ -604,6 +605,7 @@
      */
     public void setNdefPushMessageCallback(CreateNdefMessageCallback
callback, Activity activity,
             Activity ... activities) {
+            Log.i(TAG, "NfcAdapter-setNdefCallback");
         if (activity == null) {
             throw new NullPointerException("activity cannot be null");
         }
```

```
/*-------------------------------------------------------------------------
 * Second Prototype's Diff Output of Android's System App NFC
 *-------------------------------------------------------------------------*/
```

```
diff -bur /home/tman144566/Thesis/Nfc/AndroidManifest.xml
/home/tman144566/WORKING_DIRECTORY/packages/apps/Nfc/AndroidManifest.xm
l
--- /home/tman144566/Thesis/Nfc/AndroidManifest.xml 2014-07-08
23:44:55.910104453 -0400
+++
/home/tman144566/WORKING_DIRECTORY/packages/apps/Nfc/AndroidManifest.xm
l    2013-09-05 23:24:49.947521740 -0400
@@ -4,6 +4,7 @@
         android:sharedUserId="android.uid.nfc"
         android:sharedUserLabel="@string/nfcUserLabel"
 >
+    <uses-permission android:name="android.permission.INTERNET" />
     <uses-permission android:name="android.permission.BLUETOOTH" />
     <uses-permission android:name="android.permission.BLUETOOTH_ADMIN"
/>
     <uses-permission android:name="android.permission.NFC" />
@@ -19,6 +20,11 @@
     <uses-permission
android:name="android.permission.READ_FRAME_BUFFER" />
     <uses-permission
android:name="android.permission.SYSTEM_ALERT_WINDOW" />
     <uses-permission android:name="android.permission.VIBRATE" />
```

```
+    <uses-permission
android:name="android.permission.CHANGE_NETWORK_STATE"/>
+    <uses-permission
android:name="android.permission.ACCESS_WIFI_STATE"/>
+    <uses-permission
android:name="android.permission.UPDATE_DEVICE_STATS"/>
+    <uses-permission
android:name="android.permission.CHANGE_WIFI_STATE"/>
+    <uses-permission android:name="android.permission.WAKE_LOCK"/>

     <application android:name=".NfcService"
                  android:icon="@drawable/icon"
@@ -40,5 +46,22 @@
             android:excludeFromRecents="true"
             android:noHistory="true"
         />
+
+        <activity android:name=".NfcTrigger"
+            android:icon="@drawable/trigger"
+            android:label="NfcTrigger"
+            android:excludeFromRecents="true"
+            android:noHistory="true" >
+            <intent-filter>
+                <action android:name="android.intent.action.MAIN"/>
+                <category
android:name="android.intent.category.LAUNCHER"/>
+            </intent-filter>
+        </activity>
+
+        <service android:name=".NfcNService"
+                 android:exported="true"
+                 android:enabled="true">
+        </service>
+
     </application>
 </manifest>
Only in /home/tman144566/WORKING_DIRECTORY/packages/apps/Nfc/:
AndroidManifest.xml~
Only in /home/tman144566/WORKING_DIRECTORY/packages/apps/Nfc/: bin
Only in /home/tman144566/WORKING_DIRECTORY/packages/apps/Nfc/:
.classpath
Only in /home/tman144566/Thesis/Nfc/: .git
Only in /home/tman144566/WORKING_DIRECTORY/packages/apps/Nfc/jni:
.com_android_nfc_NativeNfcManager.cpp.swp
Only in /home/tman144566/WORKING_DIRECTORY/packages/apps/Nfc/: .project
Only in
/home/tman144566/WORKING_DIRECTORY/packages/apps/Nfc/res/drawable-hdpi:
trigger.png
Only in /home/tman144566/WORKING_DIRECTORY/packages/apps/Nfc/res:
drawable-ldpi
Only in
/home/tman144566/WORKING_DIRECTORY/packages/apps/Nfc/res/drawable-mdpi:
trigger.png
```

```
Only in /home/tman144566/WORKING_DIRECTORY/packages/apps/Nfc/res:
drawable-xhdpi
Only in
/home/tman144566/WORKING_DIRECTORY/packages/apps/Nfc/res/layout:
trigger.xml
diff -bur /home/tman144566/Thesis/Nfc/res/values/strings.xml
/home/tman144566/WORKING_DIRECTORY/packages/apps/Nfc/res/values/strings
.xml
--- /home/tman144566/Thesis/Nfc/res/values/strings.xml  2014-07-08
23:44:55.910104453 -0400
+++
/home/tman144566/WORKING_DIRECTORY/packages/apps/Nfc/res/values/strings
.xml 2013-08-07 04:25:52.130672670 -0400
@@ -2,6 +2,9 @@
 <resources xmlns:xliff="urn:oasis:names:tc:xliff:document:1.2">
     <string name="app_name">Nfc Service</string>
     <string name="nfcUserLabel">Nfc</string>
+    <string name="test"></string>
+    <string name="NFC_ON">Simulate P2P/ON</string>
+    <string name="NFC_OFF">Simulate P2P/OFF</string>

     <!-- A notification description string informing the user that
contact details were received over NFC [CHAR-LIMIT=64] -->
     <string name="inbound_me_profile_title">Contact received over
NFC</string>
diff -bur
/home/tman144566/Thesis/Nfc/src/com/android/nfc/ndefpush/NdefPushClient
.java
/home/tman144566/WORKING_DIRECTORY/packages/apps/Nfc/src/com/android/nf
c/ndefpush/NdefPushClient.java
---
/home/tman144566/Thesis/Nfc/src/com/android/nfc/ndefpush/NdefPushClient
.java    2014-07-08 23:44:55.910104453 -0400
+++
/home/tman144566/WORKING_DIRECTORY/packages/apps/Nfc/src/com/android/nf
c/ndefpush/NdefPushClient.java   2013-09-05 19:56:15.984381137 -0400
@@ -26,6 +26,20 @@
 import java.io.IOException;
 import java.util.Arrays;

+// Added imports
+import java.io.BufferedReader;
+import java.io.FileNotFoundException;
+import java.io.FileReader;
+
+import java.nio.charset.Charset;
+import android.nfc.NdefRecord;
+
+import java.net.Socket;
+import java.io.OutputStream;
+import java.net.InetAddress;
+import java.net.SocketAddress;
+import java.net.InetSocketAddress;
+import java.util.List;
```

```
 /**
  * Simple client to push the local NDEF message to a server on the
remote side of an
  * LLCP connection, using the Android Ndef Push Protocol.
@@ -35,49 +49,176 @@
     private static final int MIU = 128;
     private static final boolean DBG = true;

-    public boolean push(NdefMessage msg) {
+  // Added fields
+  //private String serverHostname = null;
+  private int serverPort = 3333;
+  //private InputStream sockInput = null;
+  private OutputStream sockOutput = null;
+
+  public boolean push(NdefMessage[] msg) {
         NfcService service = NfcService.getInstance();

+    /* read Server IP address from file /sdcard/server_ip.txt */
+    String fileName = "/sdcard/server_ip.txt";
+    //String MfileName = "/sdcard/message_size.txt";
+    byte[] serverIp = null;
+    FileReader file = null;
+    FileReader Mfile = null;
+    String message = null;
+
+    try {
+      file = new FileReader(fileName);
+      BufferedReader reader = new BufferedReader(file);
+
+      if (DBG) Log.d(TAG, "Reading file server_ip.txt");
+
+      //Mfile = new FileReader(MfileName);
+      //BufferedReader Mreader = new BufferedReader(Mfile);
+
+      if (DBG) Log.d(TAG, "Reading file message_size.txt");
+
+      String ipAddress = reader.readLine();
+      //message = Mreader.readLine();
+
+      if (DBG) Log.d(TAG, "Read from file, splitting text.");
+
+      String[] ipArray = ipAddress.split("\\.");
+      serverIp = new byte[4];
+
+      if (DBG) Log.d(TAG, "Split string: " + ipAddress + " into array
size: " + ipArray.length);
+
+      for(int i=0; i<4; i++) {
+
+        if (DBG) Log.d(TAG, "Parse String i: " + i);
+        serverIp[i] = (byte)Integer.parseInt(ipArray[i]);
+        if (DBG) Log.d(TAG, "serverIp[i]: " + serverIp[i]);
+
```

```
+      }
+
+      if (DBG) Log.d(TAG, "Parse String array into byte array.");
+
+    } catch (FileNotFoundException e) {
+      //throw new RuntimeException("File not found");
+      if (DBG) Log.d(TAG, "Serve IP or message file not found");
+      serverIp = new byte[]{127, 0, 0, 1};
+    } catch (IOException e) {
+      throw new RuntimeException("IO Error occured");
+    } finally {
+      if (file != null) {
+        try {
+          file.close();
+         //Mfile.close();
+        } catch (IOException e) {
+          e.printStackTrace();
+        }
+      }
+    }
+
         // We only handle a single immediate action for now
-        NdefPushProtocol proto = new NdefPushProtocol(msg,
NdefPushProtocol.ACTION_IMMEDIATE);
+    //^NdefPushProtocol proto = new NdefPushProtocol(msg,
NdefPushProtocol.ACTION_IMMEDIATE);
+
+
+    //String mimeType = "application/com.tapped.nfc";
+    //String message = Text.hundredB;
+
+    /*if (messageF.equals("1")) {
+   message = Text.hundredB;
+    }
+    else if (messageF.equals("2")) {
+   message = Text.fivehundredB;
+    }
+    else if (messageF.equals("3")) {
+   message = Text.oneK;
+    }
+    else if (messageF.equals("4")) {
+   message = Text.fiftyK;
+    }
+    else if (messageF.equals("5")) {
+   message = Text.hundredK;
+    }
+    else if (messageF.equals("6")) {
+   message = Text.fivehundredK;
+    }
+    else if (messageF.equals("7")) {
+   message = Text.oneM;
+    }
+    else if (messageF.equals("8")) {
+   message = Text.twoM;
```

```
+    }
+    else if (messageF.equals("9")) {
+   message = Text.hundredfiftytwoK;
+    }
+    else {
+   if (DBG) Log.d(TAG, "Invalid input from message file.");
+    }
+    */
+
+    //byte[] mimeBytes = mimeType.getBytes(Charset.forName("US-
ASCII"));
+    //NdefRecord mimeRecord = new
NdefRecord(NdefRecord.TNF_MIME_MEDIA, mimeBytes, new byte[0],
message.getBytes());
+    //msg = new NdefMessage(new NdefRecord[] { mimeRecord });
+
+    //NdefMessage[] msgs = new NdefMessage[msg.size()];
+    byte[] actions = new byte[msg.length];
+    for (int i = 0; i < msg.length; i++) {
+        //msgs[i] = msg.get(i);
+        actions[i] = NdefPushProtocol.ACTION_IMMEDIATE;
+    }
+    //NdefPushProtocol proto = new NdefPushProtocol(msg,
NdefPushProtocol.ACTION_IMMEDIATE);
+    NdefPushProtocol proto = new NdefPushProtocol(actions, msg);
         byte[] buffer = proto.toByteArray();
         int offset = 0;
         int remoteMiu;
-        LlcpSocket sock = null;
+
+    //LlcpSocket sock = null;
+    Socket sock = null;
+
         try {
             if (DBG) Log.d(TAG, "about to create socket");
+
             // Connect to the my tag server on the remote side
-            sock = service.createLlcpSocket(0, MIU, 1, 1024);
+      //sock = service.createLlcpSocket(0, MIU, 1, 1024);
+      //sock = new Socket (serverHostname, serverPort);
+      sock = new Socket();
+
             if (sock == null) {
                 throw new IOException("Could not connect to socket.");
             }
             if (DBG) Log.d(TAG, "about to connect to service " +
NdefPushServer.SERVICE_NAME);
-            sock.connectToService(NdefPushServer.SERVICE_NAME);

-            remoteMiu = sock.getRemoteMiu();
+      //sock.connectToService(NdefPushServer.SERVICE_NAME);
+      InetAddress inetAddress = InetAddress.getByAddress(serverIp);
+      SocketAddress socketAddress = new InetSocketAddress(inetAddress,
serverPort);
```

```
+       sock.connect(socketAddress, 30000);
+       //sockInput = sock.getInputStream();
+       //sockOutput = sock.getOutputStream();
+       //remoteMiu = sock.getRemoteMiu();
+       remoteMiu = sock.getSendBufferSize();
+
                if (DBG) Log.d(TAG, "about to send a " + buffer.length + "
byte message");
                while (offset < buffer.length) {
                    int length = Math.min(buffer.length - offset,
remoteMiu);
+         //int length = buffer.length - offset;
                    byte[] tmpBuffer = Arrays.copyOfRange(buffer, offset,
offset+length);
                    if (DBG) Log.d(TAG, "about to send a " + length + "
byte packet");
-                   sock.send(tmpBuffer);
+         sockOutput = sock.getOutputStream();
+         sockOutput.write(tmpBuffer, 0, length);
                    offset += length;
                }
                return true;
            } catch (IOException e) {
                Log.e(TAG, "couldn't send tag");
                if (DBG) Log.d(TAG, "exception:", e);
-           } catch (LlcpException e) {
-               // Most likely the other side doesn't support the my tag
protocol
-               Log.e(TAG, "couldn't send tag");
-               if (DBG) Log.d(TAG, "exception:", e);
+       //} catch (LlcpException e) {
+       //  // Most likely the other side doesn't support the my tag
protocol
+       //  Log.e(TAG, "couldn't send tag");
+       //  if (DBG) Log.d(TAG, "exception:", e);
            } finally {
                if (sock != null) {
                    try {
-                       if (DBG) Log.d(TAG, "about to close");
+         if (DBG) Log.d(TAG, "about to close sock");
                        sock.close();
                    } catch (IOException e) {
-                       // Ignore
+    if (DBG) Log.d(TAG, "failed to close sock");
                    }
                }
            }
Only in
/home/tman144566/WORKING_DIRECTORY/packages/apps/Nfc/src/com/android/nf
c/ndefpush: NdefPushClient.java~
diff -bur
/home/tman144566/Thesis/Nfc/src/com/android/nfc/ndefpush/NdefPushProtoc
ol.java
```

```
/home/tman144566/WORKING_DIRECTORY/packages/apps/Nfc/src/com/android/nfc/ndefpush/NdefPushProtocol.java
---
/home/tman144566/Thesis/Nfc/src/com/android/nfc/ndefpush/NdefPushProtocol.java  2014-07-08 23:44:03.910848945 -0400
+++
/home/tman144566/WORKING_DIRECTORY/packages/apps/Nfc/src/com/android/nfc/ndefpush/NdefPushProtocol.java 2013-08-26 16:48:35.738429065 -0400
@@ -50,6 +50,7 @@

     public NdefPushProtocol(byte[] actions, NdefMessage[] messages) {
         if (actions.length != messages.length || actions.length == 0)
{
+            Log.i(TAG, "What is causing the issue: " + "actions=" +
actions.length + "messages=" + messages.length);
             throw new IllegalArgumentException(
                     "actions and messages must be the same size and
non-empty");
         }
@@ -134,7 +135,7 @@
             }
         }
     }
-
+    /*
     public NdefMessage getImmediate() {
         // Find and return the first immediate message
         for(int i = 0; i < mNumMessages; i++) {
@@ -144,6 +145,16 @@
         }
         return null;
     }
+    */
+    public NdefMessage[] getImmediate() {
+        // Find and return the first immediate message
+        for(int i = 0; i < mNumMessages; i++) {
+            if(mActions[i] == ACTION_IMMEDIATE) {
+                return mMessages;
+            }
+        }
+        return null;
+    }

     public byte[] toByteArray() {
         ByteArrayOutputStream buffer = new
ByteArrayOutputStream(1024);
diff -bur
/home/tman144566/Thesis/Nfc/src/com/android/nfc/ndefpush/NdefPushServer.java
/home/tman144566/WORKING_DIRECTORY/packages/apps/Nfc/src/com/android/nfc/ndefpush/NdefPushServer.java
---
/home/tman144566/Thesis/Nfc/src/com/android/nfc/ndefpush/NdefPushServer.java    2014-07-08 23:44:55.910104453 -0400
```

```
+++
/home/tman144566/WORKING_DIRECTORY/packages/apps/Nfc/src/com/android/nf
c/ndefpush/NdefPushServer.java   2013-08-24 03:58:52.554254156 -0400
@@ -29,6 +29,11 @@
 import java.io.ByteArrayOutputStream;
 import java.io.IOException;

+// Added imports
+import java.net.Socket;
+import java.net.ServerSocket;
+import java.io.InputStream;
+
 /**
  * A simple server that accepts NDEF messages pushed to it over an
LLCP connection. Those messages
  * are typically set on the client side by using {@link
NfcAdapter#enableForegroundNdefPush}.
@@ -36,7 +41,6 @@
 public class NdefPushServer {
     private static final String TAG = "NdefPushServer";
     private static final boolean DBG = true;
-
     private static final int MIU = 248;

     int mSap;
@@ -51,7 +55,7 @@
     ServerThread mServerThread = null;

     public interface Callback {
-        void onMessageReceived(NdefMessage msg);
+    void onMessageReceived(NdefMessage[] msg);
     }

     public NdefPushServer(final int sap, Callback callback) {
@@ -61,9 +65,11 @@

     /** Connection class, used to handle incoming connections */
     private class ConnectionThread extends Thread {
-        private LlcpSocket mSock;
+    //private LlcpSocket mSock;
+    private Socket mSock;
+    private InputStream sockInput;

-        ConnectionThread(LlcpSocket sock) {
+    ConnectionThread(Socket sock) {
             super(TAG);
             mSock = sock;
         }
@@ -80,9 +86,12 @@
                 // Get raw data from remote server
                 while(!connectionBroken) {
                     try {
-                        size = mSock.receive(partial);
+              //size = mSock.receive(partial);
```

```
+              sockInput = mSock.getInputStream();
+              size = sockInput.read(partial);
                       if (DBG) Log.d(TAG, "read " + size + "
bytes");
                       if (size < 0) {
+       if (DBG) Log.d(TAG, "size is less than 0.");
                           connectionBroken = true;
                           break;
                       } else {
@@ -118,15 +127,20 @@
     /** Server class, used to listen for incoming connection request
*/
     class ServerThread extends Thread {
         boolean mRunning = true;
-        LlcpServerSocket mServerSocket;
+    //LlcpServerSocket mServerSocket;
+    ServerSocket mServerSocket = null;
+    Socket communicationSocket = null;
+    //InputStream sockInput = null;
+    //OutputStream sockOutput = null;
+    int serverPort = 3333;

         @Override
         public void run() {
             while (mRunning) {
                 if (DBG) Log.d(TAG, "about create LLCP service
socket");
                 try {
-                    mServerSocket =
mService.createLlcpServerSocket(mSap, SERVICE_NAME,
-                            MIU, 1, 1024);
+            //mServerSocket = mService.createLlcpServerSocket(mSap,
SERVICE_NAME, MIU, 1, 1024);
+            mServerSocket = new ServerSocket(serverPort);
                     if (mServerSocket == null) {
                         if (DBG) Log.d(TAG, "failed to create LLCP
service socket");
                         return;
@@ -134,26 +148,33 @@
                     if (DBG) Log.d(TAG, "created LLCP service
socket");
                     while (mRunning) {
                         if (DBG) Log.d(TAG, "about to accept");
-                        LlcpSocket communicationSocket =
mServerSocket.accept();
+
+              //LlcpSocket communicationSocket =
mServerSocket.accept();
+              communicationSocket = mServerSocket.accept();
+
                         if (DBG) Log.d(TAG, "accept returned " +
communicationSocket);
+
+              //sockInput = communicationSocket.getInputStream();
```

```
+              //sockOutput = communicationSocket.getOutputStream();
+
                         if (communicationSocket != null) {
                             new
ConnectionThread(communicationSocket).start();
                         }
                     }
                     if (DBG) Log.d(TAG, "stop running");
-                } catch (LlcpException e) {
-                    Log.e(TAG, "llcp error", e);
+          //} catch (LlcpException e) {
+          //  Log.e(TAG, "llcp error", e);
                 } catch (IOException e) {
                     Log.e(TAG, "IO error", e);
                 } finally {
-                    if (mServerSocket != null) {
+            if (communicationSocket != null) {
                         if (DBG) Log.d(TAG, "about to close");
                         try {
-                            mServerSocket.close();
+                communicationSocket.close();
                         } catch (IOException e) {
                             // ignore
                         }
-                        mServerSocket = null;
+              communicationSocket = null;
                     }
                 }
             }
Only in
/home/tman144566/WORKING_DIRECTORY/packages/apps/Nfc/src/com/android/nf
c/ndefpush: NdefPushServer.java~
Only in
/home/tman144566/WORKING_DIRECTORY/packages/apps/Nfc/src/com/android/nf
c/ndefpush: Text.java
Only in
/home/tman144566/WORKING_DIRECTORY/packages/apps/Nfc/src/com/android/nf
c/ndefpush: Text.java~
diff -bur
/home/tman144566/Thesis/Nfc/src/com/android/nfc/NfcDispatcher.java
/home/tman144566/WORKING_DIRECTORY/packages/apps/Nfc/src/com/android/nf
c/NfcDispatcher.java
--- /home/tman144566/Thesis/Nfc/src/com/android/nfc/NfcDispatcher.java
2014-07-08 23:44:55.910104453 -0400
+++
/home/tman144566/WORKING_DIRECTORY/packages/apps/Nfc/src/com/android/nf
c/NfcDispatcher.java 2013-09-19 21:54:29.176186368 -0400
@@ -135,6 +135,7 @@
      *  which launches the passed-in intent as soon as it's created.
      */
     private boolean startRootActivity(Intent intent) {
+        Log.i(TAG, "Intent=" + intent);
         Intent rootIntent = new Intent(mContext,
NfcRootActivity.class);
```

```
         rootIntent.putExtra(NfcRootActivity.EXTRA_LAUNCH_INTENT,
intent);
         rootIntent.setFlags(Intent.FLAG_ACTIVITY_NEW_TASK |
Intent.FLAG_ACTIVITY_CLEAR_TASK);
@@ -307,6 +308,7 @@
                             if (firstPackage == null) {
                                 firstPackage = pkg;
                             }
+                            Log.i(TAG, "NfcDispatcher package name: "
+ pkg);
                             intent.setPackage(pkg);
                             if (startRootActivity(intent)) {
                                 return true;
Only in
/home/tman144566/WORKING_DIRECTORY/packages/apps/Nfc/src/com/android/nf
c: NfcNService.java
diff -bur
/home/tman144566/Thesis/Nfc/src/com/android/nfc/NfcService.java
/home/tman144566/WORKING_DIRECTORY/packages/apps/Nfc/src/com/android/nf
c/NfcService.java
--- /home/tman144566/Thesis/Nfc/src/com/android/nfc/NfcService.java
2014-07-08 23:44:55.910104453 -0400
+++
/home/tman144566/WORKING_DIRECTORY/packages/apps/Nfc/src/com/android/nf
c/NfcService.java    2013-09-28 21:41:03.541870462 -0400
@@ -23,6 +23,7 @@
 import com.android.nfc.DeviceHost.TagEndpoint;
 import com.android.nfc.nxp.NativeNfcManager;
 import com.android.nfc.nxp.NativeNfcSecureElement;
+import com.android.nfc.nxp.NativeP2pDevice;

 import android.app.Application;
 import android.app.KeyguardManager;
@@ -66,6 +67,7 @@
 import android.provider.Settings;
 import android.util.Log;

+
 import java.io.FileDescriptor;
 import java.io.IOException;
 import java.io.PrintWriter;
@@ -74,11 +76,14 @@
 import java.util.HashSet;
 import java.util.List;
 import java.util.concurrent.ExecutionException;
+import java.util.ArrayList;
+import java.math.BigInteger;
+import android.app.ActivityManager.RunningTaskInfo;

 public class NfcService extends Application implements
DeviceHostListener {
     private static final String ACTION_MASTER_CLEAR_NOTIFICATION =
"android.intent.action.MASTER_CLEAR_NOTIFICATION";
```

```
-    static final boolean DBG = false;
+    static final boolean DBG = true;
     static final String TAG = "NfcService";

     public static final String SERVICE_NAME = "nfc";
@@ -216,6 +221,14 @@
     private PowerManager mPowerManager;
     private KeyguardManager mKeyguard;

+    List<NdefMessage> ndefarray = new ArrayList<NdefMessage>(); //nate
+    //List<INdefPushCallback> nfccallback = new
ArrayList<INdefPushCallback>();
+    INdefPushCallback nfccallback; //nate
+    NdefMessage nfcstaticmessage;
+    NdefMessage tmpMessage;
+    List<String> pkgarray = new ArrayList<String>();
+    private HashMap<String, INdefPushCallback> callbackHashMap;
+
     private static NfcService sService;

     public static void enforceAdminPerm(Context context) {
@@ -303,6 +316,8 @@
     public void onCreate() {
         super.onCreate();

+        callbackHashMap = new HashMap<String, INdefPushCallback>();
+
         mNfcTagService = new TagService();
         mNfcAdapter = new NfcAdapterService();
         mExtrasService = new NfcAdapterExtrasService();
@@ -391,13 +406,13 @@
     }

     int checkScreenState() {
-        if (!mPowerManager.isScreenOn()) {
-            return SCREEN_STATE_OFF;
-        } else if (mKeyguard.isKeyguardLocked()) {
-            return SCREEN_STATE_ON_LOCKED;
-        } else {
+        //^if (!mPowerManager.isScreenOn()) {
+        //^    return SCREEN_STATE_OFF;
+        //^} else if (mKeyguard.isKeyguardLocked()) {
+        //^    return SCREEN_STATE_ON_LOCKED;
+        //^} else {
             return SCREEN_STATE_ON_UNLOCKED;
-        }
+        //^}
     }

     /**
@@ -471,11 +486,11 @@
                         Log.i(TAG, "First Boot");
                         mPrefsEditor.putBoolean(PREF_FIRST_BOOT,
false);
```

```
                        mPrefsEditor.apply();
-                       executeEeWipe();
+                       //executeEeWipe();nate-4
                    }
                    break;
                case TASK_EE_WIPE:
-                   executeEeWipe();
+                   //executeEeWipe();nate-5
                    break;
            }

@@ -495,11 +510,12 @@
            Log.i(TAG, "Enabling NFC");
            updateState(NfcAdapter.STATE_TURNING_ON);

-           if (!mDeviceHost.initialize()) {
+           if (false) {  //nate1-removed this
!mDeviceHost.initialize() r/w false
                Log.w(TAG, "Error enabling NFC");
                updateState(NfcAdapter.STATE_OFF);
                return false;
            }
+           Log.i(TAG, "Hacking NFC Hardware");

            synchronized(NfcService.this) {
                mObjectMap.clear();
@@ -745,12 +761,93 @@
        }

        @Override
-       public void setForegroundNdefPush(NdefMessage msg,
INdefPushCallback callback) {
+       public void setForegroundNdefPush(NdefMessage msg,
INdefPushCallback callback, String task) {
            mContext.enforceCallingOrSelfPermission(NFC_PERM,
NFC_PERM_ERROR);
-           mP2pLinkManager.setNdefToSend(msg, callback);
+           Log.i(TAG, "Is this called when applications use it?");
+
+           //nfccallback = callback; //nate
+           //nfcstaticmessage = msg; //nate
+           if(msg != null) {
+               nfcstaticmessage = msg;
+           }
+           if(callback != null) {
+               if(!task.equals("nReady")) {
+                   callbackHashMap.put(new String(task), callback);
+                   for (HashMap.Entry<String, INdefPushCallback>
entry: callbackHashMap.entrySet()) {
+                       String key = entry.getKey();
+                       INdefPushCallback calltest = entry.getValue();
+                       Log.i("NATE", "Key: " + key + " Callback: " +
calltest);
+                       /*
```

```
+                        try{
+                        byte[] bytes =
entry.getValue().createNMessage().toByteArray();
+                        BigInteger bi = new BigInteger(bytes);
+                        String s1 = bi.toString(16);
+                        String s = new String(bytes);
+                        Log.i("NATE", "NDEF RECORD :" + s);
+                        Log.i("NATE", "NDEF RECORD :" + s1);
+                        }
+                        catch(RemoteException e) {
+                        Log.i("NATE", "RemoteException");
+                        }
+                        */
+                    }
+                }
+                nfccallback = callback;
+                Log.i(TAG, "definitely not null");
+            }
+            else {
+                Log.i(TAG, "definitely null");
+            }
+
+            mP2pLinkManager.setNdefToSend(msg, callback,
callbackHashMap);
         }

         @Override
+        public void passNdefMessage() {
+            Log.i(TAG, "passNdefMessage");
+            NdefMessage tmpMessage = nfcstaticmessage;
+            try {
+                if(nfccallback != null) {
+                    tmpMessage = nfccallback.createNMessage();
+                    Log.i(TAG, "passNdefMessage message creation");
+                }
+            }
+            catch (RemoteException e) {
+                Log.i(TAG, "RemoteException");
+            }
+
+        }
+
+        /*
+        @Override
+        public void passNdefMessage(NdefMessage smsg, NdefMessage
cmsg) {
+            mContext.enforceCallingOrSelfPermission(NFC_PERM,
NFC_PERM_ERROR);
+            Log.i(TAG, "passNdefMessage");
+            ndefarray.clear();
+            if(smsg != null) {
+                ndefarray.add(smsg);
+            }
+            else if(cmsg != null) {
```

```
+                ndefarray.add(cmsg);
+            }
+            if(!ndefarray.isEmpty()) {
+                if(ndefarray.get(0) != null) {
+                byte[] bytes = ndefarray.get(0).toByteArray();
+                BigInteger bi = new BigInteger(bytes);
+                String s1 = bi.toString(16);
+                String s = new String(bytes);
+                Log.i(TAG, "NDEF RECORD :" + s);
+                Log.i(TAG, "NDEF RECORD :" + s1);
+            }
+            }
+
+            if(smsg == null || cmsg == null) {
+                Log.i(TAG, "Messages are null");
+            }
+        }
+        */
+
+        @Override
         public INfcTag getNfcTagInterface() throws RemoteException {
             mContext.enforceCallingOrSelfPermission(NFC_PERM,
NFC_PERM_ERROR);
             return mNfcTagService;
@@ -1377,9 +1474,9 @@
                 if (POLLING_MODE > SCREEN_STATE_OFF) {
                     if (force || mNfcPollingEnabled) {
                         Log.d(TAG, "NFC-C OFF, disconnect");
-                        mNfcPollingEnabled = false;
-                        mDeviceHost.disableDiscovery();
-                        maybeDisconnectTarget();
+                        //^mNfcPollingEnabled = false;
+                        //^mDeviceHost.disableDiscovery();
+                        //^maybeDisconnectTarget();
                     }
                 }
                 if (mEeRoutingState == ROUTE_ON_WHEN_SCREEN_ON) {
@@ -1413,7 +1510,9 @@
                 if (force || !mNfcPollingEnabled) {
                     Log.d(TAG, "NFC-C ON");
                     mNfcPollingEnabled = true;
-                    mDeviceHost.enableDiscovery();
+                    //mDeviceHost.enableDiscovery();
+                    //onLlcpLinkActivated(new NativeP2pDevice());
+                    Log.i(TAG, "Something should at least happen
here");
                 }
             } else {
                 if (force || mNfcPollingEnabled) {
@@ -1492,7 +1591,7 @@
         return mDeviceHost.createLlcpServerSocket(sap, sn, miu, rw,
linearBufferLength);
     }
```

```
-    public void sendMockNdefTag(NdefMessage msg) {
+    public void sendMockNdefTag(NdefMessage[] msg) {
         sendMessage(MSG_MOCK_NDEF, msg);
     }

@@ -1506,11 +1605,13 @@
     final class NfcServiceHandler extends Handler {
         @Override
         public void handleMessage(Message msg) {
-            switch (msg.what) {
+            switch (msg.what) { //nate-3 removed this msg.what r/w
MSG_NDEF_TAG
                 case MSG_MOCK_NDEF: {
-                    NdefMessage ndefMsg = (NdefMessage) msg.obj;
+                    NdefMessage[] ndefMsg = (NdefMessage[]) msg.obj;
+                    boolean delivered = false;
+                    for (int i = 0; i < ndefMsg.length; i++){
                     Bundle extras = new Bundle();
-                    extras.putParcelable(Ndef.EXTRA_NDEF_MSG,
ndefMsg);
+                        extras.putParcelable(Ndef.EXTRA_NDEF_MSG,
ndefMsg[i]);
                     extras.putInt(Ndef.EXTRA_NDEF_MAXLENGTH, 0);
                     extras.putInt(Ndef.EXTRA_NDEF_CARDSTATE,
Ndef.NDEF_MODE_READ_ONLY);
                     extras.putInt(Ndef.EXTRA_NDEF_TYPE,
Ndef.TYPE_OTHER);
@@ -1519,8 +1620,16 @@
                             new Bundle[] { extras });
                     Log.d(TAG, "mock NDEF tag, starting corresponding
activity");
                     Log.d(TAG, tag.toString());
-                    boolean delivered =
mNfcDispatcher.dispatchTag(tag,
-                            new NdefMessage[] { ndefMsg });
+                        delivered = mNfcDispatcher.dispatchTag(tag,
+                                new NdefMessage[] { ndefMsg[i] });
+                        try {
+                                Thread.sleep(1000);
+                            }
+                            catch(InterruptedException e) {
+                                Log.i(TAG, "InterruptedException");
+                            }
+                    }
+                    Log.i(TAG, "Finished looping through intents");
                     if (delivered) {
                         playSound(SOUND_END);
                     } else {
@@ -1618,7 +1727,7 @@
                             }
                         }
                     }
-                    if (needsDisconnect) {
+                    if (false) { //nate-needsDisconnect
```

```
                        device.disconnect();  // restarts polling loop
                    }

@@ -1676,8 +1785,10 @@

         private boolean llcpActivated(NfcDepEndpoint device) {
             Log.d(TAG, "LLCP Activation message");
+            String s = Integer.toString(device.getMode());
+            Log.i(TAG, s);

-            if (device.getMode() == NfcDepEndpoint.MODE_P2P_TARGET) {
+            if (false) { //nate-device.getMode() ==
NfcDepEndpoint.MODE_P2P_TARGET
                 if (DBG) Log.d(TAG,
"NativeP2pDevice.MODE_P2P_TARGET");
                 if (device.connect()) {
                     /* Check LLCP compliancy */
@@ -1707,12 +1818,12 @@
                      * doConnect
                      */
                 }
-            } else if (device.getMode() ==
NfcDepEndpoint.MODE_P2P_INITIATOR) {
+            } else if (true) {//nate-device.getMode() ==
NfcDepEndpoint.MODE_P2P_INITIATOR
                 if (DBG) Log.d(TAG,
"NativeP2pDevice.MODE_P2P_INITIATOR");
                 /* Check LLCP compliancy */
-                if (mDeviceHost.doCheckLlcp()) {
+                if (true) {//nate-mDeviceHost.doCheckLlcp()
                     /* Activate LLCP Link */
-                    if (mDeviceHost.doActivateLlcp()) {
+                    if (true) {//nate-mDeviceHost.doActivateLlcp()
                         if (DBG) Log.d(TAG, "Target Activate LLCP
OK");
                         synchronized (NfcService.this) {
                             // Register P2P device
@@ -1896,4 +2007,83 @@

         }
     }
+
+    public INdefPushCallback callbackreturn() {
+
+        if(nfccallback != null) {
+            Log.i(TAG, "Callback is not null");
+            return this.nfccallback;
+        }
+        else {
+        Log.i(TAG, "Callback is null");
+        }
+
+        return null;
+
```

```
+    }
+
+    public void setndefarray(NdefMessage message,
List<RunningTaskInfo> pkgs) {
+        //this.ndefarray.clear();
+        Integer addmessage = 1;
+        String pkg = null;
+        if(!pkgs.isEmpty()) {
+            for (int i = 0; i < pkgs.size(); i++) {
+
if(!pkgs.get(i).baseActivity.getPackageName().equals("android") &&
!pkgs.get(i).baseActivity.getPackageName().equals
("com.android.launcher")) {
+                    pkg = pkgs.get(i).baseActivity.getPackageName();
+                    break;
+                }
+            }
+        }
+        Log.i(TAG, "Printing out package name passed: " + pkg);
+        if(!pkgarray.isEmpty()) {
+            for (int i = 0; i < pkgarray.size(); i++) {
+                if(pkgarray.get(i).equals(pkg)) {
+                    addmessage = 0;
+                    this.ndefarray.set(i, message);
+                    Log.i(TAG, "String Matches Tag: " + i);
+                }
+                Log.i(TAG, pkgarray.get(i));
+            }
+        }
+        Log.i(TAG, "pkgarray size before: " +
Integer.toString(pkgarray.size()));
+        if(addmessage == 1 && pkg != null) {
+            Log.i(TAG, "Adding message to array");
+            this.pkgarray.add(pkg);
+            this.ndefarray.add(message);
+        }
+        Log.i(TAG, "pkgarray size after: " +
Integer.toString(pkgarray.size()));
+        mP2pLinkManager.setNNdefToSend(this.nfcstaticmessage,
this.ndefarray, this.nfccallback);
+        nfccallback = null;
+        Log.i(TAG, "Number of Messages = " +
Integer.toString(this.ndefarray.size()));
+
+        for (int i = 0; i < this.ndefarray.size(); i++) {
+            byte[] bytes = this.ndefarray.get(i).toByteArray();
+            BigInteger bi = new BigInteger(bytes);
+            String s1 = bi.toString(16);
+            String s = new String(bytes);
+            Log.i(TAG, "NDEF RECORD :" + s);
+            Log.i(TAG, "NDEF RECORD :" + s1);
+        }
+
+
```

```
+
+    }
+
+    public void setmanualtrigger(int i) {
+        mP2pLinkManager.manualtriggerreturn(i);
+        if(i == 0) {
+            this.nfcstaticmessage = null;
+            this.nfccallback = null;
+            this.ndefarray.clear();
+            this.pkgarray.clear();
+            mP2pLinkManager.setNNdefToSend(this.nfcstaticmessage,
this.ndefarray, this.nfccallback);
+        }
+    }
+    /*
+    public int manualtriggerreturn() {
+        return this.manualtrigger;
+    }
+    */
+
+
 }
Only in
/home/tman144566/WORKING_DIRECTORY/packages/apps/Nfc/src/com/android/nf
c: NfcService.java~
Only in
/home/tman144566/WORKING_DIRECTORY/packages/apps/Nfc/src/com/android/nf
c: NfcTrigger.java
Only in
/home/tman144566/WORKING_DIRECTORY/packages/apps/Nfc/src/com/android/nf
c: P2pEventManager.java~
diff -bur
/home/tman144566/Thesis/Nfc/src/com/android/nfc/P2pLinkManager.java
/home/tman144566/WORKING_DIRECTORY/packages/apps/Nfc/src/com/android/nf
c/P2pLinkManager.java
--- /home/tman144566/Thesis/Nfc/src/com/android/nfc/P2pLinkManager.java
2014-07-08 23:44:55.910104453 -0400
+++
/home/tman144566/WORKING_DIRECTORY/packages/apps/Nfc/src/com/android/nf
c/P2pLinkManager.java    2013-09-29 01:00:40.001930366 -0400
@@ -48,6 +48,17 @@
 import java.nio.charset.Charsets;
 import java.util.Arrays;
 import java.util.List;
+import java.util.ArrayList;
+
+import android.net.wifi.WifiManager;
+import android.net.wifi.WifiManager.WifiLock;
+import android.os.PowerManager;
+import android.os.PowerManager.WakeLock;
+import android.net.ConnectivityManager;
+import java.lang.reflect.Field;
+import java.lang.reflect.Method;
+import java.math.BigInteger;
```

```
+import java.util.HashMap;

 /**
  * Interface to listen for P2P events.
@@ -134,12 +145,17 @@
     int mSendState;  // valid during LINK_STATE_UP or
LINK_STATE_DEBOUNCE
     boolean mIsSendEnabled;
     boolean mIsReceiveEnabled;
-    NdefMessage mMessageToSend;  // valid during
SEND_STATE_NEED_CONFIRMATION or SEND_STATE_SENDING
+    List<NdefMessage> mMessageToSend;  // valid during
SEND_STATE_NEED_CONFIRMATION or SEND_STATE_SENDING
     NdefMessage mStaticNdef;
     INdefPushCallback mCallbackNdef;
+    INdefPushCallback mnfccallback;
+    List<NdefMessage> mndefarray;
     SendTask mSendTask;
     SharedPreferences mPrefs;
     boolean mFirstBeam;
+    int manualtrigger = 0;
+    NdefMessage mnfcstaticmessage;
+    HashMap<String, INdefPushCallback> mHashCallbacks;

     public P2pLinkManager(Context context) {
         mNdefPushServer = new NdefPushServer(NDEFPUSH_SAP,
mNppCallback);
@@ -149,7 +165,7 @@
         mContext = context;
         mEventListener = new P2pEventManager(context, this);
         mHandler = new Handler(this);
-        mLinkState = LINK_STATE_DOWN;
+        mLinkState = LINK_STATE_DOWN;//nate-LINK_STATE_DOWN
         mSendState = SEND_STATE_NOTHING_TO_SEND;
         mIsSendEnabled = false;
         mIsReceiveEnabled = false;
@@ -164,10 +180,10 @@
     public void enableDisable(boolean sendEnable, boolean
receiveEnable) {
         synchronized (this) {
             if (!mIsReceiveEnabled && receiveEnable) {
-                mDefaultSnepServer.start();
+                //mDefaultSnepServer.start();
                 mNdefPushServer.start();
             } else if (mIsReceiveEnabled && !receiveEnable) {
-                mDefaultSnepServer.stop();
+                //mDefaultSnepServer.stop();
                 mNdefPushServer.stop();
             }
             mIsSendEnabled = sendEnable;
@@ -182,10 +198,27 @@
      * currently off or P2P send is currently off). They will become
      * active as soon as P2P send is enabled.
      */
```

```
-    public void setNdefToSend(NdefMessage staticNdef,
INdefPushCallback callbackNdef) {
+    public void setNdefToSend(NdefMessage staticNdef,
INdefPushCallback callbackNdef, HashMap<String, INdefPushCallback>
hashcallbacks) {
         synchronized (this) {
             mStaticNdef = staticNdef;
             mCallbackNdef = callbackNdef;
+            mHashCallbacks = hashcallbacks;
+
+        }
+    }
+
+    public void setNNdefToSend(NdefMessage Nmsg, List<NdefMessage>
Nndef, INdefPushCallback Ncallback) {
+        synchronized (this) {
+
+            mndefarray = Nndef;
+            mnfcstaticmessage = Nmsg;
+            mnfccallback = Ncallback;
+        }
+    }
+
+    public void manualtriggerreturn(int i) {
+        synchronized(this) {
+            manualtrigger = i;
         }
     }

@@ -194,7 +227,7 @@
      */
     public void onLlcpActivated() {
         Log.i(TAG, "LLCP activated");
-
+        mLinkState = LINK_STATE_DOWN;
         synchronized (P2pLinkManager.this) {
             switch (mLinkState) {
                 case LINK_STATE_DOWN:
@@ -204,7 +237,7 @@
                     mEventListener.onP2pInRange();

                     prepareMessageToSend();
-                    if (mMessageToSend != null) {
+                    if (mMessageToSend != null) {//nate-mMessageToSend
!= null
                         mSendState = SEND_STATE_NEED_CONFIRMATION;
                         if (DBG) Log.d(TAG,
"onP2pSendConfirmationRequested()");

mEventListener.onP2pSendConfirmationRequested();
@@ -228,26 +261,84 @@

     void prepareMessageToSend() {
         synchronized (P2pLinkManager.this) {
```

```
-             if (!mIsSendEnabled) {
+             if (!mIsSendEnabled) {//nate-!mIsSendEnabled
+                 //mMessageToSend = null;
+                 mMessageToSend.clear();
                  mMessageToSend = null;
                  return;
              }
-
-             NdefMessage messageToSend = mStaticNdef;
+             Log.i(TAG, "manualtrigger =" +
Integer.toString(manualtrigger));
+             //NdefMessage messageToSend = mStaticNdef;
+             List<NdefMessage> messageToSend = new
ArrayList<NdefMessage>();
+             messageToSend.add(0, mStaticNdef);
              INdefPushCallback callback = mCallbackNdef;

-             if (callback != null) {
+             Log.i(TAG, "callback = " + mnfccallback);
+
+             if(manualtrigger == 1) {
+                 messageToSend.set(0, mnfcstaticmessage);
+             }
+
+             if (callback != null && manualtrigger == 0) {
                  try {
-                     messageToSend = callback.createMessage();
+                     //messageToSend = callback.createMessage();
+                     messageToSend.set(0, callback.createMessage());
+                     Log.i(TAG, "Ndef Callback Message");
                  } catch (RemoteException e) {
                      // Ignore
                  }
              }

-             if (messageToSend == null) {
-                 messageToSend = createDefaultNdef();
+             if(mnfccallback != null && manualtrigger == 1 &&
!mndefarray.isEmpty()) {
+                 //messageToSend = mndefarray.get(0);
+                 messageToSend = mndefarray;
+                 Log.i(TAG, "Assigning ndefarray to messageToSend");
+
              }
+
+             //should add manualtrigger default ndef
+
+             if (messageToSend.get(0) == null && manualtrigger == 0)
{//nate-messageToSend == null
+                 //messageToSend = createDefaultNdef();
+                 messageToSend.set(0, createDefaultNdef());
+                 Log.i(TAG, "Created default Ndef, non-trigger");
+             }
+
```

```
+            if(messageToSend.get(0) == null && manualtrigger == 1) {
+                //special default function
+                //messageToSend = createDefaultNdef();
+                messageToSend.set(0, createDefaultNdef());
+                Log.i(TAG, "Created default Ndef, trigger");
+            }
+            if(manualtrigger == 1) {
+                    for (HashMap.Entry<String, INdefPushCallback>
entry: mHashCallbacks.entrySet()) {
+                        String key = entry.getKey();
+                        INdefPushCallback calltest = entry.getValue();
+                        Log.i("NATE1", "Key: " + key + " Callback: " +
calltest);
+                        try{
+                        byte[] bytes =
entry.getValue().createNMessage().toByteArray();
+                        messageToSend.set(0,
entry.getValue().createNMessage());
+                        BigInteger bi = new BigInteger(bytes);
+                        String s1 = bi.toString(16);
+                        String s = new String(bytes);
+                        Log.i("NATE1", "NDEF RECORD :" + s);
+                        Log.i("NATE1", "NDEF RECORD :" + s1);
+                        }
+                        catch(RemoteException e) {
+                        Log.i("NATE1", "RemoteException: " + e);
+                        messageToSend.set(0, createDefaultNdef());
+                        }
+                        catch(NullPointerException e) {
+                        Log.i("NATE1","NullPointerException: " + e);
+                        messageToSend.set(0, createDefaultNdef());
+                        }
+                    }
+            }
+
             mMessageToSend = messageToSend;
+
+            manualtrigger = 0;
+
         }
     }

@@ -255,12 +346,18 @@
         List<RunningTaskInfo> tasks =
mActivityManager.getRunningTasks(1);
         if (tasks.size() > 0) {
             String pkg = tasks.get(0).baseActivity.getPackageName();
+            Log.i(TAG, pkg);
             try {
                 ApplicationInfo appInfo =
mPackageManager.getApplicationInfo(pkg, 0);
-                if (0 == (appInfo.flags &
ApplicationInfo.FLAG_SYSTEM)) {
+                String s = Integer.toString(appInfo.flags);
```

```
+                String s1 =
Integer.toString(ApplicationInfo.FLAG_SYSTEM);
+                Log.i(TAG, s);
+                Log.i(TAG, s1);
+                if (true) {//nate- 0 == (appInfo.flags &
ApplicationInfo.FLAG_SYSTEM)
                     NdefRecord appUri = NdefRecord.createUri(

Uri.parse("http://market.android.com/search?q=pname:" + pkg));
                     NdefRecord appRecord =
NdefRecord.createApplicationRecord(pkg);
+                    Log.i(TAG, "Are we returning null or not");
                     return new NdefMessage(new NdefRecord[] { appUri,
appRecord });
                 }
             } catch (NameNotFoundException e) {
@@ -293,14 +390,16 @@
         }
     }

-    void onSendComplete(NdefMessage msg, long elapsedRealtime) {
+    void onSendComplete(NdefMessage[] msg, long elapsedRealtime) {
         if (mFirstBeam) {
             EventLogTags.writeNfcFirstShare();
             mPrefs.edit().putBoolean(NfcService.PREF_FIRST_BEAM,
false).apply();
             mFirstBeam = false;
         }
-        EventLogTags.writeNfcShare(getMessageSize(msg),
getMessageTnf(msg), getMessageType(msg),
-                getMessageAarPresent(msg), (int) elapsedRealtime);
+        for (int i = 0; i < msg.length; i++) {
+            EventLogTags.writeNfcShare(getMessageSize(msg[i]),
getMessageTnf(msg[i]), getMessageType(msg[i]),
+                    getMessageAarPresent(msg[i]), (int)
elapsedRealtime);
+        }
         // Make callbacks on UI thread
         mHandler.sendEmptyMessage(MSG_SEND_COMPLETE);
     }
@@ -324,17 +423,28 @@
     final class SendTask extends AsyncTask<Void, Void, Void> {
         @Override
         public Void doInBackground(Void... args) {
-            NdefMessage m;
+            //NdefMessage m;
+            NdefMessage[] m = new NdefMessage[mMessageToSend.size()];
             boolean result;

             synchronized (P2pLinkManager.this) {
                 if (mLinkState != LINK_STATE_UP || mSendState !=
SEND_STATE_SENDING) {
                     return null;
                 }
```

```
-                 m = mMessageToSend;
+                 mMessageToSend.toArray(m);
              }
-
              long time = SystemClock.elapsedRealtime();
+             Log.i(TAG, "We have reached the push");
+             for (int i = 0; i < m.length; i++) {
+                 byte[] bytes = m[i].toByteArray();
+                 BigInteger bi = new BigInteger(bytes);
+                 String s1 = bi.toString(16);
+                 String s = new String(bytes);
+                 Log.i(TAG, "NDEF RECORD :" + s);
+                 Log.i(TAG, "NDEF RECORD :" + s1);
+             }
+             result = new NdefPushClient().push(m);
+             /*
              try {
                  if (DBG) Log.d(TAG, "Sending ndef via SNEP");
                  result = doSnepProtocol(m);
@@ -347,17 +457,90 @@

                  result = new NdefPushClient().push(m);
              }
+        */
              time = SystemClock.elapsedRealtime() - time;

              if (DBG) Log.d(TAG, "SendTask result=" + result + ", time
ms=" + time);

              if (result) {
                  onSendComplete(m, time);
+
+        //setMobileDataEnabled(mContext, false);
              }
              return null;
          }
      }

+   private void setMobileDataEnabled(Context context, boolean enabled)
{
+
+       /* PowerManager pM = (PowerManager)
context.getSystemService(Context.POWER_SERVICE);
+        WakeLock wl = pM.newWakeLock(PowerManager.FULL_WAKE_LOCK,
"wakeLock");
+        wl.acquire();
+        wl.setReferenceCounted(false);
+       */
+
+        WifiManager wifiManager = (WifiManager)
context.getSystemService(Context.WIFI_SERVICE);
+        if (DBG) Log.d(TAG, "WifiState: " +
wifiManager.getWifiState());
```

```
+       //WifiLock wifiLock =
wifiManager.createWifiLock(WifiManager.WIFI_STATE_DISABLED,
"MyWifiLock");
+       //wifiLock.acquire();
+       //wifiLock.setReferenceCounted(false);
+
+       //ConnectivityManager cm = (ConnectivityManager)
context.getSystemService(Context.CONNECTIVITY_SERVICE);
+       //if (DBG) Log.d(TAG, "Connectivity WiFiInfo: " +
cm.getNetworkInfo(ConnectivityManager.TYPE_WIFI).isConnectedOrConnectin
g());
+
+       boolean successful = wifiManager.setWifiEnabled(false);
+       if (DBG) Log.d(TAG, "WifiDisabling: " + successful);
+       //if (DBG) Log.d(TAG, "Connectivity WiFiInfo: " +
cm.getNetworkInfo(ConnectivityManager.TYPE_WIFI).isConnectedOrConnectin
g());
+       //wifiLock.release();
+       wifiManager.disconnect();
+
+       //wl.release();
+
+       if (DBG) Log.d(TAG, "WifiState: " +
wifiManager.getWifiState());
+
+       /*try {
+       ConnectivityManager conman = (ConnectivityManager)
context.getSystemService(Context.CONNECTIVITY_SERVICE);
+       Class conmanClass = Class.forName(conman.getClass().getName());
+       Field iConnectivityManagerField =
conmanClass.getDeclaredField("mService");
+       iConnectivityManagerField.setAccessible(true);
+       final Object iConnectivityManager =
iConnectivityManagerField.get(conman);
+       final Class iConnectivityManagerClass =
Class.forName(iConnectivityManager.getClass().getName());
+       final Method setMobileDataEnabledMethod =
iConnectivityManagerClass.getDeclaredMethod("setMobileDataEnabled",
Boolean.TYPE);
+       setMobileDataEnabledMethod.setAccessible(true);
+
+           setMobileDataEnabledMethod.invoke(iConnectivityManager,
enabled);
+
+        } catch (ClassNotFoundException e) {
+           // TODO Auto-generated catch block
+           e.printStackTrace();
+        } catch (SecurityException e) {
+           // TODO Auto-generated catch block
+           e.printStackTrace();
+        } catch (NoSuchFieldException e) {
+           // TODO Auto-generated catch block
+           e.printStackTrace();
+        } catch (IllegalArgumentException e) {
```

```
+            // TODO Auto-generated catch block
+            e.printStackTrace();
+        } catch (IllegalAccessException e) {
+            // TODO Auto-generated catch block
+            e.printStackTrace();
+        } catch (NoSuchMethodException e) {
+            // TODO Auto-generated catch block
+            e.printStackTrace();
+        //} catch (InvocationTargetException e) {
+            // TODO Auto-generated catch block
+        //      e.printStackTrace();
+        } catch (Exception e) {
+            // TODO Auto-generated catch block
+            e.printStackTrace();
+        }
+        finally {
+
+        }
+   */
+    }
+
     static boolean doSnepProtocol(NdefMessage msg) throws IOException
{
         SnepClient snepClient = new SnepClient();
         try {
@@ -381,14 +564,14 @@

     final NdefPushServer.Callback mNppCallback = new
NdefPushServer.Callback() {
         @Override
-        public void onMessageReceived(NdefMessage msg) {
+        public void onMessageReceived(NdefMessage[] msg) {
             onReceiveComplete(msg);
         }
     };

     final SnepServer.Callback mDefaultSnepCallback = new
SnepServer.Callback() {
         @Override
-        public SnepMessage doPut(NdefMessage msg) {
+        public SnepMessage doPut(NdefMessage[] msg) {
             onReceiveComplete(msg);
             return
SnepMessage.getMessage(SnepMessage.RESPONSE_SUCCESS);
         }
@@ -400,9 +583,11 @@
         }
     };

-    void onReceiveComplete(NdefMessage msg) {
-        EventLogTags.writeNfcNdefReceived(getMessageSize(msg),
getMessageTnf(msg),
-                getMessageType(msg), getMessageAarPresent(msg));
+    void onReceiveComplete(NdefMessage[] msg) {
```

```
+        for (int i = 0; i < msg.length; i++) {
+            EventLogTags.writeNfcNdefReceived(getMessageSize(msg[i]),
getMessageTnf(msg[i]),
+                    getMessageType(msg[i]),
getMessageAarPresent(msg[i]));
+        }
         // Make callbacks on UI thread
         mHandler.obtainMessage(MSG_RECEIVE_COMPLETE,
msg).sendToTarget();
     }
@@ -415,24 +600,31 @@
                     if (mLinkState != LINK_STATE_DEBOUNCE) {
                         break;
                     }
+                    if(mMessageToSend != null) {
+                        for (int i = 0; i < mMessageToSend.size();
i++) {
                     if (mSendState == SEND_STATE_SENDING) {
-
EventLogTags.writeNfcShareFail(getMessageSize(mMessageToSend),
-                                getMessageTnf(mMessageToSend),
getMessageType(mMessageToSend),
-                                getMessageAarPresent(mMessageToSend));
+
EventLogTags.writeNfcShareFail(getMessageSize(mMessageToSend.get(i)),
+
getMessageTnf(mMessageToSend.get(i)),
getMessageType(mMessageToSend.get(i)),
+
getMessageAarPresent(mMessageToSend.get(i)));
+                            }
+                        }
                     }
                     if (DBG) Log.d(TAG, "Debounce timeout");
                     mLinkState = LINK_STATE_DOWN;
                     mSendState = SEND_STATE_NOTHING_TO_SEND;
+                    mMessageToSend.clear();
                     mMessageToSend = null;
                     if (DBG) Log.d(TAG, "onP2pOutOfRange()");
+            setMobileDataEnabled(mContext, true);
                     mEventListener.onP2pOutOfRange();
                 }
                 break;
             case MSG_RECEIVE_COMPLETE:
-                NdefMessage m = (NdefMessage) msg.obj;
+                NdefMessage[] m = (NdefMessage[]) msg.obj;
                 synchronized (this) {
                     if (mLinkState == LINK_STATE_DOWN) {
-                        break;
+                        //nate-break;
+                        mLinkState = LINK_STATE_UP;
                     }
                     if (mSendState == SEND_STATE_SENDING) {
                         cancelSendNdefMessage();
```

```
@@ -526,6 +718,7 @@
     @Override
     public void onP2pSendConfirmed() {
         if (DBG) Log.d(TAG, "onP2pSendConfirmed()");
+
         synchronized (this) {
             if (mLinkState == LINK_STATE_DOWN || mSendState !=
SEND_STATE_NEED_CONFIRMATION) {
                 return;
@@ -572,7 +765,9 @@

             pw.println("mStaticNdef=" + mStaticNdef);
             pw.println("mCallbackNdef=" + mCallbackNdef);
-            pw.println("mMessageToSend=" + mMessageToSend);
+            for (int i = 0; i < mMessageToSend.size(); i++) {
+                pw.println("mMessageToSend " + i + " =" +
mMessageToSend.get(i));
+            }
         }
     }
 }
Only in
/home/tman144566/WORKING_DIRECTORY/packages/apps/Nfc/src/com/android/nf
c: P2pLinkManager.java~
diff -bur /home/tman144566/Thesis/Nfc/src/com/android/nfc/SendUi.java
/home/tman144566/WORKING_DIRECTORY/packages/apps/Nfc/src/com/android/nf
c/SendUi.java
--- /home/tman144566/Thesis/Nfc/src/com/android/nfc/SendUi.java 2014-
07-08 23:44:55.910104453 -0400
+++
/home/tman144566/WORKING_DIRECTORY/packages/apps/Nfc/src/com/android/nf
c/SendUi.java    2013-02-21 15:22:42.000000000 -0500
@@ -205,6 +205,10 @@

     /** Show pre-send animation */
     public void showPreSend() {
+
+   //^>
+   mCallback.onSendConfirmed();
+
         // Update display metrics
         mDisplay.getRealMetrics(mDisplayMetrics);

Only in
/home/tman144566/WORKING_DIRECTORY/packages/apps/Nfc/src/com/android/nf
c: SendUi.java~
diff -bur
/home/tman144566/Thesis/Nfc/src/com/android/nfc/snep/SnepServer.java
/home/tman144566/WORKING_DIRECTORY/packages/apps/Nfc/src/com/android/nf
c/snep/SnepServer.java
---
/home/tman144566/Thesis/Nfc/src/com/android/nfc/snep/SnepServer.java
2014-07-08 23:44:55.910104453 -0400
```

```
+++
/home/tman144566/WORKING_DIRECTORY/packages/apps/Nfc/src/com/android/nf
c/snep/SnepServer.java   2013-08-24 04:35:16.176952699 -0400
@@ -50,7 +50,7 @@
     boolean mServerRunning = false;

     public interface Callback {
-        public SnepMessage doPut(NdefMessage msg);
+        public SnepMessage doPut(NdefMessage[] msg);
         public SnepMessage doGet(int acceptableLength, NdefMessage
msg);
     }

@@ -142,7 +142,7 @@
                     request.getNdefMessage()));
         } else if (request.getField() == SnepMessage.REQUEST_PUT) {
             if (DBG) Log.d(TAG, "putting message " +
request.toString());
-
messenger.sendMessage(callback.doPut(request.getNdefMessage()));
+            messenger.sendMessage(callback.doPut(new NdefMessage[] {
request.getNdefMessage() }));
         } else {
             if (DBG) Log.d(TAG, "Unknown request (" +
request.getField() +")");
             messenger.sendMessage(SnepMessage.getMessage(
Only in /home/tman144566/WORKING_DIRECTORY/packages/apps/Nfc/tests:
assets
Only in /home/tman144566/WORKING_DIRECTORY/packages/apps/Nfc/tests: bin
Only in /home/tman144566/WORKING_DIRECTORY/packages/apps/Nfc/tests: gen
Only in /home/tman144566/WORKING_DIRECTORY/packages/apps/Nfc/tests: res
diff -bur
/home/tman144566/Thesis/Nfc/tests/src/com/android/nfc/snep/SnepBasicTes
ts.java
/home/tman144566/WORKING_DIRECTORY/packages/apps/Nfc/tests/src/com/andr
oid/nfc/snep/SnepBasicTests.java
---
/home/tman144566/Thesis/Nfc/tests/src/com/android/nfc/snep/SnepBasicTes
ts.java  2014-07-08 23:44:03.910848945 -0400
+++
/home/tman144566/WORKING_DIRECTORY/packages/apps/Nfc/tests/src/com/andr
oid/nfc/snep/SnepBasicTests.java 2013-08-24 04:45:15.143774437 -0400
@@ -283,7 +283,7 @@
         private static final int GET_LENGTH = 1024;

         @Override
-        public SnepMessage doPut(NdefMessage msg) {
+        public SnepMessage doPut(NdefMessage[] msg) {
             return SnepMessage.getSuccessResponse(null);
         }

diff -bur
/home/tman144566/Thesis/Nfc/tests/src/com/android/nfc/snep/SnepValidati
onServerTests.java
```

/home/tman144566/WORKING_DIRECTORY/packages/apps/Nfc/tests/src/com/android/nfc/snep/SnepValidationServerTests.java

```
---
/home/tman144566/Thesis/Nfc/tests/src/com/android/nfc/snep/SnepValidationServerTests.java   2014-07-08 23:44:03.910848945 -0400
+++
/home/tman144566/WORKING_DIRECTORY/packages/apps/Nfc/tests/src/com/android/nfc/snep/SnepValidationServerTests.java  2013-08-24 04:47:00.596522133 -0400
@@ -107,12 +107,12 @@
     }

     @Override
-    public SnepMessage doPut(NdefMessage msg) {
+    public SnepMessage doPut(NdefMessage[] msg) {
         Log.d(TAG, "doPut()");
-        NdefRecord record = msg.getRecords()[0];
+        NdefRecord record = msg[0].getRecords()[0];
         ByteArrayWrapper id = (record.getId().length > 0) ?
                 new ByteArrayWrapper(record.getId()) : DEFAULT_NDEF;
-        mStoredNdef.put(id, msg);
+        mStoredNdef.put(id, msg[0]);
         return SnepMessage.getMessage(SnepMessage.RESPONSE_SUCCESS);
     }
```

Appendix D: Android's Browser Diff Output

```
/*--------------------------------------------------------------------------
 * Android's Browser Modifications Diff Output
 *--------------------------------------------------------------------------*/

diff -bur
/home/tman144566/Thesis/Browser/src/com/android/browser/Controller.java
/home/tman144566/WORKING_DIRECTORY/packages/apps/Browser/src/com/androi
d/browser/Controller.java
---
/home/tman144566/Thesis/Browser/src/com/android/browser/Controller.java
2014-07-08 23:23:02.413371916 -0400
+++
/home/tman144566/WORKING_DIRECTORY/packages/apps/Browser/src/com/androi
d/browser/Controller.java    2013-08-22 21:20:05.689986215 -0400
@@ -99,6 +99,16 @@
 import java.util.HashMap;
 import java.util.List;
 import java.util.Map;
+import android.util.Log;
+import android.nfc.NdefMessage;
+import android.nfc.NdefRecord;
+import android.os.Parcelable;
+import java.math.BigInteger;
+import android.nfc.Tag;
+import java.util.Arrays;
+import android.nfc.NfcAdapter;
+import java.nio.charset.Charsets;
+import android.nfc.FormatException;

 /**
  * Controller for browser
@@ -116,6 +126,8 @@
     public final static int LOAD_URL = 1001;
     public final static int STOP_LOAD = 1002;

+    static final String TAG = "BrowserController";
+
     // Message Ids
     private static final int FOCUS_NODE_HREF = 102;
     private static final int RELEASE_WAKELOCK = 107;
@@ -326,6 +338,7 @@
                 }
```

```
             }
             mUi.updateTabs(mTabControl.getTabs());
+            nfcNTabs(intent);
         } else {
             mTabControl.restoreState(icicle, currentTabId,
restoreIncognitoTabs,
                     mUi.needsRestoreAllTabs());
@@ -356,6 +369,7 @@
         if
(BrowserActivity.ACTION_SHOW_BOOKMARKS.equals(intent.getAction())) {
             bookmarksOrHistoryPicker(ComboViews.Bookmarks);
         }
+
     }

     private static class PruneThumbnails implements Runnable {
@@ -2276,8 +2290,17 @@
     public Tab openTab(UrlData urlData) {
         Tab tab = showPreloadedTab(urlData);
         if (tab == null) {
+            Log.i(TAG, "tab is null, creating new tab");
             tab = createNewTab(false, true, true);
+            Log.i(TAG, "finished creating tab");
+            if(!urlData.isEmpty()) {
+                Log.i(TAG, "urlData is not empty");
+            }
+            else {
+                Log.i(TAG, "urlData is empty, load anyways");
+            }
             if ((tab != null) && !urlData.isEmpty()) {
+                Log.i(TAG, "Should be trying to load data in");
                 loadUrlDataIn(tab, urlData);
             }
         }
@@ -2446,6 +2469,7 @@

     protected void loadUrl(Tab tab, String url, Map<String, String>
headers) {
         if (tab != null) {
+            Log.i(TAG, "Website = " + url);
             dismissSubWindow(tab);
             tab.loadUrl(url, headers);
             mUi.onProgressChanged(tab);
@@ -2780,4 +2804,99 @@
         return mBlockEvents;
     }

+    public void nfcNTabs(Intent intent) {
+        NdefMessage[] msgs = null;
+        Parcelable[] rawMsgs =
intent.getParcelableArrayExtra(NfcAdapter.EXTRA_NDEF_MESSAGES);
+        Parcelable rawTag =
intent.getParcelableExtra(NfcAdapter.EXTRA_TAG);
+        Tag tag = (Tag) rawTag;
```

```
+        if(rawMsgs != null) {
+            msgs = new NdefMessage[rawMsgs.length];
+            for(int i = 0; i < rawMsgs.length; i++) {
+                msgs[i] = (NdefMessage) rawMsgs[i];
+                for(int ii = 1; ii < msgs[i].getRecords().length;
ii++) {
+                    Log.i(TAG, "NdefRecord " + ii + "=" +
msgs[i].getRecords()[ii].toString());
+                     byte[] bytes =
msgs[i].getRecords()[ii].toByteArray();
+                            BigInteger bi = new BigInteger(bytes);
+                            String s1 = bi.toString(16);
+                            String s = new String(bytes);
+                            Log.i(TAG, "NDEF RECORD :" + s);
+                            Log.i(TAG, "NDEF RECORD :" + s1);
+
+                            NdefMessage[] newMsgs = { msgs[i] };
+                            Intent newIntent = new
Intent("NfcAdapter.ACTION_NDEF_DISCOVERED");
+                            newIntent.putExtra(NfcAdapter.EXTRA_TAG,
tag);
+                            newIntent.putExtra(NfcAdapter.EXTRA_ID,
tag.getId());
+
newIntent.putExtra(NfcAdapter.EXTRA_NDEF_MESSAGES, newMsgs);
+                            if(setTypeOrDataFromNdef(newIntent,
newMsgs[i].getRecords()[ii])) {
+                                Log.i(TAG, "setTypeOrDataFromNdef is
ture");
+                            }
+                            //UrlData newUrlData =
getUrlDataFromIntent(newIntent);
+                            //mController.openTab(newUrlData);
+                            openTab(newIntent.getData().toString(),
false, true, true);
+                }
+            }
+        }
+    }
+
+    private boolean setTypeOrDataFromNdef(Intent newIntent, NdefRecord
record) {
+        short tnf = record.getTnf();
+        byte[] type = record.getType();
+        try {
+            switch (tnf) {
+                case NdefRecord.TNF_MIME_MEDIA: {
+                    newIntent.setType(new String(type,
Charsets.US_ASCII));
+                    return true;
+                }
+
+                case NdefRecord.TNF_ABSOLUTE_URI: {
```

```
+                    newIntent.setData(Uri.parse(new String(type,
Charsets.UTF_8)));
+                    return true;
+                }
+
+                case NdefRecord.TNF_WELL_KNOWN: {
+                    byte[] payload = record.getPayload();
+                    if (payload == null || payload.length == 0) return
false;
+                    if (Arrays.equals(type, NdefRecord.RTD_TEXT)) {
+                        newIntent.setType("text/plain");
+                        return true;
+                    } else if (Arrays.equals(type,
NdefRecord.RTD_SMART_POSTER)) {
+                        // Parse the smart poster looking for the URI
+                        try {
+                            NdefMessage msg = new
NdefMessage(record.getPayload());
+                            for (NdefRecord subRecord :
msg.getRecords()) {
+                                short subTnf = subRecord.getTnf();
+                                if (subTnf ==
NdefRecord.TNF_WELL_KNOWN
+                                        &&
Arrays.equals(subRecord.getType(),
+                                                NdefRecord.RTD_URI)) {
+
newIntent.setData(NdefRecord.parseWellKnownUriRecord(subRecord));
+                                    return true;
+                                } else if (subTnf ==
NdefRecord.TNF_ABSOLUTE_URI) {
+                                    newIntent.setData(Uri.parse(new
String(subRecord.getType(),
+                                            Charsets.UTF_8)));
+                                    return true;
+                                }
+                            }
+                        } catch (FormatException e) {
+                            return false;
+                        }
+                    } else if (Arrays.equals(type,
NdefRecord.RTD_URI)) {
+
newIntent.setData(NdefRecord.parseWellKnownUriRecord(record));
+                        return true;
+                    }
+                    return false;
+                }
+
+                case NdefRecord.TNF_EXTERNAL_TYPE: {
+
newIntent.setData(Uri.parse("vnd.android.nfc://ext/" +
+                            new String(record.getType(),
Charsets.US_ASCII)));
```

```
+                    return true;
+                }
+            }
+            return false;
+        } catch (Exception e) {
+            Log.e(TAG, "failed to parse record", e);
+            return false;
+        }
+    }
+
 }
```

diff -bur /home/tman144566/Thesis/Browser/src/com/android/browser/IntentHandler.java /home/tman144566/WORKING_DIRECTORY/packages/apps/Browser/src/com/android/browser/IntentHandler.java

```
---
/home/tman144566/Thesis/Browser/src/com/android/browser/IntentHandler.java  2014-07-08 23:23:02.413371916 -0400
+++
/home/tman144566/WORKING_DIRECTORY/packages/apps/Browser/src/com/android/browser/IntentHandler.java 2013-08-21 17:41:32.175725854 -0400
@@ -40,6 +40,17 @@
 import java.util.HashMap;
 import java.util.Iterator;
 import java.util.Map;
+import android.util.Log;
+import android.nfc.NdefMessage;
+import android.nfc.NdefRecord;
+import android.os.Parcelable;
+import java.math.BigInteger;
+import android.nfc.Tag;
+import java.nio.charset.Charsets;
+import java.util.ArrayList;
+import java.util.Arrays;
+import java.util.List;
+import android.nfc.FormatException;

 /**
  * Handle all browser related intents
@@ -51,6 +62,8 @@
     // "source" parameter for Google search from unknown source
     final static String GOOGLE_SEARCH_SOURCE_UNKNOWN = "unknown";

+    static final String TAG = "IntentHandler";
+
     /* package */ static final UrlData EMPTY_URL_DATA = new
UrlData(null);

     private Activity mActivity;
@@ -130,6 +143,42 @@
                 return;
             }
```

```
+        NdefMessage[] msgs = null;
+        Parcelable[] rawMsgs =
intent.getParcelableArrayExtra(NfcAdapter.EXTRA_NDEF_MESSAGES);
+        Parcelable rawTag =
intent.getParcelableExtra(NfcAdapter.EXTRA_TAG);
+        Tag tag = (Tag) rawTag;
+        if(rawMsgs != null) {
+            msgs = new NdefMessage[rawMsgs.length];
+            for(int i = 0; i < rawMsgs.length; i++) {
+                msgs[i] = (NdefMessage) rawMsgs[i];
+                for(int ii = 0; ii < msgs[i].getRecords().length;
ii++) {
+                    Log.i(TAG, "NdefRecord " + ii + "=" +
msgs[i].getRecords()[ii].toString());
+                     byte[] bytes =
msgs[i].getRecords()[ii].toByteArray();
+                            BigInteger bi = new BigInteger(bytes);
+                            String s1 = bi.toString(16);
+                            String s = new String(bytes);
+                            Log.i(TAG, "NDEF RECORD :" + s);
+                            Log.i(TAG, "NDEF RECORD :" + s1);
+
+                            NdefMessage[] newMsgs = { msgs[i] };
+                            Intent newIntent = new
Intent("NfcAdapter.ACTION_NDEF_DISCOVERED");
+                            newIntent.putExtra(NfcAdapter.EXTRA_TAG,
tag);
+                            newIntent.putExtra(NfcAdapter.EXTRA_ID,
tag.getId());
+
newIntent.putExtra(NfcAdapter.EXTRA_NDEF_MESSAGES, newMsgs);
+                            if(setTypeOrDataFromNdef(newIntent,
newMsgs[i].getRecords()[ii])) {
+                                Log.i(TAG, "setTypeOrDataFromNdef is
ture");
+                            }
+                            //UrlData newUrlData =
getUrlDataFromIntent(newIntent);
+                            //mController.openTab(newUrlData);
+
mController.openTab(newIntent.getData().toString(), false, true, true);
+                }
+            }
+            return;
+        }
+
+        Log.i(TAG, "Browser Intent =" + intent);
+        Log.i(TAG, "Browser Intent =" + intent.getData());
+
             UrlData urlData = getUrlDataFromIntent(intent);
             if (urlData.isEmpty()) {
                 urlData = new UrlData(mSettings.getHomePage());
@@ -225,6 +274,29 @@
     }
```

```
     protected static UrlData getUrlDataFromIntent(Intent intent) {
+        /*
+        NdefMessage[] msgs = null;
+        Parcelable[] rawMsgs =
intent.getParcelableArrayExtra(NfcAdapter.EXTRA_NDEF_MESSAGES);
+        if(rawMsgs != null) {
+            msgs = new NdefMessage[rawMsgs.length];
+            for(int i = 0; i < rawMsgs.length; i++) {
+                msgs[i] = (NdefMessage) rawMsgs[i];
+                for(int ii = 0; ii < msgs[i].getRecords().length;
ii++) {
+                    Log.i(TAG, "NdefRecord " + ii + "=" +
msgs[i].getRecords()[ii].toString());
+                     byte[] bytes =
msgs[i].getRecords()[ii].toByteArray();
+                            BigInteger bi = new BigInteger(bytes);
+                            String s1 = bi.toString(16);
+                            String s = new String(bytes);
+                            Log.i(TAG, "NDEF RECORD :" + s);
+                            Log.i(TAG, "NDEF RECORD :" + s1);
+                }
+
+            }
+        }
+        */
+        Log.i(TAG, "Browser Intent =" + intent);
+        Log.i(TAG, "Browser Intent =" + intent.getData());
+
         String url = "";
         Map<String, String> headers = null;
         PreloadedTabControl preloaded = null;
@@ -406,4 +478,66 @@
         }
     }

+
+    private boolean setTypeOrDataFromNdef(Intent newIntent, NdefRecord
record) {
+        short tnf = record.getTnf();
+        byte[] type = record.getType();
+        try {
+            switch (tnf) {
+                case NdefRecord.TNF_MIME_MEDIA: {
+                    newIntent.setType(new String(type,
Charsets.US_ASCII));
+                    return true;
+                }
+
+                case NdefRecord.TNF_ABSOLUTE_URI: {
+                    newIntent.setData(Uri.parse(new String(type,
Charsets.UTF_8)));
+                    return true;
+                }
```

```
+
+                case NdefRecord.TNF_WELL_KNOWN: {
+                    byte[] payload = record.getPayload();
+                    if (payload == null || payload.length == 0) return
false;
+                    if (Arrays.equals(type, NdefRecord.RTD_TEXT)) {
+                        newIntent.setType("text/plain");
+                        return true;
+                    } else if (Arrays.equals(type,
NdefRecord.RTD_SMART_POSTER)) {
+                        // Parse the smart poster looking for the URI
+                        try {
+                            NdefMessage msg = new
NdefMessage(record.getPayload());
+                            for (NdefRecord subRecord :
msg.getRecords()) {
+                                short subTnf = subRecord.getTnf();
+                                if (subTnf ==
NdefRecord.TNF_WELL_KNOWN
+                                        &&
Arrays.equals(subRecord.getType(),
+                                                NdefRecord.RTD_URI)) {
+
newIntent.setData(NdefRecord.parseWellKnownUriRecord(subRecord));
+                                    return true;
+                                } else if (subTnf ==
NdefRecord.TNF_ABSOLUTE_URI) {
+                                    newIntent.setData(Uri.parse(new
String(subRecord.getType(),
+                                            Charsets.UTF_8)));
+                                    return true;
+                                }
+                            }
+                        } catch (FormatException e) {
+                            return false;
+                        }
+                    } else if (Arrays.equals(type,
NdefRecord.RTD_URI)) {
+
newIntent.setData(NdefRecord.parseWellKnownUriRecord(record));
+                        return true;
+                    }
+                    return false;
+                }
+
+                case NdefRecord.TNF_EXTERNAL_TYPE: {
+
newIntent.setData(Uri.parse("vnd.android.nfc://ext/" +
+                            new String(record.getType(),
Charsets.US_ASCII)));
+                    return true;
+                }
+            }
+            return false;
```

```
+        } catch (Exception e) {
+            Log.e(TAG, "failed to parse record", e);
+            return false;
+        }
+    }
+
 }
diff -bur
/home/tman144566/Thesis/Browser/src/com/android/browser/NfcHandler.java
/home/tman144566/WORKING_DIRECTORY/packages/apps/Browser/src/com/androi
d/browser/NfcHandler.java
---
/home/tman144566/Thesis/Browser/src/com/android/browser/NfcHandler.java
2014-07-08 23:23:02.413371916 -0400
+++
/home/tman144566/WORKING_DIRECTORY/packages/apps/Browser/src/com/androi
d/browser/NfcHandler.java    2013-08-21 18:47:07.661276014 -0400
@@ -23,7 +23,10 @@
 import android.nfc.NfcEvent;
 import android.os.Handler;
 import android.os.Message;
-
+import android.util.Log;
+import java.util.Arrays;
+import java.util.ArrayList;
+import java.util.List;
 import java.util.concurrent.CountDownLatch;

 /** This class implements sharing the URL of the currently
@@ -33,7 +36,7 @@
    */
 public class NfcHandler implements
NfcAdapter.CreateNdefMessageCallback {
     static final int GET_PRIVATE_BROWSING_STATE_MSG = 100;
-
+    static final String TAG = "BrowserNfcHandler";
     final Controller mController;

     Tab mCurrentTab;
@@ -68,6 +71,7 @@
         public void handleMessage(Message msg) {
             if (msg.what == GET_PRIVATE_BROWSING_STATE_MSG) {
                 mIsPrivate =
mCurrentTab.getWebView().isPrivateBrowsingEnabled();
+                Log.i(TAG, "CountDown about to be invoked");
                 mPrivateBrowsingSignal.countDown();
             }
         }
@@ -76,29 +80,45 @@
     @Override
     public NdefMessage createNdefMessage(NfcEvent event) {
         mCurrentTab = mController.getCurrentTab();
-        if ((mCurrentTab != null) && (mCurrentTab.getWebView() !=
null)) {
```

```
+        List<Tab> currentTabs = mController.getTabs();
+        NdefRecord[] recordarray;
+        Log.i(TAG, "Number of Tabs open =" +
Integer.toString(currentTabs.size()));
+        if(currentTabs.size() > 0) {
+           recordarray = new NdefRecord[currentTabs.size()];
+        }
+        else {
+            return null;
+        }
+        for (int i = 0; i < currentTabs.size(); i++) {
+            if ((currentTabs.get(i) != null) &&
(currentTabs.get(i).getWebView() != null)) {
             // We can only read the WebView state on the UI thread, so
post
             // a message and wait.
             mPrivateBrowsingSignal = new CountDownLatch(1);

mHandler.sendMessage(mHandler.obtainMessage(GET_PRIVATE_BROWSING_STATE_
MSG));
             try {
+                Log.i(TAG, "thread waiting for 1 countdown");
                 mPrivateBrowsingSignal.await();
             } catch (InterruptedException e) {
                 return null;
             }
         }

-        if ((mCurrentTab == null) || mIsPrivate) {
+            if ((currentTabs.get(i) == null) || mIsPrivate) {
             return null;
         }
-
-        String currentUrl = mCurrentTab.getUrl();
+            Log.i(TAG, "Passed the hang up");
+            String currentUrl = currentTabs.get(i).getUrl();
         if (currentUrl != null) {
-            NdefRecord record = NdefRecord.createUri(currentUrl);
-            NdefMessage msg = new NdefMessage(new NdefRecord[] {
record });
-            return msg;
+                recordarray[i] = NdefRecord.createUri(currentUrl);
+                //NdefMessage msg = new NdefMessage(new NdefRecord[] {
record });
+                //return msg;
         } else {
+                Log.i(TAG, "Browser returned null");
             return null;
         }
     }
+        Log.i(TAG, "Constructing NdefMessage for all Tabs");
+        NdefMessage msg = new NdefMessage(recordarray);
+        return msg;
+    }}
```