Wireless Distributed Computing on the Android Platform

Kiran Karra

Thesis submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Master of Science
in
Electrical Engineering

T. Charles Clancy, Chair
Jeffrey H. Reed
C. Jules White

September 27, 2012
Falls Church, Virginia

Keywords: Wireless Distributed Computing, Android, Mesh Networks

Wireless Distributed Computing on the Android Platform

Kiran Karra

(ABSTRACT)

The last couple of years have seen an explosive growth in smartphone sales. Additionally, the computational power of modern smartphones has been increasing at a high rate. For example, the popular iPhone 4S has a 1 GHz processor with 512 MB of RAM [5]. Other popular smartphones such as the Samsung Galaxy Nexus S also have similar specifications. These smartphones are as powerful as desktop computers of the 2005 era, and the tight integration of many different hardware chipsets in these mobile devices makes for a unique mobile platform that can be exploited for capabilities other than traditional uses of a phone, such as talk and text [4].

In this work, the concept using smartphones that run the Android operating system for distributed computing over a wireless mesh network is explored. This is also known as wireless distributed computing (WDC). The complexities of WDC on mobile devices are different from traditional distributed computing because of, among other things, the unreliable wireless communications channel and the limited power available to each computing node. This thesis develops the theoretical foundations for WDC. A mathematical model representing the total amount of resources required to distribute a task with WDC is developed. It is shown that given a task that is distributable, under certain conditions, there exists a theoretical minimum amount of resources that can be used in order to perform a task using WDC. Finally, the WDC architecture is developed, an Android App implementation of the WDC architecture is tested, and it is shown in a practical application that using WDC to perform a task provides a performance increase over processing the job locally on the Android OS.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Overview

The explosive growth of smartphones over the past couple of years has been unprecedented. Almost half the phones now sold to consumers are smartphones [17]. Additionally, these smartphones are becoming increasingly more powerful. The highlighted features of the most recent iPhone 4S is a 1 GHz dual-core processor, 512 MB of RAM, and integrated WiFi and Bluetooth chipsets [5]. Android smartphones such as the Samsung Galaxy S III have similar specifications, boasting a quad-core 1.4 GHz processor, 1 GB of RAM, and integrated WiFi and Bluetooth chipsets [6]. Although commercial consumers are generally attracted to the latest phone for uses such as 1080p video recording, being able to instantly update their status on social media networks, or playing graphics intensive video games to pass time, the capabilities of a smartphone can be exploited for more "useful" purposes in the military arena. For example, with existing on-board sensors such as the high resolution camera, smartphones can be used by military personnel to collect imagery intelligence (IMINT). In a clandestine scenario, the microphone on the smartphone can be used record voice conversations which can later be post-processed for mining of intelligence data. From a cyber perspective, the on-board WiFi chipsets of many smartphones can be used to potentially

hack into WiFi hotspots for intelligence gathering.

Typically, processing of intelligence data has to be done post-collection because it is computationally intensive. Raw intelligence data is generally processed at server farms days or weeks after collection of the intelligence data. If multiple devices available to operators in the field, such as smartphones, were linked together to perform cooperative processing however, analysis of raw intelligence data could be performed in a more real-time basis in the field to potentially provide actionable intelligence in a more timely manner. The idea is that multiple smartphones can be linked together to perform processing that provides useful intelligence data to the operator in the field.

Although smartphones offer the advantage of being able to potentially provide real-time intelligence through distributed processing, there are many challenges associated with using smartphones as a distributed processing platform. One of the biggest challenges to overcome is that the devices are generally very tightly integrated. On one hand, the tight integration of hardware on smartphones is a good thing in that it is designed specifically for that particular mobile device and as such is optimized for it. For example, the choices of battery, processor, memory, and communications chipsets by device manufacturers are carefully made such that the user experience is maximized. The downside to this is that although software can be written to do almost anything, the underlying hardware chosen by the device manufacturer may not support that feature. For example, software that could run on any platform (including smartphones) could be written to capture packets from the WiFi chipset, and traffic analysis can be performed. However, if the underlying WiFi chipset does not support packet capturing, the smartphone inherently cannot have this capability. This is different than a general purpose computer because hardware is not as tightly integrated and can generally either be swapped out or new modules can be plugged in.

In order to link smartphones together for computing, they must be networked in some fashion. Generally, smartphones do not have wired network capability, and thus must be linked together in a wireless network using hardware that is built into the device. The wireless

network presents many problems and adds to the already complex distributed computing problem. Traditionally, distributed computing is thought about in the context that nodes interconnected for processing have a reliable communications channel between them. However, this assumption is generally not valid for a wireless connection. Additionally, the nature of smartphones is that they have a limited amount of energy available for processing. Wireless communication between nodes in a limited energy environment does not lend itself to alleviating that problem. Therefore, wireless distributed computing must perform cross-layer optimization in both the total processing time and total energy used domains in order to make wireless distributed computing a viable option [7].

The problems described above lend themselves to many research problems. In wireless distributed computing (WDC), efficient protocols for creating mesh networks and inter-node communication is a research problem that has direct consequences to WDC [9]. However, this generally involves using customized hardware, and this research problem does not lend itself to WDC on mobile devices because the hardware is pre-defined by the manufacturer. At a higher layer, research about how to store data among a distributed set of nodes for different criterion such as protecting data integrity, ensuring information availability, or data protection is an important topic that would have many consequences to WDC applications. Finally, research about how to optimize a task such that resources are minimized is another important problem that needs to be tackled if WDC is to become a viable option for computing [20].

1.2 Prior Work

Significant research into Wireless Distributed Computing has been conducted. Much of the surveyed research focuses on custom processors and network protocols. For example, the published article "Wireless Distributed Computing: A Survey of Research Challenges" discusses that in a WDC environment, flexible network protocol design is required in order

to successfully enable cooperation among peer nodes to accomplish a task [9]. Research has gone into the possibility of using COTS components to perform distributed computing. For example, the paper "Distributed Network Computing over Wireless Links" discusses the potential of performing distributed computing using wireless LANs. However, the premise of their research was based on using wireless bridges, with wired clusters communicating with each other using bridges [21]. Although this research was performed with COTS components, it is not a truly wireless solution.

In this thesis, the concept of WDC using existing network protocols and hardware with Android enabled devices such as smartphones is explored. The envisioned usage of this research is cooperative smartphones interconnected via a mesh network to perform distributed processing. This scenario is useful in a military situation, where all soldiers in a patrol may have smartphones that are connected via a mesh network to perform processing. It is important to note here that all nodes in the distributed mesh network for processing are cooperatively participating in the network, and consequently running the same WDC control software. Current WDC research focuses on implementing new physical layer communication subsystems, as well as network protocols to perform WDC [9]. However, much of the surveyed literature does not address how existing protocols and communications systems could be leveraged to perform distributed processing. This thesis addresses these issues.

With these concept of operations (CONOPs) in mind, a prototype WDC architecture and its implementation in the Android OS was developed. Specifically, existing open-source Android software was integrated to create an Ad-Hoc WiFi network of four Android smartphones. Then, custom WDC control software, existing in the life cycle of Android as an App, was developed and deployed on these smartphones. Finally, simulations to determine the performance of WDC over local processing were run. In addition the the experimental testing and results, the theoretical foundations for optimizing the distribution of a task in a WDC mesh network based upon the state of nodes in the wireless network was developed. The simulations showed that the performance of WDC is greater than local processing, with respect to the domain of time required to process a certain task.

The first revision of the WDC software does not consider security into the system model. The implementation of the WDC system assumes a non-hostile environment. There is no security implemented in the WDC system, either at the IP layer with traditional security measures such as WEP or WPA, or at the WDC message authentication level. Currently, this means that the WDC system assumes there is nobody in the IP mesh network that is trying to "break" the system, by sending unintentional messages to the system.

Chapter 2

Software Architecture

2.1 Introduction

In order to enable wireless distributed computing, a management engine (implemented in software, hardware, or a combination of both) needs to exist. The WDC management engine is responsible for managing the distribution of tasks, the communication between nodes, and among other things the load balancing of work among available processing nodes in order to optimize the amount of time it takes to complete a task and the energy expended in completing the task. The goal of this thesis is to demonstrate that wireless distributed computing is possible on an Android platform. In order to fuse the concepts of distributed computing and Android, this chapter explains the WDC software architecture, which is the management engine implemented in software. It then provides an overview of the Android system. Finally, the Android implementation of the WDC software architecture is discussed.

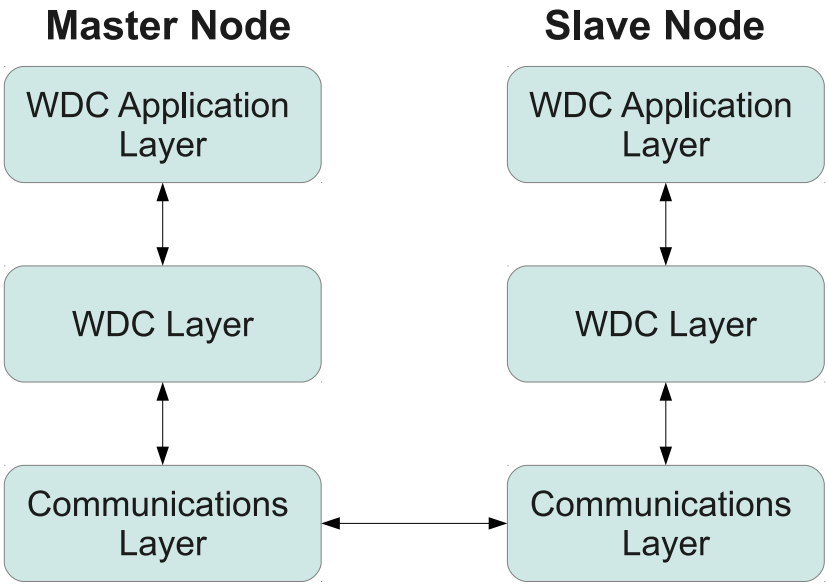

Figure 2.1: WDC Architecture - Overview [20]

2.2 WDC Architecture

The WDC software architecture is a design model for how a general wireless distributed computing system should work. The software architecture is designed with the goals of being a reusable and flexible architecture for future improvements, as well as allowing for a clean implementation. In order to meet these design goals, the WDC software architecture is based upon a layered software model similar to the OSI networking model. Each layer in the WDC system provides services to the layer immediately above it, and as such is restricted to communicate with neighboring layers only [20]. Figure 2.1 shows the high level WDC architecture.

Figure 2.1 outlines the three major layers in the WDC software architecture: 1.) the WDC application layer, 2.) the WDC layer, and 3.) the Communications layer. The WDC application layer contains the applications that are to be executed over the distributed wireless network. The applications generally consist of a complex computational task (such as facial recognition or 3D environment reconstruction) fragmented into atomic work units. An atomic work unit is defined as the minimum amount of work that can be given to a certain processing node and is relative to the application. For example, a facial recognition application may define an atomic work unit to be recognizing one face, while a 3D environment reconstruction application may define an atomic work unit to be reconstructing one of the three dimensions. The WDC Application layer performs application-dependent functions required for WDC. Fragmenting the application computational task into atomic work units is one such example.

The WDC layer is the software piece that deals with managing the entire WDC system. The main job of the WDC layer is to optimize how the task to be completed by the wireless application should be distributed such that total energy and processing time are minimized. The WDC layer makes some high-level decisions, including whether or not to use local onboard processing or distributed processing for a given WDC application. The WDC layer manages and controls various aspects of distributed computing on the wireless network. It performs application-independent functions that are relevant for distributed computing, such as providing WDC metrics to the optimizer for making decisions between local and distributed processing, optimizing WDC resource allocation, and controlling and managing the distributed computation processes [20]. Finally, as Figure 2.1 shows, the WDC layer provides an API to the WDC application layer to allow the wireless applications the ability to use the WDC system. The WDC layer can be considered as the "brains" of the WDC system, since it manages the distribution of tasks, execution of the task, and accumulation of results.

The communications layer takes care of reliable communications between the wirelessly distributed nodes. The communications layer provides an API to the WDC layer in order

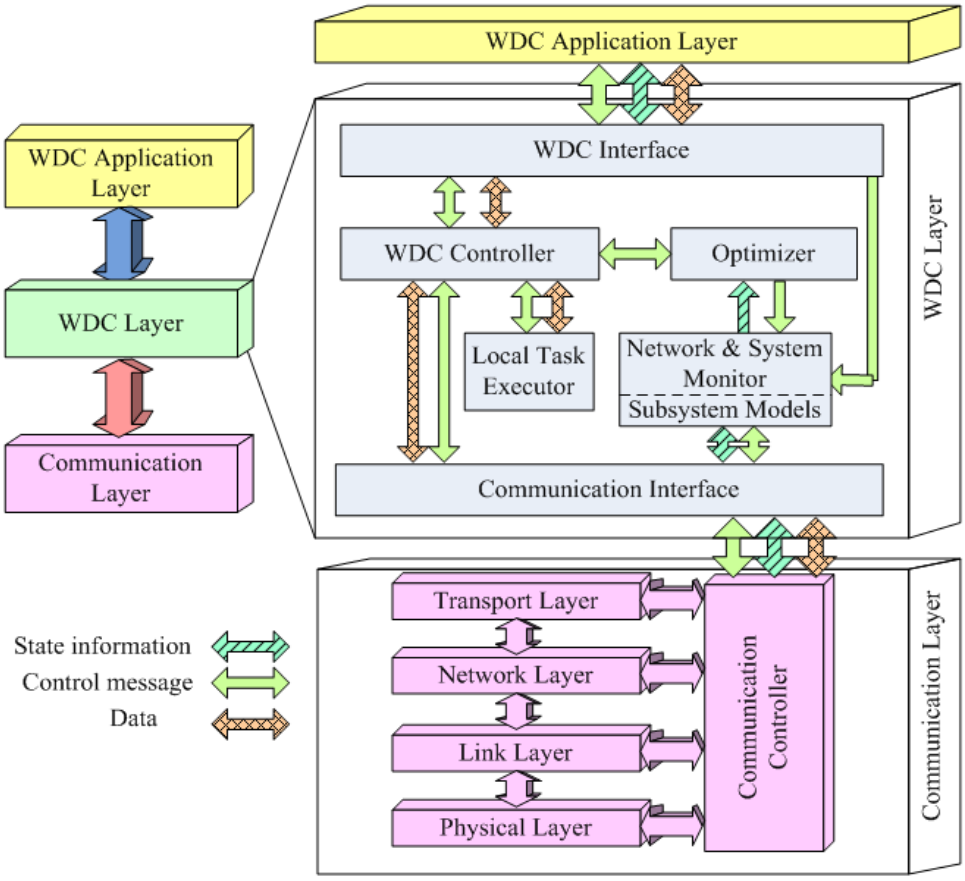

Figure 2.2: WDC Architecture - Detailed [20]

to allow the WDC system to seamlessly communicate with any other node in the system, regardless of whether the low level communications mechanism is a TCP/IP based WiFi network or a proprietary communications protocol.

It is important to note that there is no distinction between master and slave nodes in the WDC architecture, as shown by Figure 2.1. The node starting the wireless application is automatically considered the master node while nodes assisting in the processing are slave nodes. Any node can be a master or slave node in the WDC architecture described, and all nodes must run the same software that enables WDC.

The detailed WDC architecture is shown in Figure 2.2 [20]. As shown by Figure 2.2, the

WDC layer is made up of the WDC Interface, the WDC controller, Local Task Executor, Optimizer, Network and System Monitor, and the Communications interface. At a high level, these components work together to provide WDC functionality to the WDC application.

The WDC Interface is, as shown in Figure 2.2, the interface to the WDC application layer. It provides function calls that the wireless application (such as the distributed facial recognition) can use. The function calls specified by the WDC software architecture for the WDC interface are listed in Table 2.1.

Table 2.1: WDC Interface - functions

Function	**Description**
queueJob(...)	Queue a job to be executed
cancelJob(...)	Cancels a queued job
changeJobPriority(...)	Changes the priority of the queued job
requestJobStatus(...)	Requests the job status
requestJobResults(...)	Retrieves the results of the queued job

The API specified by the WDC interface is used by the wireless application to manage a task. For example, a wireless application would start a task by calling the *queueJob()* function. It would receive the job results by calling the *requestJobResults()* function.

The WDC controller can be considered the nerve center of the WDC layer software. On the master node, the WDC controller coordinates the entire distribution of the job. Once a job is queued by the user to the WDC Interface through the *queueJob()* function, the WDC controller pulls in that specific job request. The WDC controller then sends a request to the Optimizer block to determine how best to distribute the job. The optimizer's response is then used by the WDC controller to allocate atomic work units to each slave node through the Communications interface. For a slave node, the WDC controller's job is to listen for task execution requests from the Communications interface. If a task execution request

is received on a slave node, the WDC controller will pass the atomic work units to the Local Task Executor for algorithm execution. After the Local Task Executor has completed processing the task, the WDC controller on the slave node will push the results back to the node from which the request came.

The job of the Local Task Executor (LTE) is to run the user-specific algorithm processing software with the work assigned to that specific node. After the atomic work units have been processed by the node, the results are pushed back to the node which assigned the slave node that specific task.

The goal of the Optimizer is to determine the optimal distribution of the job such that both total energy required to complete the task and the total time required to process the task are minimized. In order to perform its function, the Optimizer communicates with the Network and State monitor (NSM) to determine the the state of every node available for processing in the WDC network. The theory and the algorithm behind the optimizer is explained in detail in Chapter 3. In short however, the optimizer uses the state information of all the available nodes to determine how best to distribute the atomic work units among the available processing nodes. For example, suppose there are 100 atomic work units to be processed and five available processing nodes. The optimizer will use the network state monitor to get state information about each available processing nodes. Based on the battery life available to each node, the current processing load that the node is under, the number of network hops that a node is away from the master node, and an optimization criterion, the optimizer may decide to use three of the five available processing nodes with an even distribution among the nodes, or use all five nodes with an uneven job distribution among all nodes.

As stated above, the job of the Network and State Monitor (NSM) is to provide state information to the optimizer when requested. State information includes percentage battery life remaining, current processing load, and the communications latency between the node and the master node.

Finally, the communications interface is a generalized API that allows the WDC controller to communicate with the other nodes. The generalized API allows for a standard interface to be defined between the WDC controller and the communications interface, regardless of the actual wireless protocol or technology that is used for communication between wireless nodes, that allows nodes to communicate with each other.

2.3 Android

The WDC software architecture described in Section 2.2 above is generalized such that it can be implemented on any architecture and operating system, because the WDC software architecture essentially defines a generalized API rather than how it should be implemented. This section describes the Android operating system and the components that make up an Android application, such as a wireless application that uses the WDC service or the WDC service itself.

The Android operating system is a Linux based operating system built for mobile devices by the Open Handset Alliance. It contains many components including a Linux based kernel, middleware, and core applications required for basic mobile device functionality.

Figure 2.3 shows the Android architecture in detail [11]. It highlights the major components that make up Android. As can be seen from Figure 2.3, the major Android components are the Applications, Application Framework, Libraries, Android Runtime, and the modified Linux kernel. These components work together in the form of applications, or "Apps", in order to provide mobile device functionality to the user. The Linux based Android kernel provides core system services to the Android OS including security, memory management, process management, hardware access, and network connectivity. The kernel essentially acts as an abstraction layer between the hardware on-board the mobile device and the rest of the software stack. Kernel and middleware code (software libraries, application framework code) is generally written in the C programming language. Applications for Android are

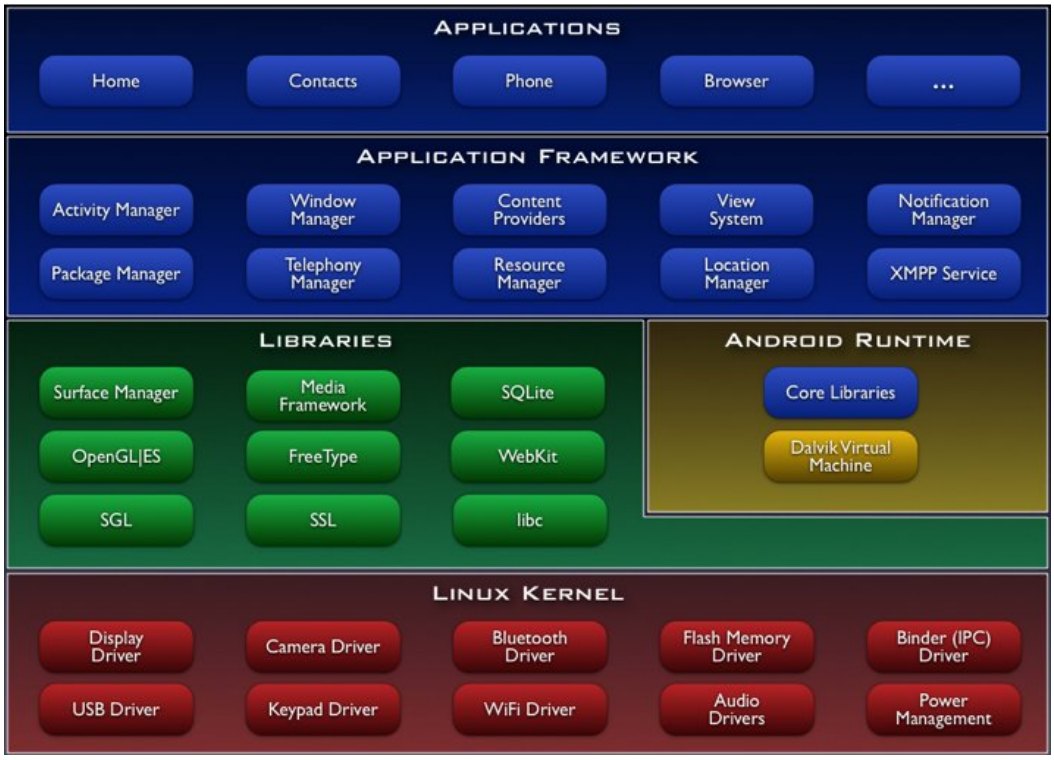

Figure 2.3: Android OS Architecture [11]

generally written in the Java programming language and work with the kernel through the Application Framework and Libraries to provide functionality to the user [11]. The Android runtime pulls it all together and allows the application code to be executed in a real-time fashion to provide a user experience.

2.3.1 Android Applications

An Android application, or "App", is the user interface to the mobile device which performs a particular function. A user's primary interaction with the mobile devices is through the App. All functionality in Android is provided through Apps, including core functionality such as phone or text message capability. Android Apps consist of four types of software components, 1.) Activities, 2.) Services, 3.) Content Providers, and 4.) Broadcast Receivers. An App is a combination of one or more of these application components.

2.3.2 Activities

An Android activity is an application component that provides a user interface on the screen through which the user can interact with the application. An activity represents a single focused thing that the user can do with an App. Examples of Android activities include the phone dialer screen or the MMS (multimedia messaging service) screens. As the definition states, these activities are part of the phone application and the MMS application and let the user perform one function, either dialing a phone number or sending a text message respectively [11].

2.3.3 Services

An Android service is an application component that generally performs operations in the background without providing a user interface. Examples of services are the phone service

and the MMS service. These are background running operations that wait for phone calls or text messages. As stated above, Apps are a combination of many application components, and the services described above are part of the phone and MMS applications respectively. It is important to note that even though the function of Android services is to perform computationally intensive tasks in the background, they run in the same execution thread as the main UI thread which started the service.

Android defines multiple kinds of services, 1.) an Intent Service, 2.) a bound service, or 3.) just a regular service. Intent services and bound services are still subclasses of the original Service class, but they overload different functions to provide slightly different functionality. An intent service is a service, that when started, will spawn a new thread to perform a task. After the task is completed, the intent service will automatically shut down. A bound service is one that is started by a user (by broadcasting an Intent) or another application component which binds to that service. The bound service is then available for performing operations (through function calls, similar to remote procedure calls) and stays alive as long as another application component is bound to it. Therefore, it differs from the Intent service in that it is not a one-shot service that provides a background function, but stays alive and has the ability to perform repeated calculations upon request by the application component which is bound to that specific service [11].

2.3.4 Content Providers

Android content providers are the standard interface that connects data running in one process with code running in another process. They essentially allow for multiple applications to share data. For example, a content provider may be used by the phone application to communicate the phone number of a contact to a business card application looking to aggregate phone book information into the electronic business cards [11].

2.3.5 Broadcast Receivers

Broadcast receivers are components configured to receive data from other Android components. They can be used to receive asynchronous system wide notifications such as battery level or connectivity to the cellular or WiFi network. Broadcast receivers can also be used internally within an application to communicate asynchronously between services and other application components [11].

2.3.6 Intents

Activities, Services, and Broadcast Receivers are all activated through messages called intents. An Intent is a passive data structure holding an abstract description of an action to be performed. The description is usually defined by the Intent action property and the Intent category property. The Intent also has the ability to hold data required for the components it is activating, known as Intent extras. When an Intent is broadcast, the Android OS determines which components or applications are configured to be activated upon receiving that Intent (known as an Intent filter). If an application or component is found, it is started. If it is not, the Intent gets lost [11].

An example of how an Intent would be used in an application is as follows. Suppose that App1 is configured to start App2 if a certain button is clicked. If the user clicks that button, then App1 will fire an Intent that App2 is configured to filter on. When App1 fires that Intent, the OS receives it and starts App2. The Intent can contain data that App1 may want to pass to App2.

2.4 Android Implementation of WDC Architecture

Figure 2.4 shows the design of the Android WDC application, which is an Android implementation of the WDC layer in Figure 2.2. The direction of the arrows indicates how data

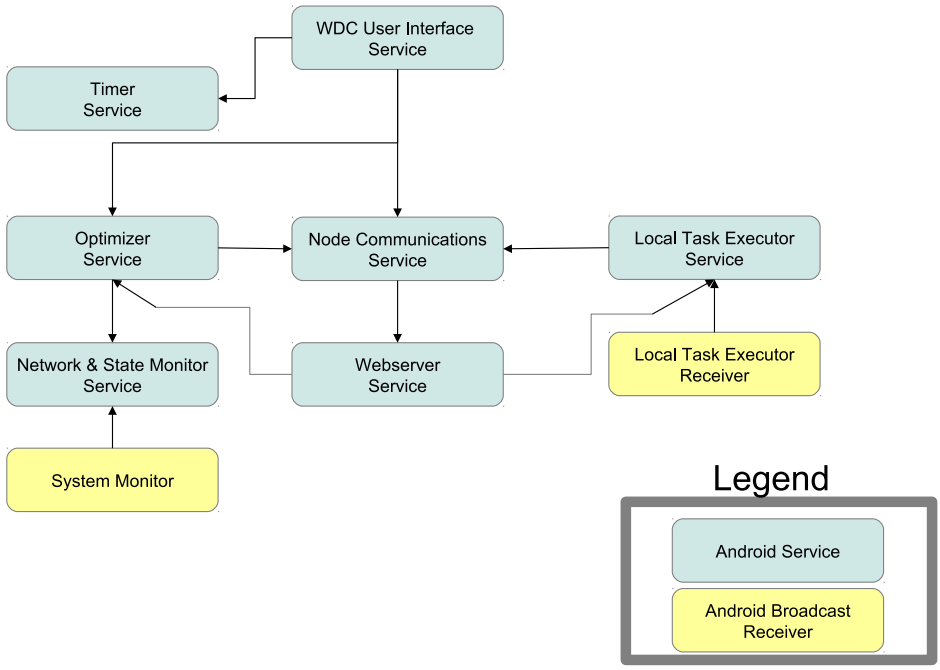

Figure 2.4: Android implementation of WDC Architecture

flows between the application components. As Figure 2.4 shows, the WDC application consists of multiple bound services and broadcast receivers. For testing purposes, the WDC application also contains an Android activity which allows for easy testing, but this is not shown in Figure 2.4 because the Android activity does not contribute towards the WDC application's core functionality.

2.4.1 Overview

Figure 2.5 shows the program flow of how the Android WDC application works at a high level and how the major components in the WDC Android implementation work together to complete a task. The first step in executing a wireless task is for the wireless application to submit a task to the WDC layer. The WDC User Interface then requests the Optimizer to determine the optimal distribution of the task, as shown in Step 1. The Optimizer sends messages to all of the slave nodes (only one shown in Figure 2.5) for state information,

as shown in Step 2. The slave node receives the request for state information and that request gets routed to the optimizer. The optimizer returns the state information back the communications hub, which sends out the state information back to the master node. After the master node has accumulated the state information from all slave nodes, the optimizer performs optimization and sends the job distribution information to the WDC User Interface Service. The WDC User Interface Service then sends the job information to the slave nodes for processing through the Webserver to the Local Task Executor (LTE). The LTE performs the task, and returns the results back to the master node. Although it is not shown in Figure 2.5 for the sake of clarity, all inter-node communication occurs via the Webserver service.

2.4.2 WDC User Interface Service

The WDC User Interface Service encapsulates the WDC Interface and the WDC controller components shown in Figure 2.2. It is a bound service (for testing purposes bound to the WDC test Activity which is the user interface) which waits for intents to be fired to kick off a WDC job. The WDC User Interface Service also binds the Optimizer service and the Node Communications Service to itself so that it can determine how to optimally distribute the job, as well as communicate instructions to all the slave nodes.

A wireless application can start a wirelessly distributed job by broadcasting an intent with the action property **wdc.queue.job** and the category property set to **android.intent.category.DEFAULT**. When a wireless application broadcasts this intent, it is expected to have at least one Intent extra with the key **intent_to_fire**. This extra is a string data type which tells the local task executor the intent action to fire to start the local processing. Based on the user application however, more Intent extras can be packaged into the Intent for the specific user application. These intent extras are passed through the system to the destination node for application specific processing. When an Intent with the action property set to **wdc.queue.job** and the category property set to **android.intent.category.DEFAULT** is broadcast, the WDC User Interface Service receives this intent. The WDC user interface

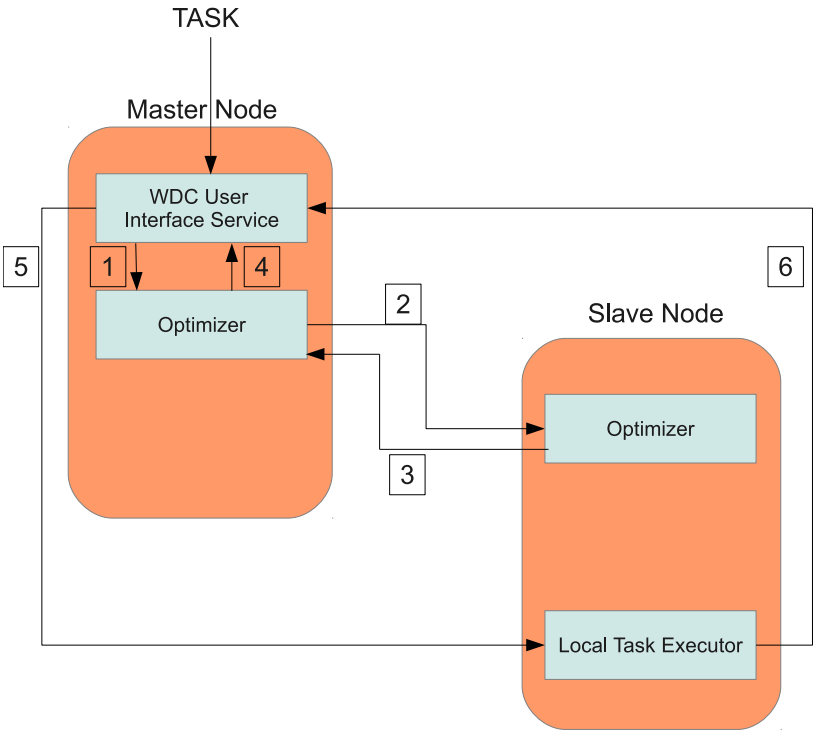

Figure 2.5: High level program flow of Android WDC Application

1.) WDC User Interface requests state information of all slave nodes

2.) Optimizer on Master node requests state information from all slave nodes

3.) Slave nodes respond with state information

4.) State information processed by optimizer and job distribution is allocated to each node, and passed to WDC Interface

5.) WDC Interface sends job requests to slave nodes

6.) Slave nodes respond with processing results

gets all the Intent extras (data associated with the Intent) and passes that information to the Optimizer service through the function call *optimize()*. The Optimizer service runs an optimization algorithm after receiving state information from all available processing nodes (more about this later) on the provided parameters and creates commands for each of the slave nodes. The WDC User Interface Service then sends these commands to each of the respective nodes via the Node Communications Service. The WDC User Interface Service is also bound to a Timer service, as shown by Figure 2.4. When a job is submitted to the WDC User Interface Service by broadcasting an Intent as specified above, the Timer service simply records a start time. When all the results have been aggregated by the WDC User Interface Service after a job is complete, the Timer service records the stop time, and the elapsed time of the job can be measured.

It should be noted that broadcasting an Intent with property **wdc.queue.job** and category **android.intent.category.DEFAULT** is equivalent to calling the *queueJob()* function described in Table 2.1. This is the only API function call for the WDC User Interface which is currently implemented in the Android WDC App.

2.4.3 Node Communications Service

The job of the Node Communications Service is to handle overall communications between nodes in the WDC mesh network. It is bound by the WDC User Interface Service, LTE service, and Optimizer service. It provides two functions that can be called by these binding services. These functions and their uses are defined in Table 2.2.

The Optimizer Service, LTE Service, and the WDC User Interface Service all use the *sendHTTPPostCommand()* function to send HTTP POST commands to other nodes. The Optimizer Service uses the *sendHTTPPostCommand()* function to send StateRequest messages (applicable to the master node) and respond with StateResponse messages (applicable to slave nodes). The LTE service uses the *sendHTTPPostCommand()* function to send results back to the master node when it receives data from the local task/service that was

Table 2.2: Node Communications Service Functions

Function	**Description**
sendHTTPPostCommand(...)	Sends a HTTP POST XML Command to a specified IP
getWebserverService(...)	Returns handle to Webserver Service

executed as part of the wireless application. Finally, the WDC User Interface service uses *sendHTTPPostCommand()* function to post JobRequest commands to all the available processing nodes. The *getWebserverService()* function is used as a maintenance function by the WDC User Interface Service to detect when a job is complete.

2.4.4 Webserver Service

The Webserver service can be considered part of the communications interface, with reference to Figure 2.2. The job of the Webserver service is to host a webserver that accepts HTTP POST and HTTP GET commands. As stated in the overview section, the Android implementation of WDC restricts all communication between nodes to use the HTTP protocol. This requires that a webserver with the ability to accept HTTP commands exist on every device running the WDC software. One of the functions of the Webserver service is to operate the webserver which can handle HTTP POST commands. When a HTTP POST command is received, the webserver parses the XML command. Based on the XML command, the Webserver service will pass the data to either the Local Task Executor service (if the XML command is a tasking command) or the Optimizer Service (if the XML command is a state request). As seen by figure 2.2, the Webserver service binds both the Local Task Executor Service and the Optimizer Service.

The WDC Android implementation allows for a restricted set of XML commands to be transmitted between nodes. Although these XML commands are extensible to allow flexibility

for the user to run specialized applications, the XML parent message type must be in one of the allowed XML commands. The allowed XML commands are described in Listings 2.1, 2.2, 2.3, and 2.4.

```
<xml>
    <message>
        <type>StateRequest</type>
        <nodeIP> the ip of the requestor </nodeIP>
        <requestID> a request id </requestID>
    </message>
</xml>
```

Listing 2.1: StateRequest XML Message

```
<xml>
    <message>
        <type>StateResponse</type>
        <nodeIP>the IP of the node responding</nodeIP>
        <responseID>the same as the request id</responseID>
        <procLoad> the processor load value </procLoad>
        <batteryLevel> the battery level </batteryLevel>
    </message>
</xml>
```

Listing 2.2: StateResponse XML Message

```
<xml>
    <message>
        <type>JobRequest</type>
        <nodeIP> the ip of the requestor </nodeIP>
        <jobID> a job id that is assigned </jobID>
        <intentToFire> the android intent to fire </intentToFire>
        <fileURL> the URL of the file that needs to be got. This one file
            will contain the URLs of all the files that we need to pull </
            fileURL>
```

```
            ** any other XML tags that can be interpreted by the specific end
               service that the end service needs in order to perform the
               task
    </message>
</xml>
```

Listing 2.3: JobRequest XML Message

```
<xml>
    <message>
        <type>JobResponse</type>
        <nodeIP> the ip of the node </nodeIP>
        <jobID> the job id that this response to </jobID>
        <fileURL> the URL of the file that needs to be got.  This one file
            will contain all the results. </fileURL>
            ** any other XML tags that the requestor of the job needs to fully
                understand the results
    </message>
</xml>
```

Listing 2.4: JobResponse XML Message

Listing 2.1 shows the XML message that is sent by a node (presumably master node) to request the state of another node. This message is generated internally by the Optimizer service of the master node. When the webserver receives this message, it passes it to the Optimizer service through the function call *processStateRequestMessage()*. The response to a StateRequestMessage is the StateResponseMessage, defined in Listing 2.2. The response message contains information about the device being queried. More specifically, the StateResponseMessage contains information about the processor load and the battery level. This information is used by the Optimizer service to determine the optimal job distribution. When a StateResponseMessage is received, the webserver passes the message to the Optimizer service through the function call *processStateResponseMessage()*. Currently, the network hop count/latency is not yet implemented in the WDC Android App.

```
http://<ipAddressOfServer>:<port>/fileXfer?filename=file1.txt
http://<ipAddressOfServer>:<port>/fileXfer?filename=file2.txt
http://<ipAddressOfServer>:<port>/fileXfer?filename=file3.txt
http://<ipAddressOfServer>:<port>/fileXfer?filename=file4.txt
http://<ipAddressOfServer>:<port>/fileXfer?filename=file5.txt
```

Figure 2.6: Contents of a top-level file described by the XML tag **fileURL**

Listing 2.3 shows the XML message that is sent by the master node to command a slave node to perform some work. The **intentToFire** XML tag tells the slave node the Intent action property that it must set to start the service that will process the data. The **fileURL** XML tag tells the slave node the URL of the file which contains the URLs of all the required files for processing. The contents of a sample file described by the fileURL is shown in Figure 2.6. As Figure 2.6 shows, the file contains URLs for the five files that needs to be gotten by the slave node in order to perform processing. The first thing that the slave node does when it receives a JobRequest message is to download all files from the URL's in the file specified by the **fileURL** XML tag.

When a JobRequestMessage is received, the webserver passes the message to the LocalTaskExecutor through the function call *queueLocalJob()*. As noted by Listing 2.3, the WDC system allows for additional XML tags to be specified that allow the user to provide application specific information that is not needed to manage the WDC task. After a node finishes processing a task, it responds to the master node with a JobResponseMessage. Listing 2.4 defines the JobResponseMessage. Similar to the JobRequestMessage, the **fileURL** XML tag tells the master node the URL of the file which contains the results of the processed job. Again, as noted by Listing 2.4, the WDC system allows for additional XML tags to be specified to convey application specific information that is not needed by WDC to manage the task.

2.4.5 Optimizer Service

The Optimizer service is a bound service which performs many of the core functions of the WDC layer system. Its main job is to take information from the WDC User Interface Service, and run an optimization algorithm which determines the optimal distribution of the jobs. The first step in running the optimization algorithm is to determine all available processing nodes, and determine the state of each node. The available processing nodes are determined by a properties file that is statically configured before the WDC system is started. The optimizer then sends the StateRequestMessage described by Listing 2.1 to each of the available nodes, by calling the function *generateStateRequestMessage()*. The Optimizer service then waits for each node to respond to the StateRequestMessage.

All nodes are expected to respond to the StateRequestMessage with a StateResponseMessage. The StateResponseMessages are parsed by the Webserver Service as stated above, and the information is passed from the Webserver to the optimizer. The optimizer combines the state information of all available nodes with an optimization metric to determine the optimal distribution of work among all available processing nodes. Using this information, the Optimizer service creates JobRequestMessage commands described by Listing 2.3 for each node using the command *generateJobRequestMessage()*. As stated above, the WDC User Interface Service then takes these commands and sends them to each of the processing nodes.

2.4.6 Network and State Monitor Service

The Network and State Monitor (NSM) Service is a bound service, to which the Optimizer Service binds to. The NSM service provides function calls which allow the Optimizer to determine the state of the local node. Table 2.3 shows the functions provided by the NSM service.

Table 2.3: Network and State Monitor Service Functions

Function	**Description**
getBatteryLevel()	Returns the battery level
getBatteryVoltage()	Returns the battery voltage
getBatteryTemp()	Returns the battery temperature
getProcLoad()	Returns the current processor load

2.4.7 System Monitor

The System Monitor is a Broadcast Receiver configured to receive system wide broadcasts of the battery level. Additionally, the System Monitor also reads the '/proc/stat' file every five seconds to determine the current processor load. The NSM service has handles to the System Monitor, and can thus access this information.

2.4.8 Local Task Executor Service

The Local Task Executor (LTE) Service is a bound service which executes services specified by the user. As specified by Figure 2.2, the LTE Service is bound by the Webserver Service. When the webserver receives a JobRequestMessage, it passes it to the LTE service. The LTE service then reads the **fileURL** tag specified by the JobRequestMessage, detailed in Listing 2.3. The LTE service then sends an HTTP GET command to retrieve the file specified by the **fileURL** tag. The retrieved file is then opened, and each URL specified inside the retrieved file is "gotten" by the LTE service. After all the files required for processing have been retrieved, the LTE service fires the intent specified by the **intentToFire** XML tag to start the application specific task. All files that are downloaded are in the base folder '/mnt/sdcard/data/wdc/webserver'. Additional folders are created relative to the file request path. For example, if the URL for getting a file is

http://192.168.2.254:8087/fileXfer?filename=faces/s1/s1.png, the file s1.png will appear in the folder '/mnt/sdcard/data/wdc/webserver/faces/s1/s1.png'.

2.4.9 Local Task Executor Receiver

The LTE Receiver is a Broadcast Receiver which listens for results from an application specific local task. When the local task completes processing, it broadcasts its results to the LTE receiver which then routes the results via HTTP POST to the master node.

2.4.10 Timer Service

The Timer service is a bound service which provides function calls to the WDC User Interface Service to start and stop a timer. The timer service is not an essential service to the WDC system, and exists mainly to benchmark the WDC system performance.

Chapter 3

Theory

3.1 Introduction

The purpose of a wireless distributed computing (WDC) system is to distribute a task that can be performed on one node to many wirelessly connected processing nodes. The idea is to optimize the distribution of processing the task such that both energy and processing time are minimized. The WDC optimizer must take into account variables related to the unreliable communications channel, and the limited amount of power available for each processing node in order to make WDC a more efficient approach to computing than local processing. The variables and their associated meanings considered in this thesis are outlined in Table 3.1.

In this chapter, the time and energy required to process a task using WDC is modeled. Requisite knowledge about mesh network topologies and mesh models are discussed. A continuous approximation of the mathematical models for time and energy required to complete a task is then developed. Finally, using the continuous approximations, the theoretical optimal number of nodes to use for processing a task using WDC is derived.

Table 3.1: WDC Variables

Variable	**Definition**
n	The total number of available processing nodes
m_i	The number of hops away that node i is from the master node
K	The total number of units of work to be performed by the WDC system
δ	The startup time of a process in the Android OS
E	The energy required to process one unit of work
ϵ	The energy required to start a process in the Android OS
t_x	The time it takes to transfer one unit of work to a node one hop away
t_o	The time required to establish a connection with a node one hop away
k_i	The number of units of work to be performed by node i
m_i	The number of hops required to communicate with node i
e_x	The energy required to transfer one unit of work to a node one hop away
e_o	The energy required to setup a connection with a node one hop away

3.2 Assumptions

The mathematical model developed to represent the WDC system and the optimizer assumes that all the data required for processing resides at the master node (as would be the case in a local processing scenario). This means that the master node will need to transfer all data that the slave node needs to process its portion of the task. Additionally, because the goal of this thesis is to characterize wireless distributed computing and its performance, the mathematical models assume that the algorithm to be executed is perfectly parallelizable.

3.3 Processing

In this section, the processing time and processing energy required to distribute a task using WDC are modeled.

3.3.1 Processing Time

If a certain algorithm is perfectly parallelizable, the time required to process that algorithm decreases proportionally to the number of nodes processing that algorithm. This relationship can be expressed as (3.1).

$$T_i[n] = \frac{K}{n} \tag{3.1}$$

Equation (3.1) is an idealized modeling of the processing time required to complete a job with K units of work and n nodes to process that job. It shows that in an idealized model, the time it takes to process a job is inversely proportional to the number of nodes that are processing that job. Equation (3.3) is modeled as a discrete function because n, the number of nodes available to process the task, is only defined for integer values. To provide a more realistic model, the amount of overhead time it takes to startup a process as a function of the number of processing nodes available must be taken into account. The overhead time is modeled in (3.2).

$$T_o[n] = \delta \cdot n \tag{3.2}$$

Equation (3.2) above shows that the overhead time associated with completing a task increases as the number of nodes available for processing increases. Combining (3.1) and (3.2) gives the mathematical model for the amount of time it takes to process a certain task using a set of nodes distributed over a wireless mesh network, represented by (3.3).

$$\begin{aligned} T_p[n] &= T_i[n] + T_o[n] \\ &= \frac{K}{n} + \delta n \end{aligned} \tag{3.3}$$

The model contains both the inverse relationship describing the performance increase one can achieve by distributing the job over multiple nodes, as well as the linear relationship which describes the performance decrease that occurs because of the overhead time required to start a process in the Android OS. Although $\lim_{n\to+\infty} T_p[n] = +\infty$, $T_p[n]$ is a convex function, as shown by Theorem 3.7.4 in section 3.7.4. This means that the processing time for a task decreases up to the critical point ($T_p'[n] = \delta - \frac{K}{n^2} = 0 \rightarrow n = \sqrt{\frac{K}{\delta}}$) of $T_p[n]$. Because $T_p[n]$ is a convex function, the function is decreasing up to the critical point which is on the interval of [0,n]. In the WDC context, the processing time decreases for certain values of K and δ as long as the number of nodes used in the calculation are in the interval of [0,n]. Figure 3.1 shows $T_p[n]$ for a fixed value of K and δ. It can be seen from Figure 3.1 that $T_p[n]$ is a decreasing function until the critical point. The critical point (which in the WDC context means the maximum number of nodes that can be used in the calculation before $T_o[n] > T_p[n]$) as a function of the ratio of the amount of work to be completed, K, and the overhead time required to start a process, δ, is shown in Figure 3.2.

Equation (3.3) does not account for the communication time required to transmit the data required for processing the algorithm or the communication time required to send the "start job" request. In order to properly model the total time required to distribute a job using WDC, the communication time must also be taken into account. These parameters are explored and developed later in this chapter.

3.3.2 Processing Energy

The amount of energy required to process a certain number of instructions (which equates to a job) can be viewed as a constant for a certain device. This is represented by (3.4).

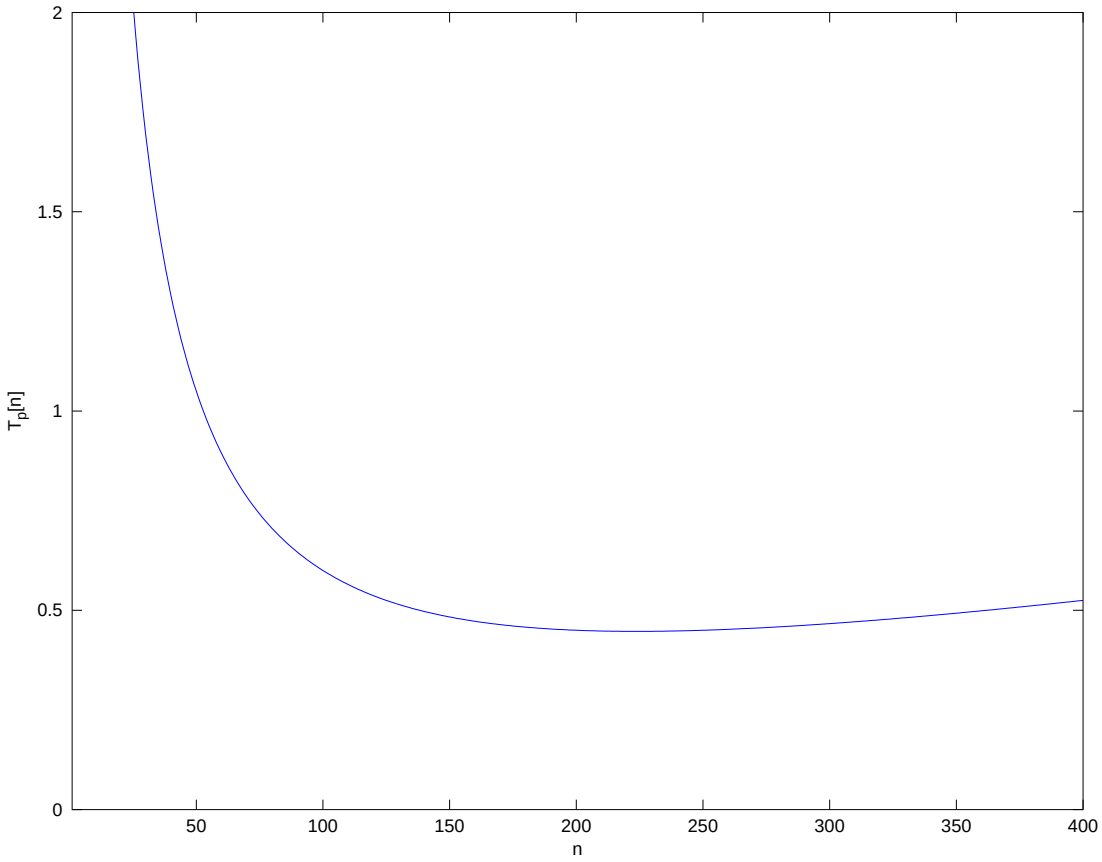

Figure 3.1: Convexity of $T_p[n]$

$K = 50$, $\delta = 0.001$

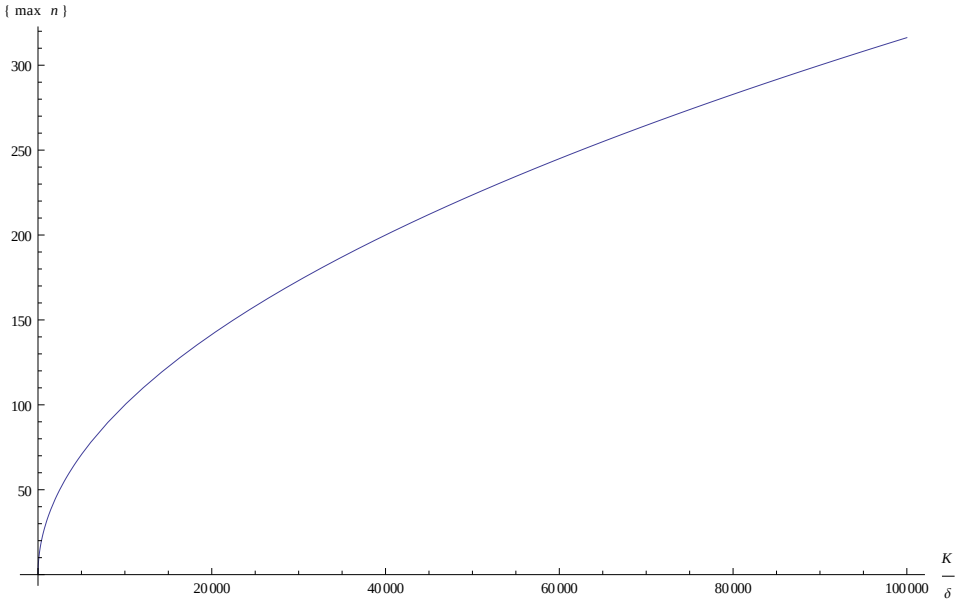

Figure 3.2: Critical Point as a function of $\frac{K}{\delta}$

$$E_i[n] = E \cdot K \tag{3.4}$$

It is important to note here that (3.4) does not take into account certain parameters of modern operating systems that cannot be modeled. Examples of this include scheduling of processes, and time slicing, and environmental factors. For example, if the environment that the processing unit is located in heats up, the cooling fan may be turned on by the operating system which in turn increases energy draw. Therefore, although it is difficult to accurately model energy draw, Equation 3.4 represents a simplistic model for total processing energy required to process K units of work.

Since WDC will distribute the job among many nodes, the amount of energy required to process the job (execute the same amount of instructions) with WDC will be equivalent to executing the job locally. However, the total processing energy required by WDC will be greater than executing the job locally because there is overhead energy required to start a process on the local executing environment. The overhead energy is represented by (3.5).

$$E_o[n] = \epsilon \cdot n \tag{3.5}$$

Combining Equations (3.4) and (3.5) yields the mathematical model for the amount of energy it takes to process a certain task using a set of nodes distributed over a wireless mesh network, as shown in Equation (3.6).

$$E_p[n] = KE + \epsilon n \tag{3.6}$$

The model shows that the total amount of processing energy required by WDC increases linearly, at a rate of ϵ, with an increase in the number of nodes available for processing. Equation (3.6) above however does not account for the communication energy required to transmit the data required for processing the algorithm or the communication energy required to send the "start job" request. Communication energy must be taken into account in order to accurately model the total energy required to distribute a job to wirelessly mesh networked devices. Communication energy is modeled later in this chapter.

3.4 Communication

In this section, the communication time and energy required to distribute a task using WDC is modeled. Requisite knowledge about mesh network topologies and mesh data models is also discussed, because it correlates to the communication time and energy required for WDC.

3.4.1 Mesh Network Topologies

In a wireless mesh network, the number of hops that two nodes that wish to communicate are away determine the energy and time required to communicate with another node. Network

topologies and routing protocols map directly to the number of hops through which data must travel to reach the destination node in a wireless mesh network. Many network topologies exist; Ad-Hoc, Linear, and Mixed network topologies are discussed below.

An Ad-Hoc mesh network as shown in Figure 3.3. As seen in Figure 3.3, in an Ad-Hoc mesh network, every node is one hop away and can communicate directly with any other node in the network. Because the hop count correlates to the energy required to communicate, it is useful to model the number of hops each node is away from node 1 (assumed to be the master node). Figure 3.4 shows the number of hops away node i is away from the master node. It models exactly what Figure 3.3 shows, that every node is one hop away from the master node. In this situation, since every node is the same number of hops away, the energy required to communicate with any node is the same.

A linear mesh network topology is shown in Figure 3.5. In a linear topology, every node can only communicate with its most immediate neighbor. Again, it is useful to model the number of hops each node is away from node 1 (assumed to be the master node). Figure 3.6 shows the discrete modeling of $m[i]$, as well as a continuous approximation. It shows that node i requires i hops to communicate with the master node. In this case, the energy required to communicate with node i is more than the energy required to communicate with node *i-1*.

Finally, a mixed mesh network topology is shown in Figure 3.7, in which every node can only communicate with its immediate north, south, east, and west neighbors. The number of hops each node is away from node 1 (assumed to be the master node) can be modeled as a discrete function. Figure 3.8 shows the discrete modeling of $m[i]$, as well as a continuous approximation. Similar to the linear mesh network, the energy required to communicate with node i is greater than the energy required to communicate with node *i-1*. Unlike the linear case however, the rate of energy increase is lower.

Table 3.2 summarizes the discussed network topologies as well as their mathematical hop count models.

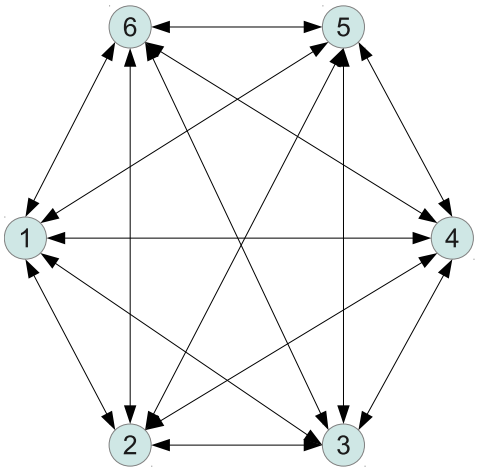

Figure 3.3: Ad-Hoc Network Topology

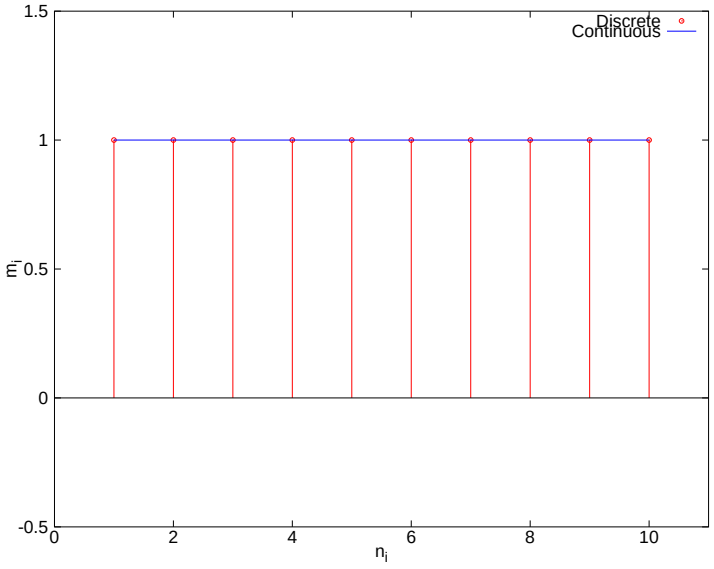

Figure 3.4: m(i) - Ad-Hoc Network Topology

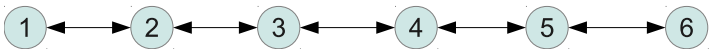

Figure 3.5: Linear Network Topology

Table 3.2: Mapping of network topologies and their mathematical hop count model

Network Topology	**Hop Model**	**Energy for communication with node i**
Ad-Hoc	$m[i] = 1$	$e_x \cdot m[i] = e_x$
Linear	$m[i] = i$	$e_x \cdot m[i] = e_x \cdot i$
Mixed	$m[i] \propto \sqrt{i}$	$e_x \cdot m[i] \propto e_x \cdot \sqrt{i}$

3.4.2 Mesh Network Data Throughput Models

In addition to the mesh topology, the data throughput model for the mesh network must be defined in order to accurately model the time and energy required to communicate between nodes in a wireless mesh network. Data throughput refers to how data is moved between nodes.

The router model of moving data through a mesh network is described by Figure 3.9. As shown in Figure 3.9, data is pipelined through the system. In Figure 3.9, k1, k2, k3, and k4 are all atomic units of work that must be transmitted from node one to node four. The pipelined model shown in Figure 3.9 is different from a store-and-forward model, where all required data is aggregated at each hop before going to the next hop. Only the pipelined data throughput model is considered in this thesis.

A linear network topology is shown in Figure 3.9 in order to highlight how data moves from node to node in the router model. As shown in Figure 3.9, atomic units of work are moved from node to node every time instance. It must be emphasized that the network topology has no effect on the general mathematical model for the amount of time it takes data to move through the system, as long as the number of hops and the way the data moves through the network is modeled. Network topologies force models, but do not dictate energy and time consumption directly. That general mathematical model is developed below.

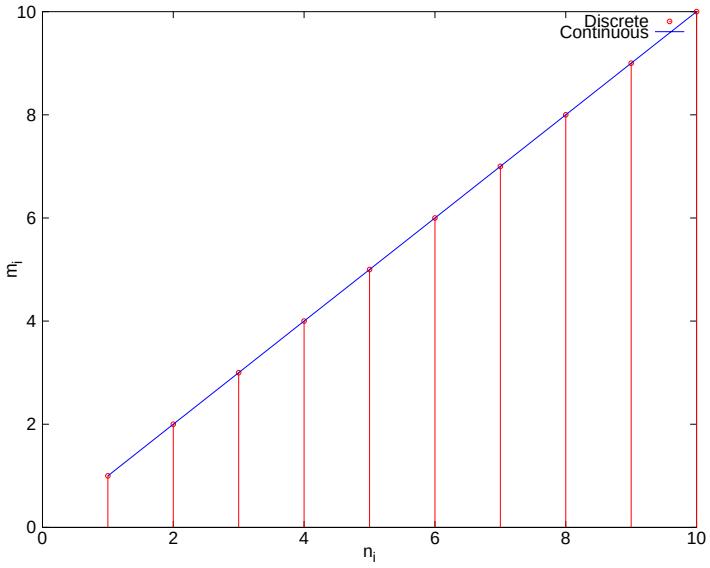

Figure 3.6: m(i) - Linear Network Topology

3.4.3 Communication Time

In section 3.3.2, the processing time and energy required for processing a job within WDC is modeled. However, in order to distribute a job over a set of nodes in a mesh network, there is an associated communication time and communication energy. These factors must also be accounted for, if the WDC system will accurately determine the optimal number of nodes to use for computation. The number of hops that a node is away from the node it needs to communicate with and the mesh data throughput model determines the communication time and energy required.

The most complete scenario for modeling communications time to enable WDC is where every node available for processing is an arbitrary amount of hops away, and the job to be performed is arbitrarily distributed among all available processing nodes. The mathematical model, assuming a router data throughput model described earlier, is given by (3.7).

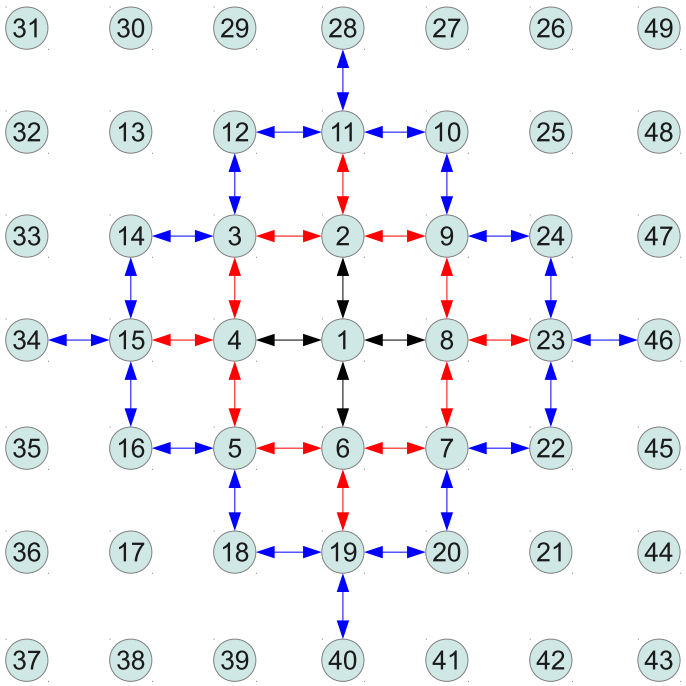

Figure 3.7: Mixed Network Topology

Black arrows represent 1^{st} hop

Red arrows represent 2^{nd} hop

Blue arrows represent 3^{rd} hop

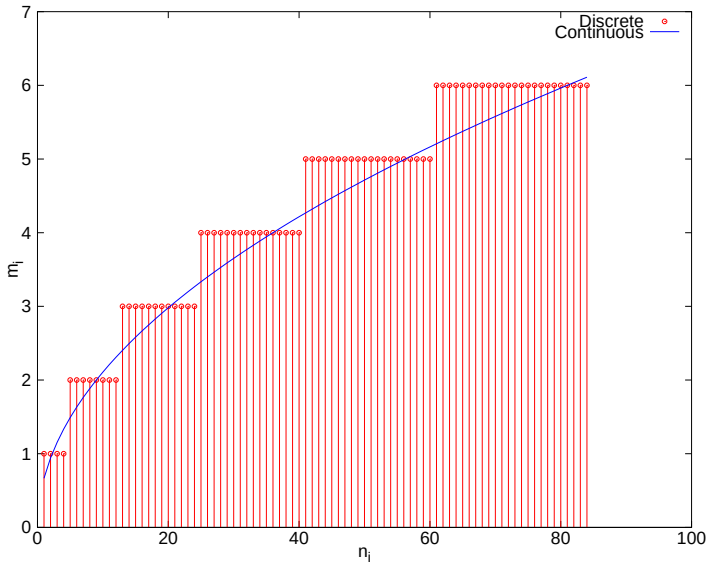

Figure 3.8: m(i) - Mixed Network Topology

$$T_c[n] = \sum_{i=1}^{n} [m_i t_o + (k_i + m_i - 1)t_x] \tag{3.7}$$

In (3.7), as given by Table 3.1, m_i represents the number of hops away that node i is from the master node, and k_i represents the amount of work that is assigned to node i. In (3.7), $m_i t_o$ represents the overhead time required to communicate to node i, and $(k_i + m_i - 1)t_x$ represents the amount of time required to transmit the data that node i requires to complete its processing.

Combining (3.3) and (3.7) yields the total time required to distribute a task to n nodes, shown in Equation (3.8).

$$\begin{aligned} T[n] &= T_p[n] + T_c[n] \\ &= \frac{K}{n} + \delta n + \sum_{i=1}^{n} [m_i t_o + (k_i + m_i - 1)t_x] \end{aligned} \tag{3.8}$$

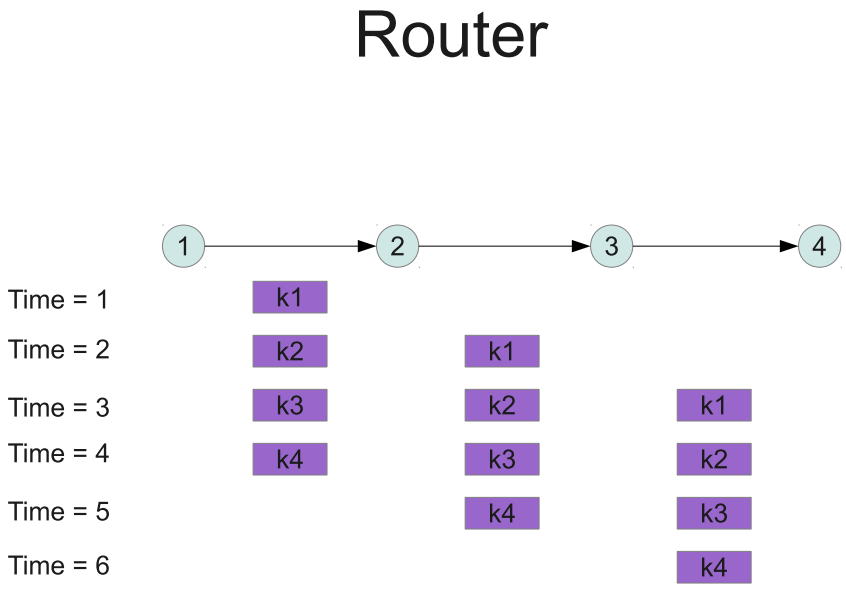

Figure 3.9: Router Model

3.4.4 Communication Energy

The most complete model for defining the requirements for communications energy to enable WDC is where every node available for processing is an arbitrary amount of hops away, and the job to be performed is arbitrarily distributed among all available processing nodes. The mathematical model, assuming a router data throughput model, is given by (3.9).

$$E_c[n] = \sum_{i=1}^{n} [m_i(k_i e_x + e_o)] \tag{3.9}$$

In (3.9), as given by Table 3.1, m_i represents the number of hops away that node i is from the master node, and k_i represents the amount of work that is assigned to node i. In (3.9), e_o represents the overhead energy required to establish a connection with the neighboring node, and $k_i e_x$ represents the energy required to transmit k units of work to the neighboring node. Therefore, total energy required to communicate is the product of the number of hops, given by m_i and the total energy required to communicate with the neighboring node, given by $(k_i e_x + e_o)$.

Combining 3.6 and 3.9 yields the equation for total energy, represented by (3.10).

$$\begin{aligned} E[n] &= E_p[n] + E_c[n] \\ &= KE_p + \epsilon n + \sum_{i=1}^{n} [m_i(k_i e_x + e_o)] \end{aligned} \tag{3.10}$$

3.5 Resource Function

A resource function, $R[n]$, can be defined as a weighted linear combination of the total time required for WDC given by (3.8) and the total energy required for WDC given by (3.10). This function represents the total amount of resources being used by the system to distribute a task among n nodes. The resource function is mathematically represented by (3.11).

$$\begin{aligned} R[n] &= \alpha T[n] + \beta E[n] \\ &= \alpha \left(\frac{K}{n} + \delta n + \sum_{i=1}^{n} \left[m_i t_o + (k_i + m_i - 1)\, t_x \right] \right) + \\ &\quad \beta \left(KE + \epsilon n + \sum_{i=1}^{n} \left[m_i \left(k_i e_x + e_o \right) \right] \right) \end{aligned} \tag{3.11}$$

The goal of the optimizer is to minimize the total resources used, which means minimizing $R[n]$.

3.6 Continuous Approximation

Equations (3.8) and (3.10) are discrete equations, based on m_i and k_i. m_i and k_i are discrete vectors because they are only defined for integer values of i. In order to analytically solve for an optimal number of nodes to use for WDC, it is convenient to approximate Equations (3.8) and (3.10) with continuous equations. This requires that m_i and k_i, which are vectors, be continuously approximated. Table 3.3 describes the mapping between the discrete vectors and their continuous representations.

Table 3.3: Discrete and Continuous Function notations

Discrete Vector	**Continuous Approximation**
m_i	$m(i)$
k_i	$k(i)$

Substituting m_i and k_i with their continuous approximations into (3.8) yields the continuous approximation of the total time required to perform WDC, shown below in (3.12).

$$\begin{aligned} T(n) &= T_p(n) + T_c(n) \\ &= \frac{K}{n} + \delta n + \int_{i=1}^{n} \left[m(i) t_o + \left(k(i) + m(i) - 1 \right) t_x \right] \, \mathrm{d}i \end{aligned} \tag{3.12}$$

Similarly, the continuous approximation of the discrete equation representing the total energy required to perform WDC, shown in (3.13), is found by substituting m_i and k_i with $m(i)$ and $k(i)$ in (3.10).

$$\begin{aligned} E(n) &= E_p(n) + E_c(n) \\ &= KE + \epsilon n + \int_{i=1}^{n} \left[m(i) \left(k(i) e_x + e_o \right) \right] \, \mathrm{d}i \end{aligned} \tag{3.13}$$

The continuous approximation of the resource function then becomes (3.14).

$$\begin{aligned} R(n) &= \alpha T(n) + \beta E(n) \\ &= \alpha \left(\frac{K}{n} + \delta n + \int_{i=1}^{n} \left[m(i) t_o + \left(k(i) + m(i) - 1 \right) t_x \right] \, \mathrm{d}i \right) + \\ &\quad \beta \left(KE + \epsilon n + \int_{i=1}^{n} \left[m(i) \left(k(i) e_x + e_o \right) \right] \, \mathrm{d}i \right) \end{aligned} \tag{3.14}$$

3.7 Convexity

Equation (3.14) represents a continuous approximation of the amount of resources that the WDC system will use, as a function of the number of available processing nodes. The goal then becomes minimization of the resource function. In order for a minimum to exist, the function must be convex over a certain interval. The convexity of the resource function is proven in this section.

A function f : $C \to \Re$, with domain $C \subset \Re^N$ is said to be a convex function if for any $\mathbf{x}, \mathbf{y} \in C$,

$$f\left(\lambda x + (1-\lambda)y\right) \leq \lambda f(x) + (1-\lambda) f(y)) \tag{3.15}$$

for any $\mathbf{x}, \mathbf{y} \in C$ with $x \neq y$ [13]. Additionally, the convexity of the differentiable function f : $R^n \to R$ can be characterized by conditions on its gradient ∇f and the Hessian $\nabla^2 f$. It can be shown that a twice differentiable function f is convex if and only if for all $x \in \mathbf{dom} f, \nabla^2 f(x) \succeq 0$, *i.e.*, its Hessian is semidefinite on its domain [13]. In order to prove that R(n) is a convex function, Lemmas 3.7.1, 3.7.2, and 3.7.3, are introduced below.

Lemma 3.7.1. *Linear functions are convex functions.*

Proof. Affine functions $f(x) = a^T x + b$, where $a \in R^n, b \in R$ are convex and concave because $\nabla^2 f \equiv 0$ [13]. □

Lemma 3.7.2. *Any conic combination of a convex function is a convex function. A conic combination is defined as (3.16).*

$$\sum_i^N \gamma_i f(x) \qquad \gamma_i \geq 0 \tag{3.16}$$

Proof. See [22]. □

Lemma 3.7.3. *Nonnegative infinite sums, integrals:* $p(y) \geq 0, g(x,y)$ *convex in* $x \implies \int p(y) g(x,y) dy$ *convex in x.*

Proof. See [13]. □

To prove that $R(n)$ is a convex function, it will be shown that $R(n)$ is a conic combination of convex functions.

Theorem 3.7.4. $R(n)$ *is a convex function.*

Proof. 1. $R(n) \triangleq \alpha \cdot T(n) + \beta \cdot E(n)$, where $[0 \leq \alpha, \beta \leq 1]$. Because $R(n)$ is a conic combination, as defined by Lemma 3.7.2, it can be proved that $R(n)$ is convex if $E(n)$ and $T(n)$ are shown to be convex.

2. $T(n) = T_p(n) + T_c(n)$. Because $T(n)$ is defined to be a conic combination, if $T_p(n)$ and $T_c(n)$ are shown to be convex functions, it can be proved that $T(n)$ is a convex function.

   (a) $T_p(n) = T_i(n) + T_o(n)$. Because $T_p(n)$ is defined to be a conic combination, if $T_i(n)$ and $T_o(n)$ are convex functions, then $T_p(n)$ is a convex function.

      i. $T_i(n) = \frac{K}{n}$ $K > 0$ is convex for $n > 0$. See [16].

      ii. $T_o(n) = \delta \cdot n$ $\delta > 0$ is convex, by Lemma 3.7.1.

      $\therefore T_p(n)$ is convex.

   (b) $T_c(n) = \int_{i=1}^{n} [m(i)t_o + (k(i) + m(i) - 1)\, t_x]\, \mathrm{d}i$.

      i. It is convenient to define $m'(i) \triangleq m(i) - 1$. Because $m(i)$ is guaranteed to be ≥ 1 (because the number of hops that a node is away from the master node must be at least one), $m'(i)$ will be ≥ 0. Both $m(i)$ and $m'(i)$ will be "never decreasing" functions.

      ii. $T_c(n) = \int_{i=1}^{n} [m(i)t_o + (k(i) + m'(i))\, t_x]\, \mathrm{d}i$.
      $= t_o \int_{i=1}^{n} m(i)\, \mathrm{d}i + t_x \int_{i=1}^{n} k(i)\, \mathrm{d}i + t_o \int_{i=1}^{n} m'(i)\, \mathrm{d}i$

         A. In the WDC frame of reference, $m(i)$ is a non-decreasing function. This is because by definition, $m(i)$ is an ordered function representing the number of hops away node i is from the master node. $\therefore$ $m(i)$ is a piecewise increasing linear function and thus by Lemma 3.7.1 is a convex function.

         B. By Lemma 3.7.3, $\int m(i)$ is convex. Similarly, $\int m'(i)$ is a convex function.

         C. $k(i)$, in the context of WDC, is defined to be an arbitrary positive function (positive because a node cannot perform negative work). If $k(i)$ is strictly constant or increasing function as $m(i)$, then $\int k(i)$ will be guaranteed to

be convex by Lemma 3.7.3, as is the case with $m(i)$. If $k(i)$ is decreasing, then the integral of $k(i)$ is not guaranteed to be convex, but may have some sections which are convex.

3. $E(n) = E_p(n) + E_c(n)$. Similar to $T(n)$, $E(n)$ will be shown to be convex because it is a conic sum of convex functions.

   (a) $E_p(n) = K \cdot E + \epsilon \cdot n$. By Lemma 3.7.1, $E_p(n)$ is a convex function.

   (b) $E_c(n) = \int_{i=1}^{n} \left[m(i) \left(k(i) e_x + e_o \right) \right] di$
   $= e_x \int_{i=1}^{n} \left[m(i) k(i) \right] di + e_o \int_{i=1}^{n} m(i) di$

      i. $\int_{i=1}^{n} m(i) k(i) di$

         A. $m(i)$ is convex by definition.

         B. Let's restrict $k(i)$ to be a constant (even distribution of jobs among all nodes). Define $\phi = k(i)$. Now, $\int_{i=1}^{n} m(i) k(i) di$ becomes $\phi \int_{i=1}^{n} m(i) di$, which is convex.

      ii. $e_o \int_{i=1}^{n} m(i) di$ is convex.

         A. By definition, $m(i)$ is an ordered function representing the number of hops node i is from the master node and as described before, is an affine function. $\therefore$ by Lemma 3.7.1, $m(i)$ is convex.

         B. By Lemma 3.7.3, $\int_{i=1}^{n} m(i) di$ is a convex function. The constant multiplier e_o does not affect the convexity of $\int_{i=1}^{n} m(i) di$.

      iii. $\therefore E_c(n)$ is the sum of convex functions, with the restriction that $k(i) = \phi$. $\therefore E_c(n)$ is a convex function.

   (c) $\therefore E(n)$ is a convex function by Lemma 3.7.2.

4. $T(n)$ and $E(n)$ are shown to be convex functions. Therefore, by Lemma 3.7.2, $R(n)$ is shown to be convex.

□

Because $R(n)$ is shown to be convex under the conditions of $k(i) = \phi$, it is guaranteed that $R(n)$ has a global minimum. With respect to WDC, the minimum of $R(n)$ corresponds to the optimization of the distribution of the job such that total system resources used are minimized.

3.8 Optimizer

The goal of the optimizer is to determine the optimal number of nodes to use, from the mathematical models describing the total energy and time required to complete a job using WDC. An optimization algorithm is based on the minimization of the resource function, $R(n)$.

Theorem 3.8.1. *There exists a minimum for $R(n)$.*

The mathematical foundation for minimization of $R(n)$ is detailed in Proof 3.8.

Proof.

1. $R(n) = \alpha T(n) + \beta E(n)$. These calculations are performed with $k(i) = \frac{K}{n}$, which means that the total amount of work to be completed, K, is evenly distributed among the available processing nodes, n. $k(i)$ is a constant with respect to i, so this meets the requirements to guarantee convexity of $R(n)$.

2. To minimize $R(n)$, the formula $R'(n) = 0$ must be calculated.

   (a) $R'(n) = \alpha T'(n) + \beta E'(n)$

      i. Recall that $T(n) = \frac{K}{n} + \delta n + \int_{i=1}^{n} \left[m(i) t_o + (k(i) + m(i) - 1)\, t_x \right] \mathrm{d}i$.

         A. $T'(n) = T_p'(n) + T_c'(n)$

         B. $T_p'(n) = \frac{\partial}{\partial n} \left(\frac{K}{n} + \delta n \right)$
            $= -\frac{K}{n^2} + \delta$

C. $T_c'(n) = \frac{\partial}{\partial n}\left(t_o \int_{i=1}^{n} m(i)di + t_x \int_{i=1}^{n} [k(i)+m(i)-1]\,di\right)$
$= \frac{\partial}{\partial n}\left((t_o+t_x)\int_{i=1}^{n} m(i)di + t_x \int_{i=1}^{n}(k(i)-1)di\right)$
$= (t_o+t_x)\frac{\partial}{\partial n}\int_{i=1}^{n} m(i)di + t_x\frac{\partial}{\partial n}\int_{i=1}^{n}(k(i)-1)di$

D. By the fundamental theorem of Calculus, $\frac{\partial}{\partial n}\int_{i=1}^{n} m(i)di = m(n)$

E. Similarly, $\frac{\partial}{\partial n}\int_{i=1}^{n}(k(i)-1)di = k(n)-1 = \frac{K}{n}-1$

F. $T'(n) = -\frac{K}{n^2} + \delta + (t_o+t_x)m(n) + t_x\cdot\left(\frac{K}{n}-1\right)$

ii. Recall that $E(n) = K\cdot E + \epsilon\cdot n + \int_{i=1}^{n}\left[m(i)\left(k(i)e_x+e_o\right)\right]\,\mathrm{d}i$

A. $E'(n) = E_p'(n) + E_c'(n)$

B. $E_p'(n) = \frac{\partial}{\partial n}\left(K\cdot E + \epsilon\cdot n\right)$
$= \epsilon$

C. $E_c'(n) = \frac{\partial}{\partial n}\left(\int_{i=1}^{n}\left[m(i)k(i)e_x + m(i)e_o\right]\right)$
$= m(n)\cdot\left(e_x\frac{K}{n}+e_o\right)$

D. $E'(n) = \epsilon + m(n)\cdot\left(e_x\frac{K}{n}+e_o\right)$

(b) $R'(n) = \alpha\cdot T'(n) + \beta\cdot E'(n)$
$= \alpha\left[-\frac{K}{n^2}+\delta+(t_o+t_x)m(n)+t_x\left(\frac{K}{n}-1\right)\right] + \beta\left[\epsilon + m(n)\cdot\left(e_x\frac{K}{n}+e_o\right)\right]$

(c) $R'(n) = 0$
$\frac{-K\alpha}{n^2} + \delta\alpha + \alpha(t_o+t_x)m(n) + \alpha t_x\left(\frac{K}{n}-1\right) + \beta\epsilon + \beta m(n)\left[e_x\frac{K}{n}+e_o\right] = 0$

□

In an Ad-Hoc WiFi network, every node is a one hop away. To solve for the optimal number of nodes, $m(n)$ can be substituted by 1 (as defined by Figure 3.3).

$$\frac{-K\alpha}{n^2} + \delta\alpha + \alpha(t_o+t_x) + \alpha t_x\left(\frac{K}{n}-1\right) + \beta\epsilon + \beta\left[e_x\frac{K}{n}+e_o\right] = 0 \tag{3.17}$$

Solving (3.17) for n would show the optimal number of nodes to use in an Ad-Hoc mesh network. The closed form solution for (3.17) is too mathematically complex to analyze

analytically. It is graphed below in Figure 3.10 to analyze with constants defined to show trends.

In a linear network topology, it was defined in Figure 3.5 that $m(i)$ is a linear function. Additionally by the fundamental theorem of Calculus, $\int_{i=1}^{n} m(i)di = m(n)$. It can be seen that as the number of nodes increases, $m(n)$ increases at a proportional rate. Mathematically, this can be expressed as $m(n) = n$. Substituting this into $R'(n)$, we get

$$\frac{-K\alpha}{n^2} + \delta\alpha + \alpha n(t_o + t_x) + \alpha t_x \left(\frac{K}{n} - 1\right) + \beta\epsilon + \beta n \left[e_x \frac{K}{n} + e_o\right] = 0 \tag{3.18}$$

Solving (3.18) for n would show the optimal number of nodes to use in an Linear mesh network. The closed form solution for (3.18) is too mathematically complex to analyze analytically. It is graphed below to analyze with constants defined to show trends.

Finally, in a mixed network topology, it was defined in Figure 3.7 that $m(i)$ is a square-root function. Additionally by the fundamental theorem of Calculus, $\int_{i=1}^{n} m(i)di = m(n)$. For the mixed network topology, $m(n) = \sqrt{n}$. Substituting this into $R'(n)$, we get

$$\frac{-K\alpha}{n^2} + \delta\alpha + \alpha\sqrt{n}(t_o + t_x) + \alpha t_x \left(\frac{K}{n} - 1\right) + \beta\epsilon + \beta\sqrt{n} \left[e_x \frac{K}{n} + e_o\right] = 0 \tag{3.19}$$

Solving (3.19) for n would show the optimal number of nodes to use in an mixed mesh network. The closed form solution for (3.19) is too mathematically complex to analyze analytically. It is graphed below to analyze with constants defined to show trends.

Because equations (3.17), (3.18), and (3.19) are too complex to analyze mathematically, they were numerically evaluated to show trends. Figure 3.10 shows the optimal number of nodes, n, to use as the amount of work, K, increases in the Ad-Hoc, Linear, and Mixed mesh network topologies. In order to generate numerical values, certain variables had to be fixed. Figure 3.10 shows that the optimal number of nodes to use in a WDC calculation is dependent upon the network topology. As the number of hops to communicate with slave

nodes increases, it becomes less useful to use WDC. Intuitively, this makes sense because as the communications overhead increases, it is more useful to process the job locally or with nodes that are closer than transmitting the job to nodes which take a lot of energy to communicate with.

Figure 3.10 showed how the optimal number of nodes to use varies based on the total amount of work to be completed. It also shows that it is optimal to use more nodes if the number of hops required to reach those nodes is low. This makes logical sense because as the number of hops required to reach node i goes up, so does the communication time and energy. It is also useful to see how the optimal number of nodes use varies, if the atomic work unit size is varied.

Because equations (3.17), (3.18), and (3.19) are too complex to analyze mathematically, they were numerically evaluated to show trends. Figure 3.11 shows how the atomic work unit size should be programmed for a certain amount of available nodes for processing. Similar to figure 3.10, figure 3.11 shows that the atomic work unit size to use in a WDC calculation is dependent upon the network topology. Intuitively, Figure 3.11 makes sense because if more nodes are available for processing, it makes sense to distribute the job into smaller pieces such that all available nodes are utilized.

From figures 3.10 and 3.11, it can be seen that the fundamental optimization equation can be used in multiple ways. The fundamental optimization algorithm is repeated below in (3.20).

$$\frac{-K\alpha}{n^2} + \delta\alpha + \alpha(t_o + t_x)m(n) + \alpha t_x\left(\frac{K}{n} - 1\right) + \beta\epsilon + \beta m(n)\left[e_x\frac{K}{n} + e_o\right] = 0 \qquad (3.20)$$

In (3.20) for a practical scenario, the fixed parameters are: 1.) the network topology given by $m(n)$, 2.) the processing time weight given by α, 3.) the processing energy weight given by β, 4.) the startup time of a process given by δ, and 5.) the startup energy of a process given by ϵ. The two variable parameters are 1.) the optimal number of nodes to use in a

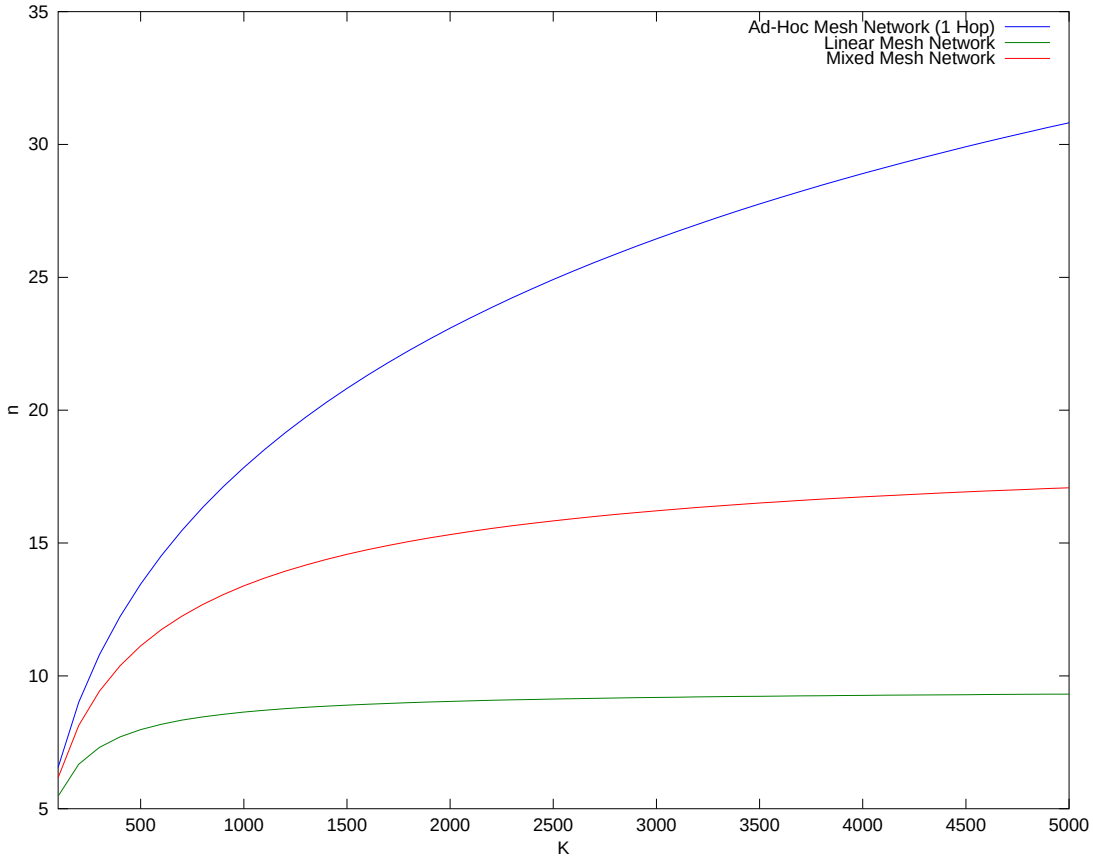

Figure 3.10: Optimal Number of Nodes for different network topologies with K as parameter

$$\alpha = 1$$

$$\beta = 1$$

$$\delta = 1$$

$$\epsilon = 1$$

$$e_x = 0.01$$

$$e_o = 0.01$$

$$t_x = 0.01$$

$$t_o = 0.01$$

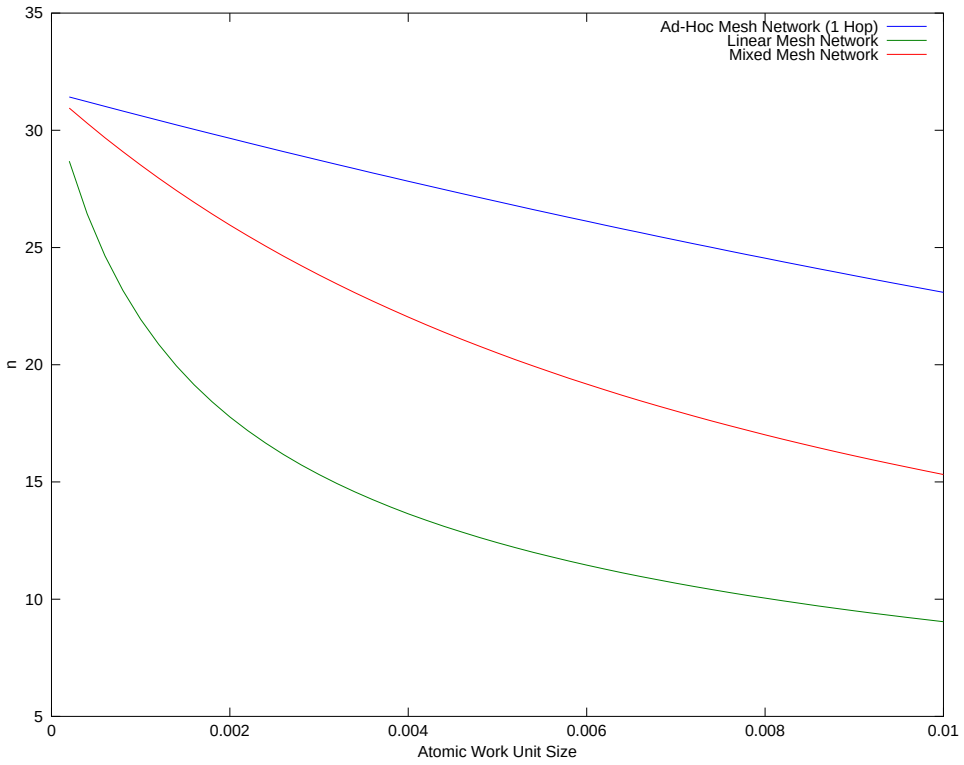

Figure 3.11: Optimal Number of Nodes for different network topologies with the atomic work unit size as parameter

$$\alpha = 1$$

$$\beta = 1$$

$$\delta = 1$$

$$\epsilon = 1$$

$$K = 2000$$

calculation or 2.) how to "chunk" up the work to maintain optimality. The fundamental equation given by (3.20) can be used to solve for either, based on the user's concept of operations (CONOPs).

Chapter 4

Simulation and Results

4.1 Introduction

Chapter 2 discussed the Android OS and the implementation of the WDC application. Chapter 3 discussed the theory behind calculating the optimal number of nodes that should be used in a WDC calculation. This chapter details the testing of the WDC application. The testing environment and test setup are detailed. Next, the mesh network implementation on the mobile phones is discussed. Finally, the tests completed and their corresponding results are shown.

4.2 Testing Environment

The WDC application implemented in Android and its performance was tested on four Samsung Galaxy S phones. Three of the phones had a custom ROM running Android 2.3.6 (Gingerbread) because the operating system level software had to be modified to allow the phones to see Ad-Hoc WiFi networks. The fourth phone was running Android 4.0.1 (Ice Cream Sandwich) and was stock ROM provided by the manufacturer. It was however

Figure 4.1: Test Setup

"rooted" in order to allow it to use android-wifi-tether (open source software that broadcasts Ad-Hoc WiFi networks) to run on the device. Additionally, the testing was completed with the phones placed in close proximity (approximately 6" away) of each other in the same room. Although the phones communicate wirelessly to perform distributed computing, the automated test environment, which is discussed in detail below, required all of the phones to be connected to a computer via USB cable. It is important to emphasize here that the USB connection was not used for WDC data transfer, but rather for debugging data used to accumulate performance metrics of the WDC system. All phones used for testing were loaded with the same version of the WDC software. Figure 4.1 shows the phones all connected to a USB hub, which is connected to the laptop computer that orchestrated the testing.

As explained in Chapter 2, the WDC application is designed to take advantage of any communications technology for mesh networking, as long as it conforms to the API that is specified by the WDC communications interface. The WDC application however does not provide software that creates a mesh network. It merely uses the networking stack provided to it by the system. Therefore, a mesh networking solution must be implemented on the

Android device(s) before being able to use the WDC application.

4.2.1 Mesh Networking Implementation

The Android OS is an interesting platform for mesh networking because of two opposing factors: 1.) it is a Linux based operating system which has general support for a wide variety of commercial off-the-shelf (COTS) hardware, and 2.) the OS is generally used on mobile phones, which are tightly integrated (in terms of hardware) devices. The second reason forces the Android mesh network to be based on a standard that can be implemented with hardware on-board the mobile device. The communications devices that are generally available on an Android device are: 1.) Infrared communications chipsets, 2.) Bluetooth communications chipsets, 3.) WiFi communications chipsets, and 4.) 3G/4G chipsets. The unstated goal of WDC is to have the ability to perform distributed computing without any infrastructure, with the ability to perform computations with nodes at a reasonable distance. The infrastructure restriction rules out using the 3G/4G chipsets for creating a mesh network. The reasonable distance restriction rules out the infrared communications chipsets for creating a mesh network. Due to open-source software readily available that utilizes WiFi, the WiFi chipset was chosen to be the device used to implement the mesh network.

Using a WiFi chipset on an Android device implies that the WiFi (802.11) standard will be the basis of the mesh networking. The WiFi standard supports mesh networking both at Layer 2 (MAC layer) and Layer 3 (IP layer) of the OSI model. Layer 2 mesh networking is achieved using 802.11s, which is a protocol amendment to the 802.11 standard that defines mesh networking at the MAC layer. Layer 3 mesh networking is achieved by creating an Ad-Hoc WiFi network on a selected Android phone, and having the other phones connect to the Ad-Hoc network.

Layer-2 Mesh Networking - 802.11s

The first way to create a mesh network in Android is to create it at Layer 2 of the OSI model (MAC layer) using 802.11s. 802.11s is an amendment to the IEEE 802.11 standard. It is an extension to the 802.11 MAC layer to support unicast/multicast/broadcast delivery using "radio aware metrics over self-configuring multi-hop technologies" [12]. Layer 2 mesh networking is an efficient implementation of mesh networking because it takes into account link metrics that are not available at Layer 3. Additionally, because it is implemented at Layer-2, it integrates seamlessly into any application that operates at OSI Layer-3 or above.

802.11s is an emerging standard, and an open implementation is available at o11s.org. Because 802.11s is a new amendment to the 802.11 standard (ratified 2012), many WiFi chipsets do not implement this standard natively in hardware. Because of this restriction, in order to support 802.11s, the WiFi chipset must be configurable. In WiFi terminology, a configurable WiFi chipset is also known as a SoftMAC device. The problem with popular mobile devices which run Android is that they generally do not have SoftMAC WiFi chipsets. Without a SoftMAC WiFi chipset, the Android OS will be unable to create a mesh network at Layer-2. It is important to note that the Android OS itself is not the shortcoming here, but rather the hardware that is used by manufacturers of mobile devices.

Layer-3 Mesh Networking - Ad-Hoc WiFi

The second way to create a mesh network with an Android enabled device is to create it at Layer 3 (Network layer) using the Ad-Hoc WiFi protocol. Using an Ad-Hoc WiFi system requires that one of the nodes in the mesh network broadcast an SSID for an Ad-Hoc wireless network. The further complication with Ad-Hoc networks is that the wpa_supplicant software running in Android is, by default, configured to ignore Ad-Hoc WiFi networks. Additionally, broadcasting a WiFi network is not built into the Android OS. Because 802.11s is not widely supported by the Android OS due to hardware limitations, overcoming the limitations of the

wpa_supplicant and mesh network, broadcasting and creating an Ad-Hoc WiFi is the best option for creating the wireless mesh network.

An Android mobile device needs to be running specialized software that creates an Ad-Hoc WiFi network. The popular open source android-wifi-tether application was used to create an Ad-Hoc mesh network on the master node [1]. This required that the phone be "rooted." The designated phone for this was the master node, which was running the stock Android 4.0.1, with superuser privileges. Therefore in the test environment, the master node was always designated to be the phone running Android 4.0.1 with the android-wifi-tether application. In order to allow the other phones to see the Ad-Hoc network that was being broadcast by the master node, the wpa_supplicant software was modified with an open source patch that enabled the phones to see Ad-Hoc WiFi networks [3].

Wireless Application

The wireless application that was chosen to test WDC performance was a facial recognition application. The application was built using the JavaCV open source library. JavaCV provides image processing routines built upon OpenCV (Open Computer Vision) [2]. These routines were used to create an Android service called FaceRecognitionService that is activated when an Intent with action **wdc.user.services.facerecognize** is broadcast. The WDC application on a slave node receives a JobRequestMessage. In the testing scenario, the **intent_to_fire** XML tag contains the Intent to start the FaceRecognitionService. In the Intent, a file which contains the URI's to all the face images to be recognized is specified. This information comes from the XML that the master node sends to all the processing nodes. The FaceRecognitionService then recognizes all of these faces, and places the results in a file. The results are then sent back to the WDC application, which then sends the results wirelessly back to the master node.

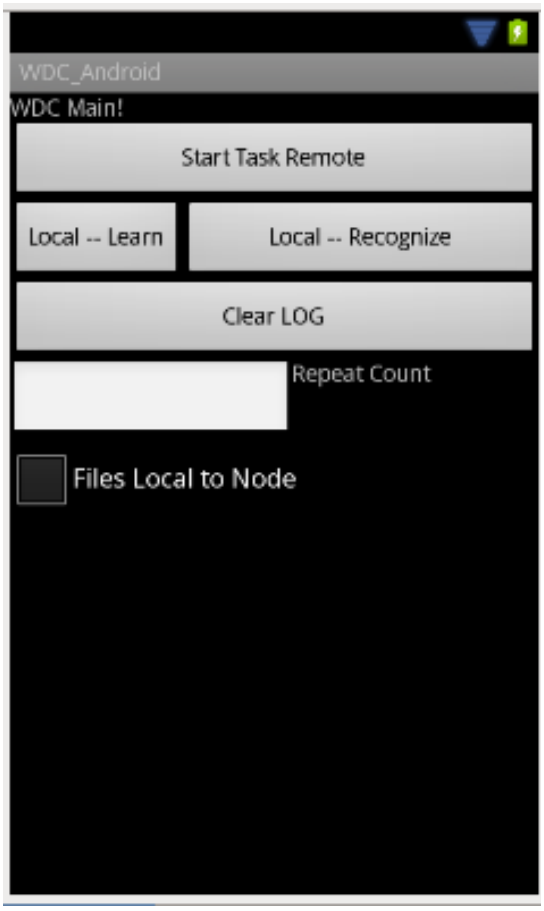

Figure 4.2: WDC Application Activity

4.2.2 Testing Procedure

As stated throughout this thesis, the premise of WDC is that it provides a performance increase over local processing. In order to properly test the performance of WDC, the FaceRecognitionService wireless application was chosen as the processing back-end. The atomic work unit for the FaceRecognitionService testing was defined to be the recognition of 45 faces.

Figure 4.2 shows the WDC application that was built to test WDC performance. The WDC application has multiple buttons, and a log area at the bottom half of the screen to inform the user of what is currently happening with WDC. The "Start Task Remote"

button begins a WDC job. A properties file embedded in the application tells the master node how many slave nodes exist and what their IP addresses are. This is preconfigured before the application is loaded onto the mobile device. When the "Start Task Remote" button is clicked, the WDC application reads the properties file and determines how many nodes there are to use for processing. The WDC application also reads the value of "Repeat count." The repeat count refers to the number of times one atomic work unit is re-processed. The reason for repeat count is that the FaceRecognitionService requires a database of faces to recognize. The database that was used for the FaceRecognitionService had 400 faces. For example, if 800 faces need to be recognized, the same set of faces must be recognized twice to simulate the processing load of processing 800 faces. Therefore, the repeat count is used to perform longer running calculations to test the performance of WDC.

Generally in WDC, it is assumed that when a master node distributes a job, it also passes the required data to the node it is requesting processing from. If the "Files Local to Node" checkbox is checked, the master node assumes that the slave node has all the required data for processing and only sends a JobProcessing request. This is not a realistic use scenario, but is provided for debugging purposes.

The suite of tests that were run were designed to test WDC performance over a range of differing work amounts and a parametric number of available processing nodes. Designing the tests in this fashion allows us to determine the usefulness of WDC, as the number of available processing nodes and the amount of work to be completed increases. In order to develop these results, the WDC performance was measured with processing one atomic unit of work to 21 atomic units of work, with incrementing by one atomic unit at a time. Additionally, each processing task was run five times to get a smooth average of the results. Finally, this suite of tests was run locally, with one processing node, two processing nodes, and three processing nodes at one compression level. This ends up being 315 tests to run.

Initially, testing was conducted by loading the application onto the mobile devices manually and clicking the buttons on the screen to commence tests. However, this proved to be

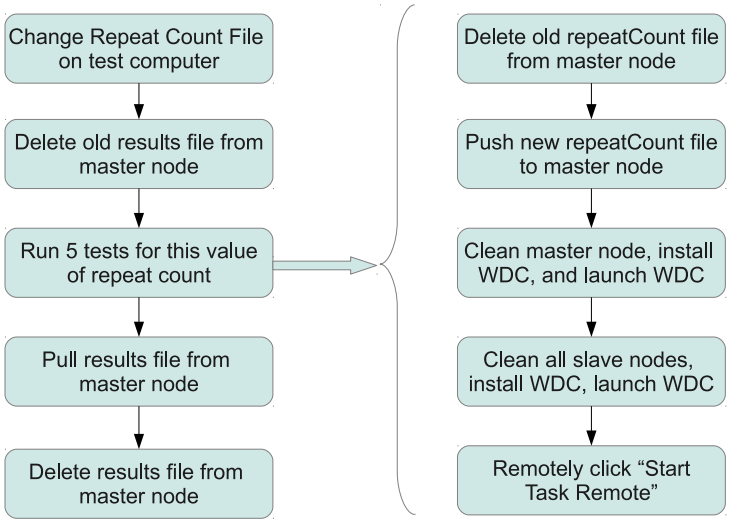

Figure 4.3: Ant Script Flow

cumbersome as the number tests to be conducted increased. In order to automate testing, Ant build scripts and Python scripts were used for testing.

The flow of the Ant script used for testing is described in Figure 4.3.

1. The first thing that happens when the Ant script is started is the "repeatCount" variable is incremented. It initially starts out at one (recognizing 45 faces), and increments by one up to 21 (recognizing 945 faces). As stated earlier, the "repeatCount" represents the number of times an atomic work unit is re-processed, to simulate a long running task. The value of the current repeatCount is stored in a file. This file is pushed to the master node in a later step, so the master node knows how much work to assign the slave nodes.

2. After the value of the repeatCount is modified on the local computer, the Ant script deletes any result files which may exist on the master node. Result files are files which

are accumulated by the master node after a test is complete. More information about results is discussed in later steps.

3. After the results files are deleted, 5 tests are run and the results accumulated. The reason for running five of each test is to get an average that smooths out variances due to communication and other environmental effects. The substeps outlined below are the steps that are performed to run a test.

   (a) Delete the old repeatCount file from the master node.

   (b) Push the new repeatCount file up to the master node. The repeatCount file is used by the WDC application to decide how much work to distribute to each of the slave nodes. It is important to note here that in operational usage, the WDC application would have no concept of "repeatCount." The "repeatCount" is used only to simulate varying amounts of processing.

   (c) After the "repeatCount" is setup, the master node is cleaned. This involves removing the old results file, as well as removing a "complete" file, which is checked by the host computer to determine whether the test is complete or not. The final step of cleaning the master node is to uninstall the old WDC application. This is done to ensure that no WDC information is maintained in cache, to obtain accurate performance metrics.

   (d) After the master node is setup, all participating slave nodes are cleaned of the WDC application and any data required for processing is removed. Again, this is to ensure that nothing is cached in memory to obtain more accurate performance metrics.

   (e) Remotely click the "Start Task Remote" button. After the start button is clicked, the computer waits for the 'complete' file to appear in the master node. This indicates that the test is complete, and the test process is repeated five times.

4. The results from the master node are pulled down and stored on disk. The master node accumulates results files every time a test is run. After the completion of a set of

tests (which is defined to be 5 tests run with the exact same parameters), results are stored and pulled from the master node.

5. The results file is the deleted from the master node. This happens to clear the master node for new results files to be created.

4.3 Testing and Results

In order to test the performance of WDC, the amount of work to process (controlled by "repeatCount") and the distribution of work was controlled. Because all the processing nodes are networked using Ad-Hoc WiFi, all nodes are one hop away. Therefore, all nodes received the same amount of work to process. The amount of work each node performs is determined by the (4.1).

$$amtWork = \frac{repeatCount}{totalNodesAvailableForProcessing} \tag{4.1}$$

When the master node begins a WDC job, as explained above, it sends a URL to each of the slave nodes which contains all the files to be gotten for processing a task. In this specific case, the files contain the URLs of each of the images that needs to be downloaded for processing and the facial recognition database. The webserver on the master node serves all of these files to each of the slave nodes simultaneously via the HTTP protocol. In order to minimize the overhead required to setup a new HTTP connection every time a file is requested by a particular slave node, it was decided that all the image files required for transmission would be compressed by the zip protocol before transmission. From an optimizer perspective, the compression directly relates to the trade off between processing time/energy and communication time/energy. This is because if a higher compression level is chosen, it takes the local node a longer amount of time to process but because the files are more compressed, a shorter amount of time to communicate that data to the node that

requires it. Only one compression level was tested.

Figure 4.4 shows the processing units, K, versus the job time using local processing, distributing the job to one slave node, two slave nodes, and three slave nodes. In each configuration, the total amount of work that is performed is the same and evenly distributed among the number of slave nodes. As was defined earlier, an atomic unit of work was defined to be recognizing 45 faces. In the one slave node configuration, all 45 faces were recognized by that one slave node. In the two slave nodes configuration, the first slave node recognized 23 faces and the second slave node recognized 22. Finally, in the three slave node configuration, each slave node recognizes 15 faces. Figure 4.4 shows that as the amount of work to be completed increases, the performance of the WDC system increases. It also shows that using three slave nodes is better than using two, or one, or performing the processing locally as the amount of work to be completed increases.

Figure 4.5 shows the time required to compress all the files for all the slave nodes, in all three cases. Obviously, local processing is omitted here because no compression is required. As explained above, slavenodes download the files after all the required files for processing have been compressed. In the current WDC implementation, compression of the files is performed serially (meaning that if three different work packages need to be made for the $n = 3$ case, three packages are made one at a time rather than in three separate threads), so the times shown in 4.5 is the time required to create the compressed packages for all the slave nodes, in the $n = 1, 2, 3$ case.

Figure 4.6 shows the average time required to download the compressed file for each slavenode. Because the file download is a parallel operation, the file download time for each slave node in the $n = 1, 2, 3$ cases are averaged. Specifically, the data plotted for the $n = 2$ case is the average file download time of each slavenode in the $n = 2$ case. Similarly, the data plotted for the $n = 3$ case is the average file download time of each slavenode in the $n = 3$ case. Figure 4.6 shows that as the amount of work to be completed, K, increases, the file download time increases. This is because as the amount of work to be completed increases,

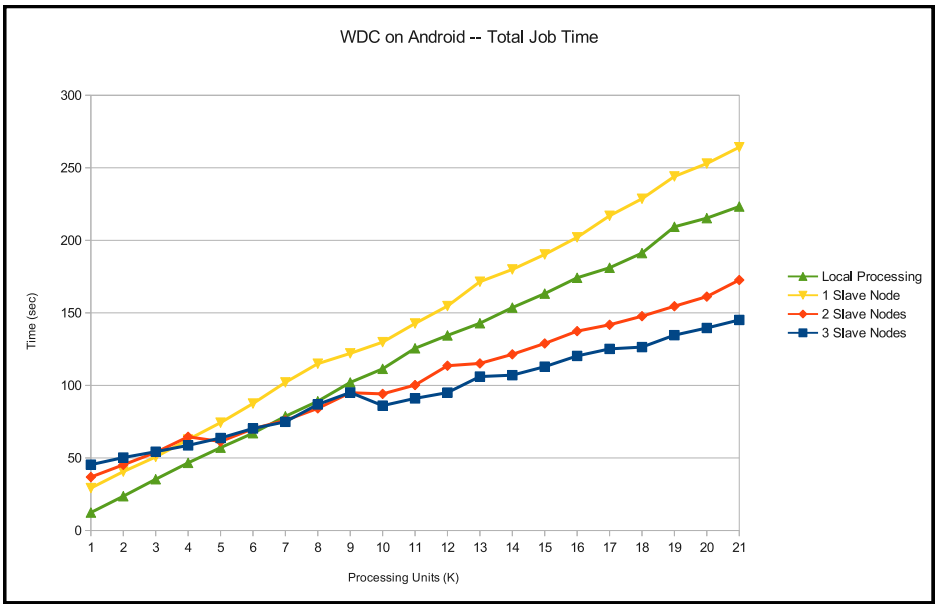

Figure 4.4: WDC Results - Overall Job Time - ZIP Compression Level=1

the file size of the work packages that need to be transmitted to each slave node increases.

Figure 4.7 shows the average time required to decompress all the required files, for each slave node in the $n = 1, 2, 3$ cases. Because decompression is a parallel process, the decompression times for each slave node in the $n = 1, 2, 3$ cases were averaged. Using the download times, it is possible to determine the average datarates that each slave node is able to acheive while downloading. Figure 4.8 shows this. Additionally, Figure 4.9 shows the total data rate that the masternode acheives, for the $n = 1, 2, 3$ cases.

Finally, 4.10 shows the average time required to process the data with the facial recognition algorithm for each of the slavenodes, in the $n = 1, 2, 3$ cases. Similar to the file download time graphs and the decompression time graphs, the data plotted was averaged among all slave nodes in the multiple slave nodes tests.

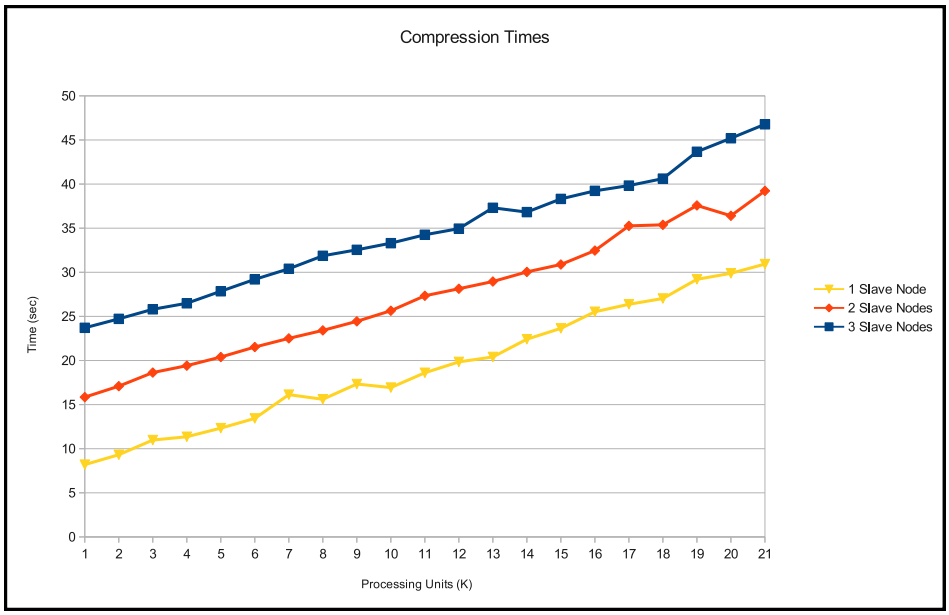

Figure 4.5: WDC Results - Compression time required for all files - ZIP Compression Level=1

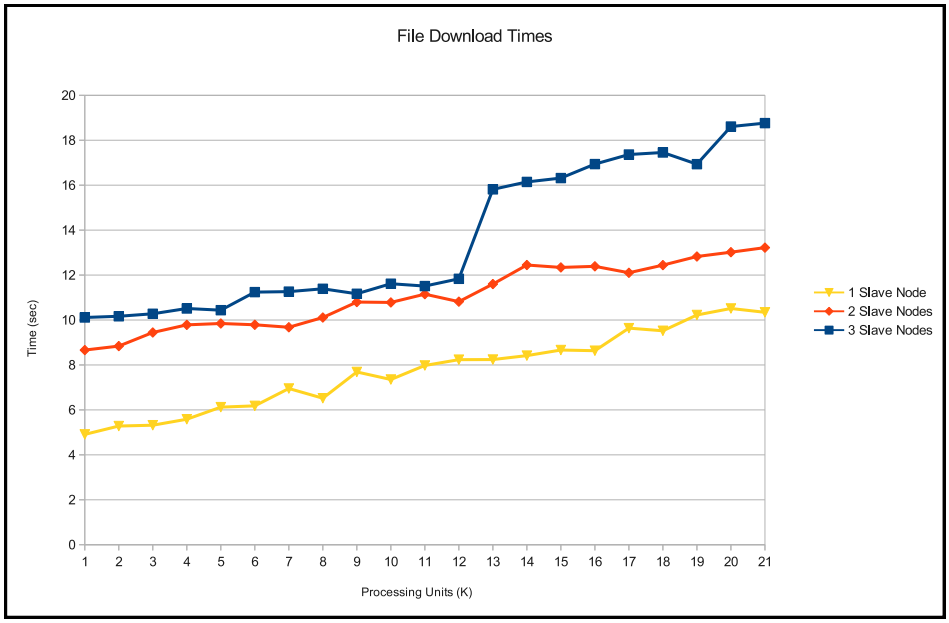

Figure 4.6: WDC Results - 3 Slave Nodes File Download + Decompression time - ZIP Compression Level=1

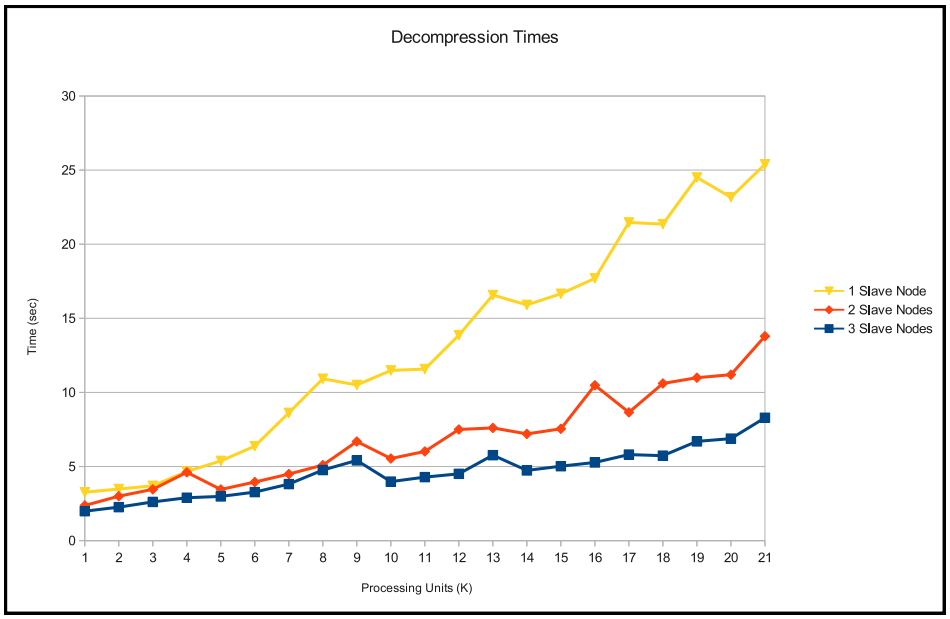

Figure 4.7: WDC Results - Decompression time required for all files - ZIP Compression Level=1

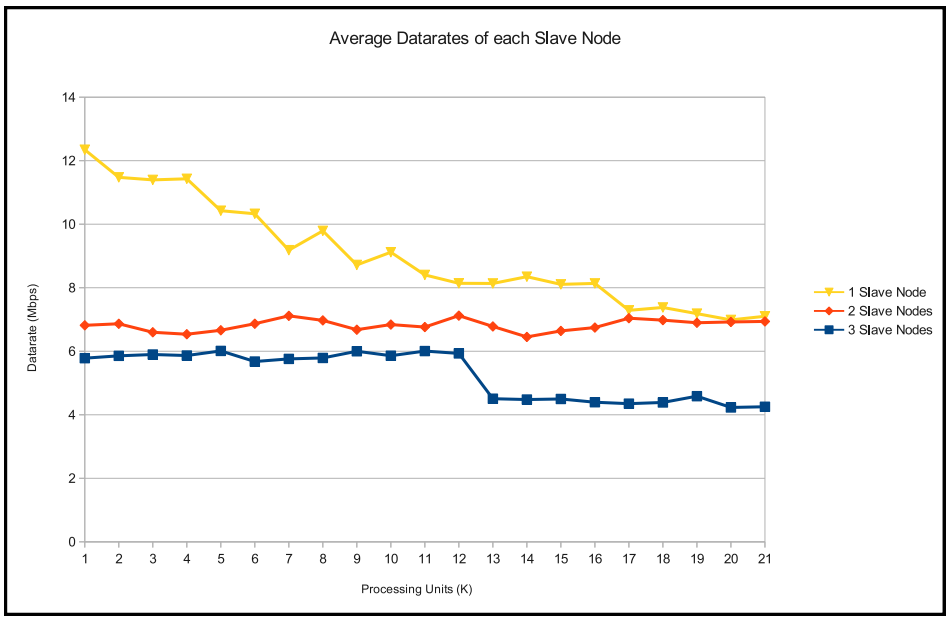

Figure 4.8: WDC Results - Average datarates of each node - ZIP Compression Level=1

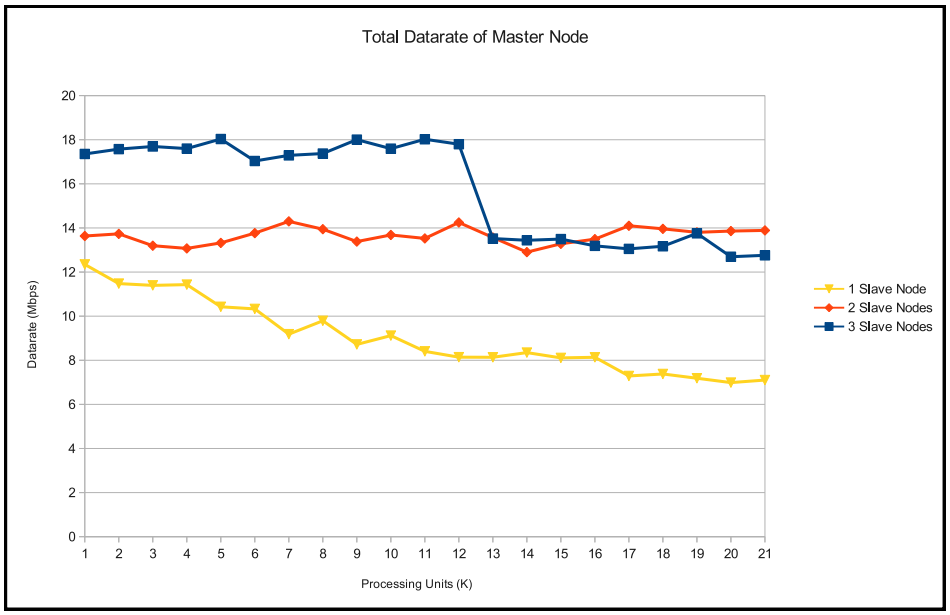

Figure 4.9: WDC Results - Total datarate of master node - ZIP Compression Level=1

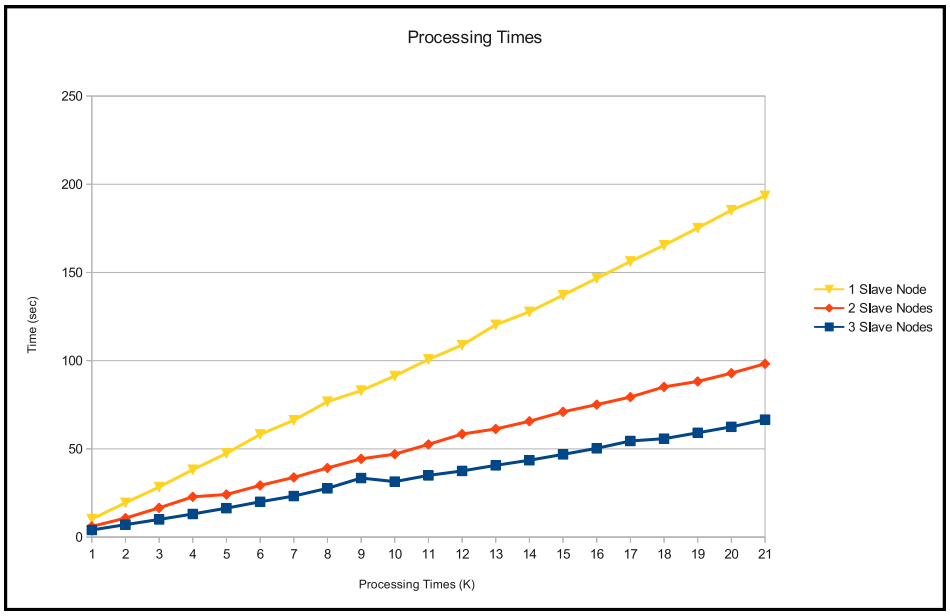

Figure 4.10: WDC Results - Average processing time for each slavenode - ZIP Compression Level=1

4.4 Theoretical Model Validation

The theoretical models developed in Chapter 3 are difficult to validate on the Android platform, because there are many constants that cannot be measured accurately. Specifically, the constants E, ϵ, e_x, and e_o, are not easily measurable on the Samsung smartphones that the experiments were conducted on, due to the granularity with which battery levels are reported on this specific device. However, the theoretical models for $T_c[n]$ and $T_p[n]$ were validated against experimental data, by measuring the constants t_x and t_o. In the test setup, t_o can be considered 0. This is because as soon as the mesh network is setup, it remains connected and no messaging is required to keep the connection alive. Experimentally, t_x was measured to be, on average, 0.18.

Figure 4.11 shows the experimental processing times versus the theoretically predicted ones. It shows that the mathematical model for $T_p[n]$ is accurate. Figures 4.12, 4.13 and 4.14 show the experimental file transfer times (measured as the time required to compress and package all required data by all nodes, download the required file for processing and decompressing that file) versus the theoretically predicted times in the one slave node, two slave nodes, and three slave nodes scenario. The figures show in general that the theoretical model for $T_c[n]$ follows the experimental results, although there is variability in the experimental results, due to environmental conditions which are out of the control of WDC.

As described above, although the processing time, T_p, is accurately predicted by the model, the communication time T_c exhibits some variability. This is due to the inherent variability of the RF environment, as well as the fact that the ISM bands are being used for RF communications [19]. The interference in the RF environment, specifically in the ISM bands, has been well documented [18] [14]. Because a lot of commercial equipment operates in the ISM bands such as Bluetooth and even commercial microwaves, there is random interference at anytime and this could severely impact the performance of the WiFi mesh network [18]. The random interference in the ISM bands combined with the uncontrolled RF test environment accounts for the variability in the communications time measured in the experiments.

From 4.6, it can be seen from the three slave nodes case that the download time shows an uncharacteristic spike at $K = 13$. Incidentally, the tests for the three slave nodes case for $K = 1$ to $K = 12$ were run on a separate day than the three slave nodes case for $K = 13$ to $K = 21$. The uncontrolled RF environment yielded a different effective datarate, as shown in Figure 4.8.

The constants, t_x and δ, were measured and validated against experimental results, as shown in Figures 4.11, 4.12, 4.13, and 4.14. The constants t_x and δ were substituted into Equation 3.20, with $\alpha = 0$, to determine the optimal number of nodes, with K as a parameter. Figure 4.15 shows the optimal number of nodes to use, with the parameters measured for the experiments conducted with the Samsung phones.

4.5 Model Anomalies

The models for $T_p[n]$ and $T_c[n]$ were validated in the previous section. However, an anomaly was noticed in the model verification of $T_c[n]$. The theoretical model for $T_c[n]$ is repeated below in Equation (4.2).

$$T_c[n] = \sum_{i=1}^{n} [m_i t_o + (k_i + m_i - 1)t_x] \tag{4.2}$$

In the experimental test setup, the constant t_o can be essentially considered zero. This is because once the mesh network is setup with the android-wifi-tether App, there is no overhead required to establish a connection. The connection is maintained until the mesh network is brought down by the node hosting the Ad-Hoc IBSS broadcaster (the master node in this case). This reduces the equation for $T_c[n]$ to (4.3).

$$T_c[n] = \sum_{i=1}^{n} [(k_i + m_i - 1)t_x] \tag{4.3}$$

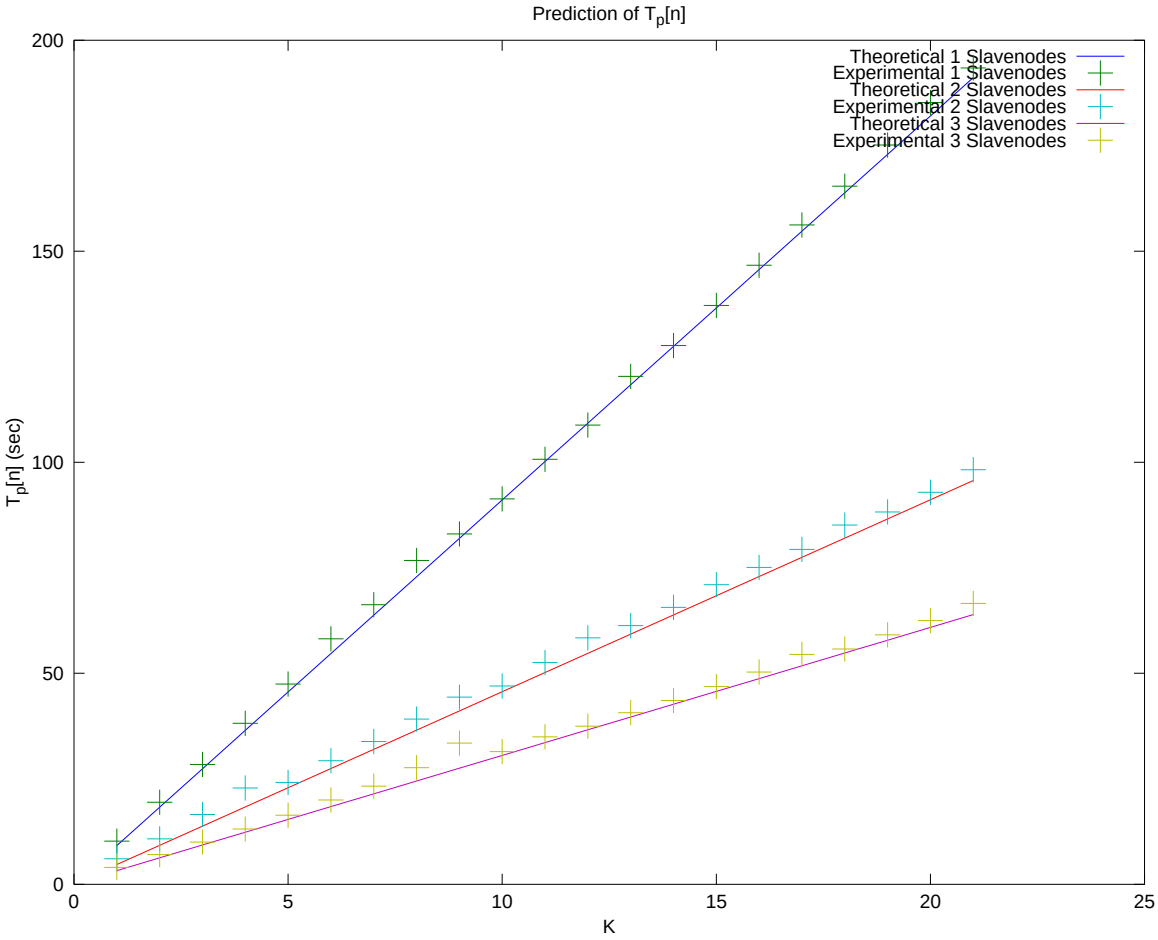

Figure 4.11: Theoretical validation of $T_p[n]$

1 Slave node : $T_p[1] = 9.1 \cdot \frac{K}{1} + 0.071 \cdot 1$

2 Slave nodes: $T_p[2] = 9.1 \cdot \frac{K}{2} + 0.071 \cdot 2$

3 Slave nodes: $T_p[3] = 9.1 \cdot \frac{K}{3} + 0.071 \cdot 3$

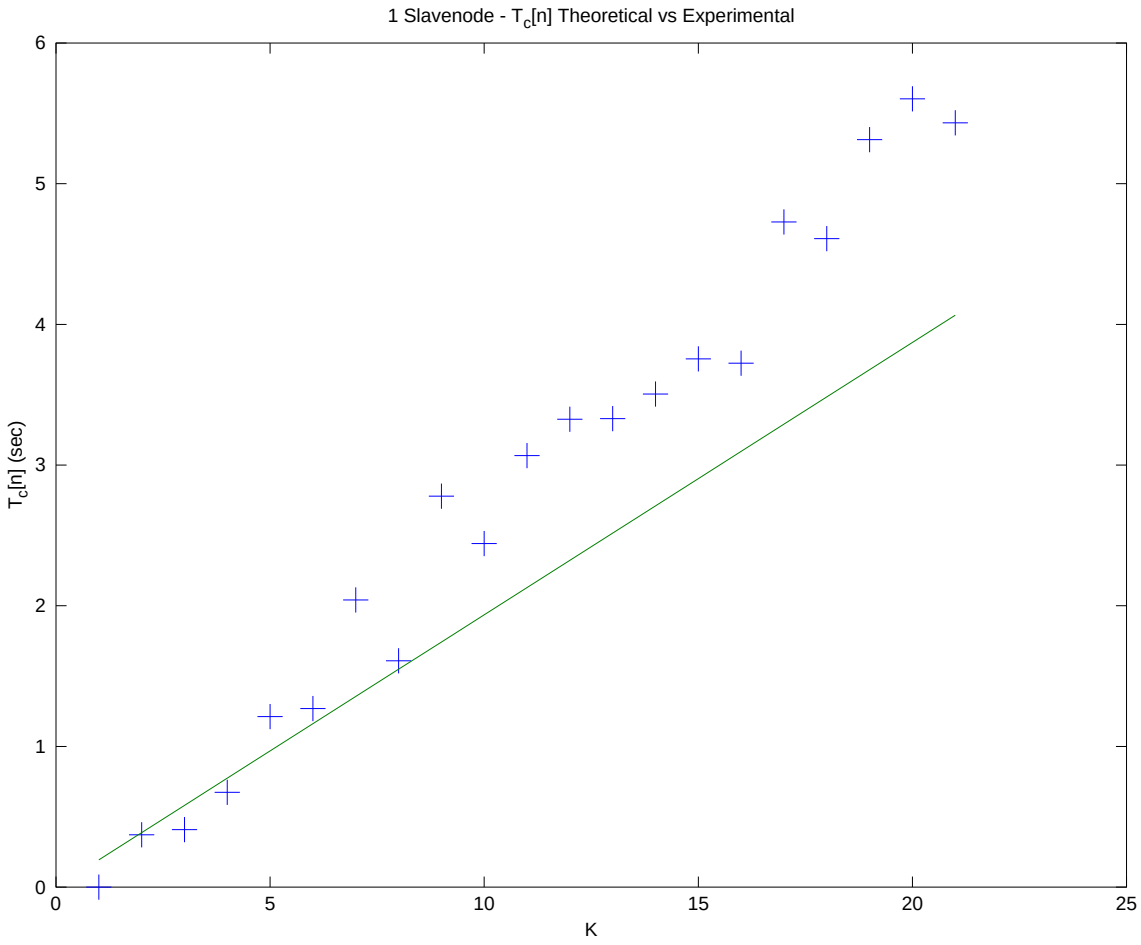

Figure 4.12: Theoretical validation of $T_c[n]$ for 1 Slave node

$$T_c[1] = 0.19 \cdot K$$

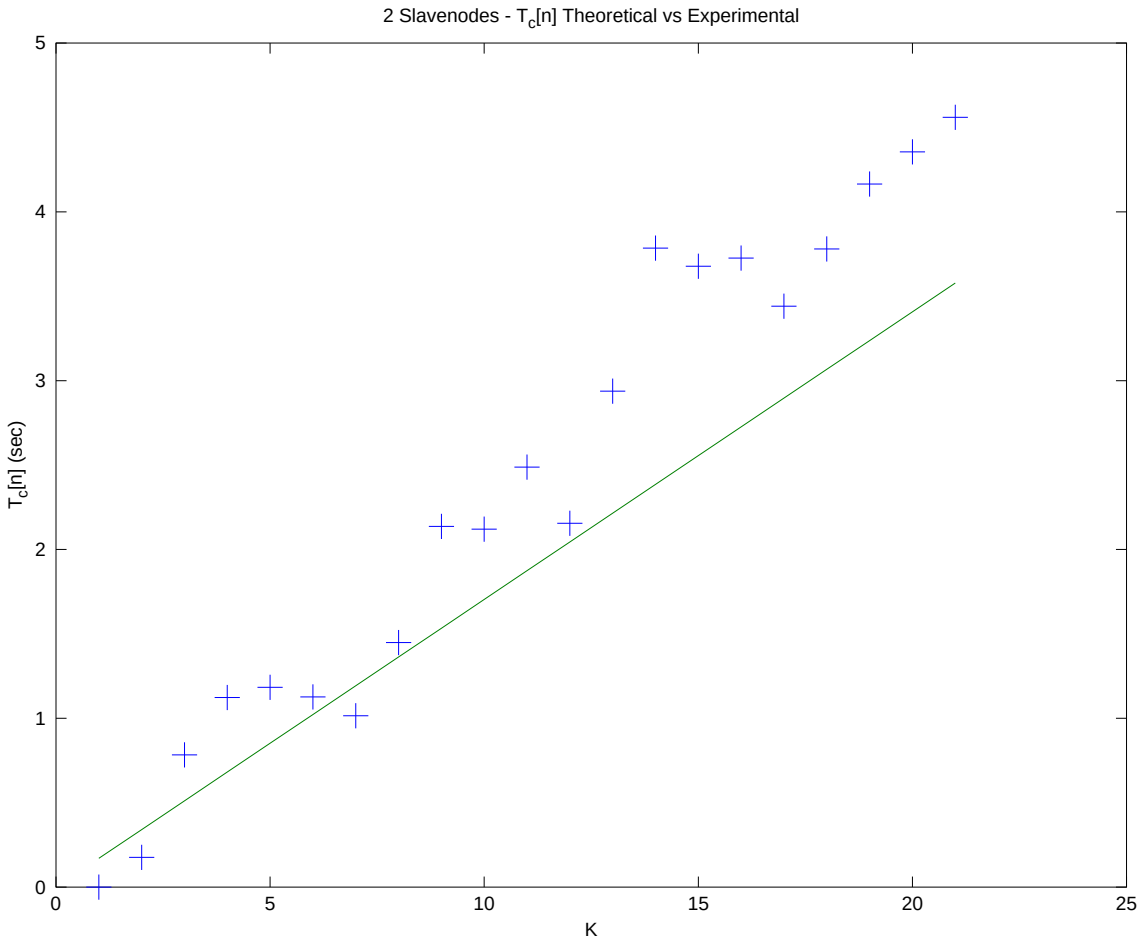

Figure 4.13: Theoretical validation of $T_c[n]$ for 2 Slave nodes

$$T_c[2] = 0.17 \cdot K$$

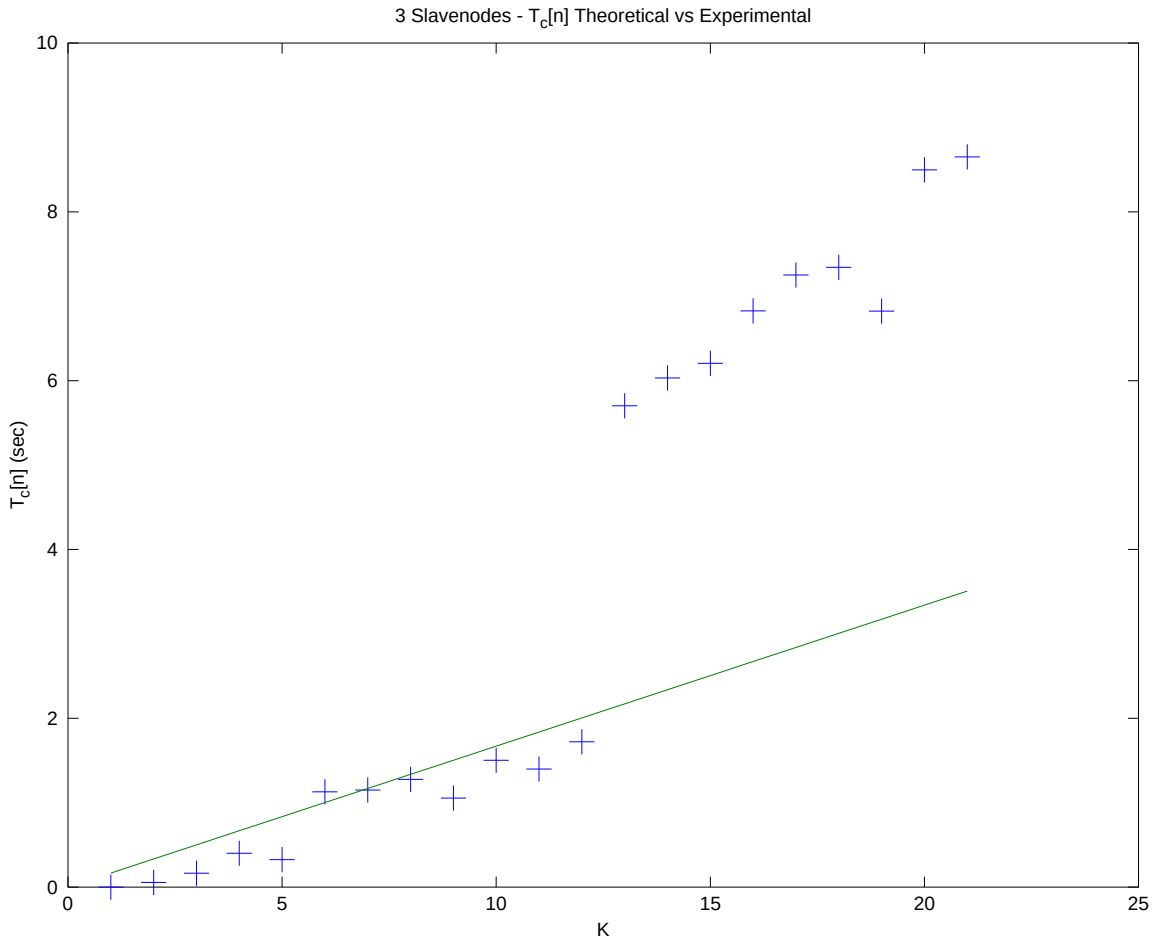

Figure 4.14: Theoretical validation of $T_c[n]$ for 3 Slave nodes

$$T_c[3] = 0.17 \cdot K$$

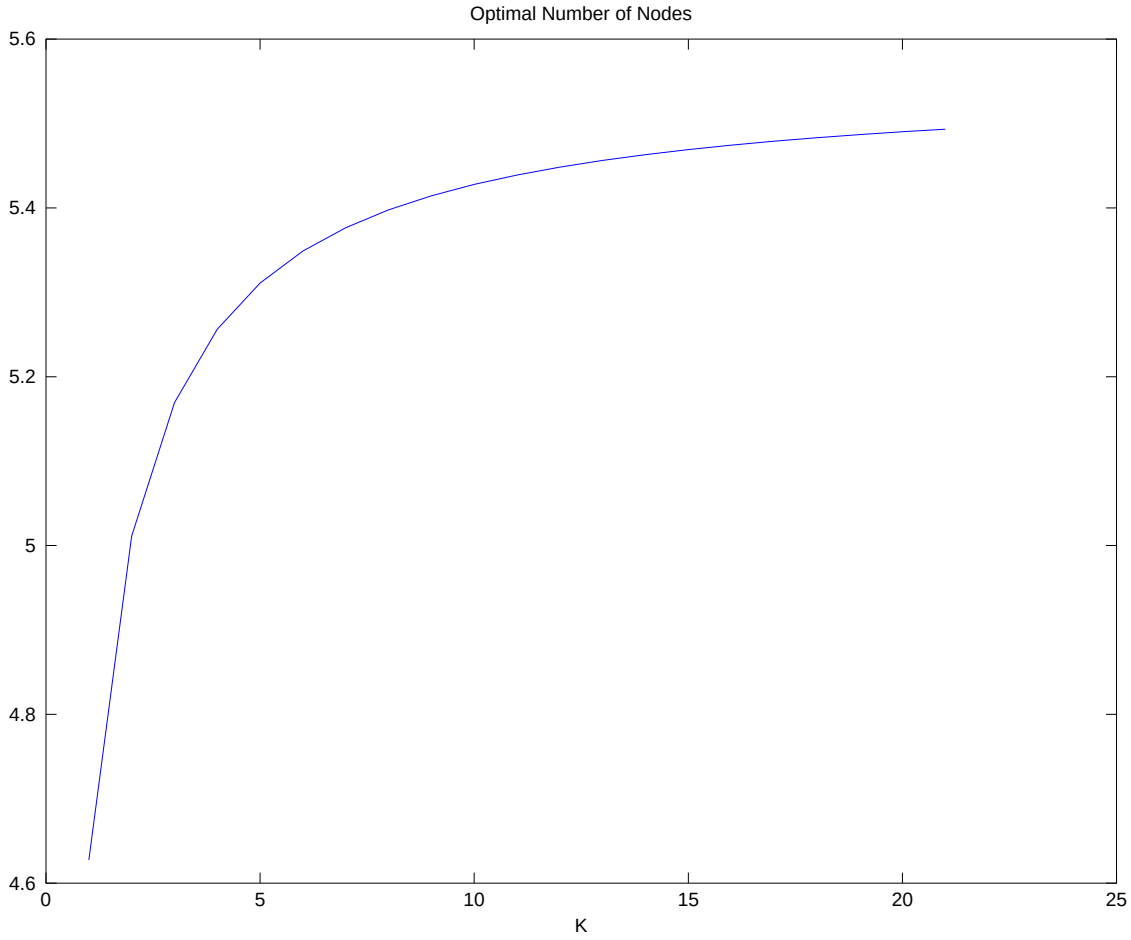

Figure 4.15: Optimal Number of Nodes with Experimental Setup

$$\alpha = 1$$

$$\beta = 0$$

$$\delta = 0.071$$

$$\epsilon = 0$$

$$e_x = 0$$

$$e_o = 0$$

$$t_x = 0.18$$

$$t_o = 0$$

Additionally, because the test setup is an Ad-Hoc WiFi network, $m_i = 1$. This reduces the equation even further, and the reduced form is given below in (4.4).

$$T_c[n] = t_x \sum_{i=1}^{n} k_i \tag{4.4}$$

Additionally, it is known that $\sum_{i=1}^{n} k_i = K$. Substituting, the final equation for $T_c[n]$ is shown in (4.5).

$$T_c[n] = t_x K \tag{4.5}$$

Equation (4.5) shows that $T_c[n]$ does not depend on n. Essentially, because K is a constant, (4.5) implies that t_x is independent of n. Table 3.1 also describes t_x as a constant. However, the measured values of t_x as a function of n is shown below in Table 4.1.

Table 4.1: Variance of t_x as a function of n

n	t_x
1	0.19358
2	0.17039
3	0.16704

This shows that t_x decreases as the number of nodes increases. Although the differences are small, the values in Table 4.1 for each value of n are 50-point averages. Therefore, there is a general decreasing trend for t_x as n increases. Figure 4.9 also conveys the same information, as it shows that the average sustained datarate increases for the master node as the number of slave nodes increases. The reason for this can be due to implementation of the webserver technology (JETTY) or the network stack implementation in the Android Operating System. Another reason is that mesh network efficiency increases as the number of nodes increases

[15]. It should be noted here that the mesh network efficiency does not increase linearly as n increases; there is a threshold for which the efficiency of the mesh network decreases due to MAC layer constraints.

It was experimentally determined that t_x is actually not a constant, but dependent upon n with the definition described above. It should be modeled as such to determine a more accurate model for resources that WDC requires, which will in turn provide a more accurate prediction of the optimal number of nodes to use.

4.6 Interesting Observations

An interesting observation that was made with all the collected data was the decompression times for all the slave nodes. Figures 4.16, 4.17, and 4.18 show the large variability that is exhibited in the decompression of the file packages that each slavenode receives. The tasks involved in decompressing a file is inflating the ZIP archive and writing many files to the external storage device very quickly. As Figures 4.16, 4.17, and 4.18 show, this seems to be a highly variable process. This is primarily due to the fact that this operation requires writing files to disk at a high rate, and input/output (I/O) is inherently a slow and highly variable process. One remedy to this issue is to either buffer the decompressed files all in memory for processing. Because these files do not need to be persisted on the slave nodes, this is a viable option.

4.7 Scalability

To characterize the scalability of the WDC system, one must consider both the software scalability as well as the wireless network scalability. The Android software that was developed to manage the WDC tasking is scalable in that it allows easy integration of new wireless applications. If an App developer wants to integrate a new wireless application into

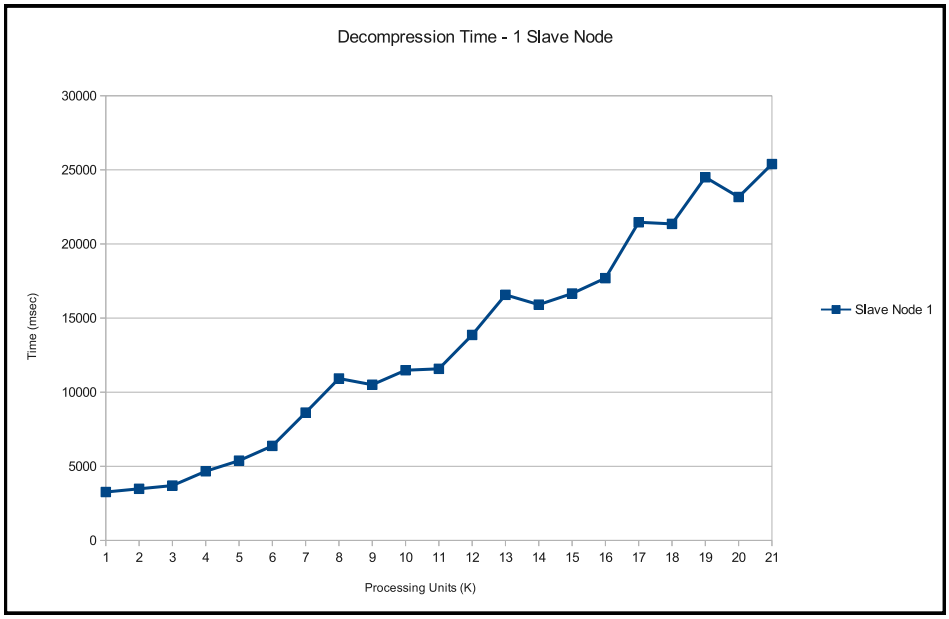

Figure 4.16: Decompression Times for each slave node in the 1 slave node case

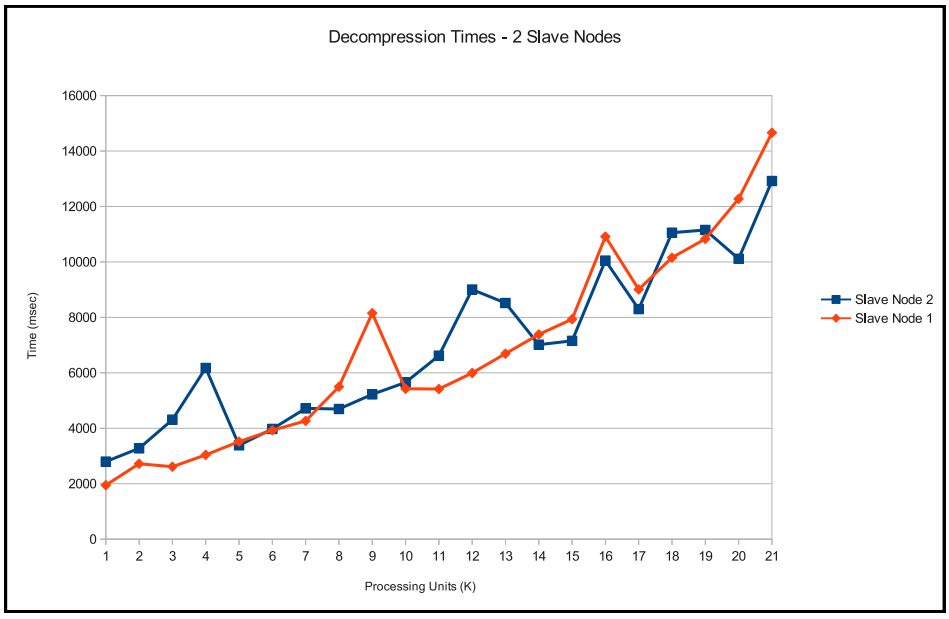

Figure 4.17: Decompression Times for each slave node in the 2 slave node case

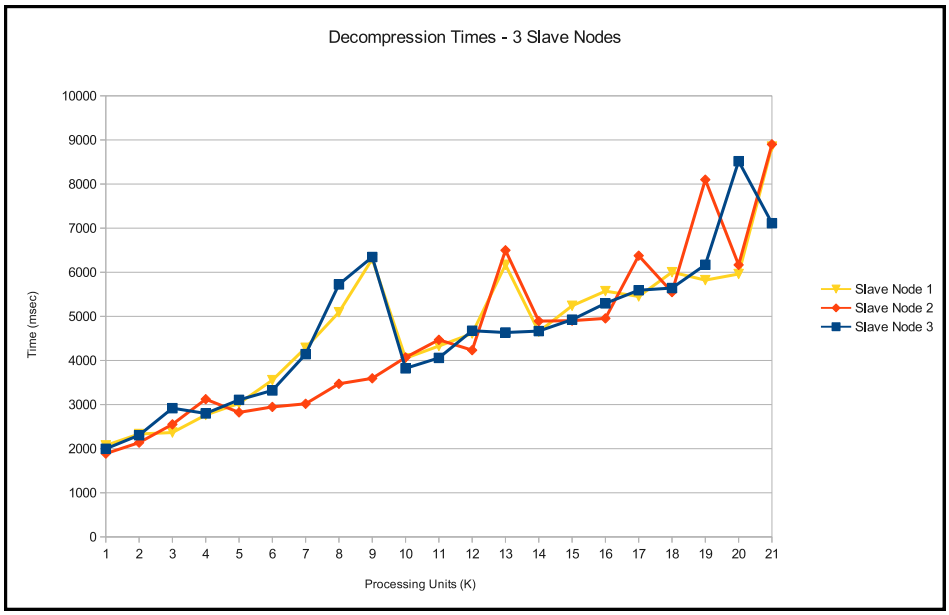

Figure 4.18: Decompression Times for each slave node in the 3 slave node case

the WDC software framework, it is easy to do so as long as an agreed upon intent tag is specified and used among all the nodes wanting to process the data in this way. Additionally, messaging between nodes is easily extensible, due to the fact that XML messages are passed between nodes. XML, by definition, is an extensible language and using it for messaging allows for future capabilities and scalability.

Scalability from a wireless networking perspective is a different issue. As the number of nodes in an Ad-Hoc WiFi mesh network increases, it is seen from the results that the total data throughput of the wireless network decreases. This is due to contention in the MAC layer of the WiFi specification. As the number of nodes in the Ad-Hoc network increases, each node's random contention will conflict at an increasing rate. Therefore, from a wireless perspective, the system is not highly scalable. One way to mediate this issue is to potentially have groups of nodes, all on one of the three non-overlapping 802.11 wireless channels, and use out-of-band signaling to send control messages with another communication mechanism such as Bluetooth.

Chapter 5

Conclusions and Future Work

5.1 Summary

The explosion of Android devices over the last couple of years has brought new opportunity to the mobile market. Modern smartphones are not only as powerful as desktop PC's a few years ago, but they are unique in that there has never been a device in history that has so many electronic components integrated into it for consumer usage. The goal of this thesis was to explore the usage of smartphones for wireless distributed computing (WDC). The theoretical foundations of wireless distributed computing, mesh networking in the Android platform, and implementation of a management engine to perform WDC were all topics explored in this research.

The theoretical foundations focused on proving that there is an optimal number of nodes to use in a WDC calculation, based on certain parameters and constants. The implementation focused on software that makes WDC possible. The software includes the mesh network implementation, as well as the WDC App which manages the WDC tasks. The software was then loaded onto four available smartphones running the Android operating system. It was shown that it is possible to perform wireless distributed computing using the on-board WiFi

chipsets of Android smartphones. Additionally, performance testing showed that WDC was a viable solution for increasing performance for performing a computationally intensive task.

5.2 Recommendations for Future Work

The work that was performed for this thesis has only begun to explore the implications of distributed computing over mesh networks using Android enabled devices. This work can be extended in many ways in order to bring WDC to being a reality. Significant research needs to be made into how to use an unreliable WiFi network for distributed computing. As the file transfer time Figure 4.6 show, there is variability in file transfer time due to the varying communications channel, even in the relatively fixed environment in which these experiments were conducted. In order to make WDC a viable option in the field, ways to use the WiFi mesh network as a backbone for WDC in variable communications channels must be researched.

Another recommendation for future work is to theoretically analyze the usefulness of WDC in an environment where nodes have to perform more inter-node communication for collaborative problem solving. The current WDC problem set explores the usefulness of WDC in an environment where each node in the WDC system performs large calculations independently and returns the results to the main node. Essentially, each node in the WDC mesh network can be considered a core on a multi-core processor. However, the real power of distributed computing is harnessed when multiple nodes in the network use their unique location to provide an extra dimension of information that can be used for solving problems and developing new intelligence. The current WDC architecture was not designed with this goal in mind, and research into how to better design the WDC architecture to support this would be useful.

Additionally, the research conducted in the paper "The Impact of Channel Variations on Wireless Distributed Computing Networks" could be extended and applied to the mesh

implementation that was used here (802.11 WLAN) [8] [10]. The results of that paper to enable the optimizer to do uneven distribution of work to gain more efficiency of the system could be applied to the work performed in this thesis.

Finally, research into how to build an optimization algorithm that accounts for varying conditions could be useful in realistic scenarios. The current optimizer takes a snapshot of the environment, and performs optimization based on that static information. A probabilistic model could be used in order to better predict how the varying wireless channel would affect the optimization results, and then that information taken into account to perform better optimization.

Bibliography

[1] android-wifi-tether. http://code.google.com/p/android-wifi-tether/.

[2] JavaCV. http://code.google.com/p/javacv/.

[3] wpa_supplicant_6_PATCH. https://github.com/KFire-Android/wpa_supplicant_6.

[4] Top Trends in Smartphones. White Paper, Motorola Corporation - Rhomobile, May 2011. Available Online.

[5] GSM Arena. Apple iPhone 4S - Full phone specifications. http://www.gsmarena.com/apple_iphone_4s-4212.php.

[6] GSM Arena. Samsung I9300 Galaxy S III. http://www.gsmarena.com/samsung_i9300_galaxy_s_iii-4238.php.

[7] Xuetao Chen, S.M. Hasan, T. Bose, and J.H. Reed. Cross-layer resource allocation for wireless distributed computing networks. In *Radio and Wireless Symposium (RWS), 2010 IEEE*, pages 605 –608, jan. 2010.

[8] Xuetao Chen, T.R. Newman, D. Datla, T. Bose, and J.H. Reed. The impact of channel variations on wireless distributed computing networks. In *Global Telecommunications Conference, 2009. GLOBECOM 2009. IEEE*, pages 1 –6, 30 2009-dec. 4 2009.

[9] D. Datla, X. Chen, T. Tsou, S. Raghunandan, S.M.S. Hasan, J.H. Reed, C.B. Dietrich, T. Bose, B. Fette, and J. Kim. Wireless distributed computing: a survey of research challenges. *Communications Magazine, IEEE*, 50(1):144 –152, january 2012.

[10] S. Dhakal, M.M. Hayat, M. Elyas, J. Ghanein, and C. T Abdallah. Load balancing in distributed computing over wireless lan: effects of network delay. In *Wireless Communications and Networking Conference, 2005 IEEE*, volume 3, pages 1755 – 1760 Vol. 3, march 2005.

[11] Google. Android Developers. `http://developer.android.com/index.html`, July 2012.

[12] G.R. Hiertz. IEEE 802.11s: The WLAN Mesh Standard. *Wireless Communications, IEEE*, 17(1):104–111, February 2010.

[13] Haitham Hindi. A Tutorial on Convex Optimization. Available Online.

[14] Tomasz Keller and Jozef Modelski. Experimental results of testing interferences in 2.4 ghz ism band. In *Microwave Conference, 2003. 33rd European*, pages 1043 –1046, oct. 2003.

[15] Jinyang Li, Charles Blake, Douglas S.J. De Couto, Hu Imm Lee, and Robert Morris. Capacity of ad hoc wireless networks. In *Proceedings of the 7th annual international conference on Mobile computing and networking*, MobiCom '01, pages 61–69, New York, NY, USA, 2001. ACM.

[16] Mila Mrsevic. Convexity of the inverse function. *The Teaching of Mathematics*, 11(1):21–24, 2008.

[17] Nielsen. More US Consumers Choosing Smartphones as Apple Closes the Gap on Android. `http://blog.nielsen.com/nielsenwire/consumer/more-us-consumers-choosing-smartphones-as-apple-closes-the-gap-on-android/`, January 2012.

[18] Jin-A Park, Seung-Keun Park, Dong-Ho Kim, Pyung-Dong Cho, and Kyoung-Rok Cho. Experiments on radio interference between wireless lan and other radio devices on a 2.4

ghz ism band. In *Vehicular Technology Conference, 2003. VTC 2003-Spring. The 57th IEEE Semiannual*, volume 3, pages 1798 – 1801 vol.3, april 2003.

[19] John G. Proakis. *Digital Communications*. McGraw-Hill, 1995.

[20] Jeff Reed. Wireless Distributed Computing on Android Devices. White Paper.

[21] Janche Sang, C.-H.M. Lin, S. Varadaraja, and Mu-Cheng Wang. Distributed network computing over wireless links. In *Parallel and Distributed Systems, 1997. Proceedings., 1997 International Conference on*, pages 252 –259, dec 1997.

[22] John MacLaren Walsh. Convex Sets, Convex Functions, and some of their properties. University Lecture.