Spontaneous Mobile Cloud Computing
Final Year Dissertation

Ian Deed
id39@hw.ac.uk
Supervisor: Sven-Bodo Scholz

April 2015
Abstract

In this paper a framework for distributing computation amongst mobile devices on a local area network is considered. This framework is designated as the Mobile Device Cluster (MDC). This framework is constructed and open sourced for future development.

The motivations for the MDC are discussed in reference to existing cloud infrastructure and previous research in the field. The implementation of the framework is documented and its usage is described for its implementation in 3rd party applications. An implementation of the MDC is used to obtain sets of test results for performance analysis which is used to describe and advice on future directions for the software.
I, Ian Deed confirm that this work submitted for assessment is my own and is expressed in my own words. Any uses made within it of the works of other authors in any form (e.g., ideas, equations, figures, text, tables, programs) are properly acknowledged at any point of their use. A list of the references employed is included.
Contents

1 Introduction .................................................. 1
   1.1 Supporting Technologies ............................. 4

2 Mobile Computation Offloading ............................ 7
   2.1 Offloading Locally .................................. 9

3 Local Cluster Computing approach ....................... 11
   3.1 Project aims ....................................... 11
   3.2 Framework Structure ................................ 12
      3.2.1 System Requirements .......................... 14
      3.2.2 User Perspective ............................... 16
      3.2.3 WiFi Direct .................................... 18
      3.2.4 User system requirements ..................... 19
   3.3 Security .............................................. 20
   3.4 Comparative Approach ............................... 20

4 Experimental Work ........................................ 23
   4.1 Integral Pi Calculator ............................... 24
   4.2 Runtime Results .................................... 24
      4.2.1 Increasing Computation Size .................. 28
   4.3 Networking .......................................... 29
   4.4 Energy use .......................................... 32

5 Experimental Analysis ..................................... 34
   5.1 System Capabilities .................................. 34
   5.2 Limitations Of This Approach ....................... 36

6 Conclusions & Future Work ............................... 38
Chapter 1

Introduction

Computational offloading is increasingly more common in modern computing. [5] Offering parallelism, vast runtime improvements, and access to external resources: there is a growing trend for connecting our applications to the cloud. This is driven by a desire for simultaneous functionality improvements and cost reductions.

In mobile smart phone technology offloading to cloud resources such as Amazon’s EC2 infrastructure has shown to decrease runtime and increase the battery life of smart phone handsets.[6] The argument for introducing offloading capability for certain classes of mobile application is compelling; despite incurring notable overheads that comes with the networking operations required.

Chapter 2 details current research and methods employed in using cloud computing to augment mobile devices.

The aim of this paper is to investigate the application and benefits of bringing resources closer to the offloader. For this purpose, resources are brought into a local networking context for more energy and time efficient use. This network model has been discussed in detail as having strong potential, particularly in areas of poor connectivity and restricted access to computing resources. This is an emerging area of interest and as such is subject to parallel avenues of research.

In this paper the local area computing network described is defined as the mobile device cluster (MDC). This refers to a set of one or more devices connected to a single host using a peer-to-peer local WiFi connection (fig 1.1). Such a network was examined in Making the Case For Computational Offloading in Mobile Device Clouds [8]. The intent of this paper is to build on the experimental analysis employed in the research shown to develop a
practical, open source framework for importing and managing a local device cluster within any android application.

The goal of this framework should be to effect potential runtime and energy use advantages that are available for local computation offloading [8] and enable them to be deployed in real world applications. As such we may define the framework as a successful implementation where we can show a reduction in program runtime and/or reduction or offload in energy use when compared to running the same algorithm in a sequential manner using only the host device.

Such a system is implemented as a first step and evaluation towards building a working framework that can facilitate distributing computation amongst mobile devices.

Chapter 3 details the construction and use of the mobile device cluster.

So that the framework implementation may be evaluated against the performance metrics defined it is applied in the construction of problems often referred to as embarrassingly parallel. By showing that runtime performance improvements can be realised, it can be determined that the potential for performance gains in real-world applications exists. This also affords the opportunity to profile the framework in use with Android Studio native profiling tools and runtime tests.
Chapter 4 outlines the methodology and results of experimentation done to assess the MDC framework.

In order to assess the quality and usability of our framework we are required to consider two primary users: The framework user and the end user.

The framework user, or programmer, is defined as the individual that is utilising the MDC framework within their own application to meet their cloud computing application specification.

The goal of the programmer is to create a suitably useful, responsive, and engaging application that meets the needs of the end user. The end user is defined as the user that will interact with the final application that the programmer creates.

The experience of the end user is entirely dependant on the application design of the programmer. It is the role of the MDC framework to provide an interface that is easy to use and flexible to help the programmer meet the needs of the end user.

Chapter 5 takes a look at the results obtained in chapter 4 to draw conclusions about the advantages and disadvantages of the MDC framework. From here it can be determined what kind of operations the framework is suited for.

Results obtained from experiments that were run on applications created using the framework show significant runtime improvements over executing code on a single device. Experimental results further validate the mobile local distribution approach by demonstrating clear runtime performance improvements.

The resulting output of this investigation has been to build a modular, maintainable framework that can be open sourced for future improvements and adaptation. This approach has also been validated by the experimental results showing that potential system gains have been realised in its implementation.

Chapter 6 assesses the relative merits of the project and looks at directions that the project will take in future iterations.

A comparative approach is taken in Spontaneous Mobile Cloud App [11] targeting iOS devices. This line of research, carried out in parallel with the work presented here, continues to show potential for the use of the technology.
The number of recent papers in the subject and parallel lines of research indicates that the field is a growing area of interest. This interest is increasingly likely to grow as more and more demands are made of mobile devices in the future.

1.1 Supporting Technologies

The MDC framework has been constructed using the Android API and associated construction and profiling tools. This makes it possible to deploy a single application to target any device, regardless of hardware, that runs using the Android operating system.

This operating system was chosen as it holds the majority market share in operating systems targeted towards mobile phones, tablet computers, games consoles, and more recently wearable and automotive platforms. It therefore offers the largest potential pool of local devices to offload computation to in comparison with similar platforms.

The Android OS is a layered design, using the Linux kernel version 2.6 [12]. In this model mobile applications are constructed on top of an application framework layer which provides UI views, data content providers, and resource managers along with other services for the construction of an application. Figure 1.2 shows a diagram of the layered structure of the Android OS.

In this structure, device drivers are written to interface with the Linux kernel, abstracting driver concerns from the application layers of the operating system. By separating these concerns away from the programmer Android applications can be compiled once to function across any number of supported devices.

Android applications are centred around a default class defined in the documentation as a main activity. The main activity acts as an entry point into the application, handling core logic, delegating to subsequent activities as well as defining behaviour for pausing, resuming, and exiting the application.

The Java code written in the application and framework layers is executed in an Android native virtual machine designed for use in mobile devices [12].
Figure 1.2: Diagram view of the layered structure of the Android operating system
The purpose of the MDC framework is to augment and extend the native application framework layer (see fig 1.2) to provide the resources and programmer interface required for computation offloading.

The approach to mobile computation offloading taken here has been made available by the emergence of new technologies in mobile device networking, namely Wi-Fi direct.

Since 2010 WiFi enabled Android devices have shipped with WiFi direct technology [1]. This facilitates peer-to-peer connection of two WiFi enabled devices without requiring a static access point. With this we can provide the user with the flexibility and minimal set-up time to create a temporary and modestly sized local network on an as needed basis.

Without use of WiFi direct it is likely that the peer-to-peer networking essential to the MDC framework would be too time consuming, impractical and present an infeasible number of security limitations to make constructing the system possible.

With this technology it is possible to spontaneously construct a local mobile network using a social or professional gathering as the basis for a collection of devices.

Another key technology in the design of the MDC framework is Java’s Callable interface. This allows for instances of the class to be transferred to and executed on another thread, or device. From the execution of this instance a result object is returned to the parent thread where it is stored until transmission to the network host is available.

In combination these modern technologies make the basis of the MDC framework design. This can be seen as a novel system with the potential to open new avenues in mobile computing.
Chapter 2

Mobile Computation Offloading

Mobile computing devices have experienced an enormous growth in features and performance as they have grown in popularity and use. Despite the growth in most hardware resources and software availability; battery power has seen a drastically shallower rate of growth [7]. As a result energy consumption represents a significant bottleneck in the computing power of modern mobile devices.

One solution to this problem has been to offload elements of the computation to be done to commercially available cloud services. This approach allows relatively resource-poor devices to tap into an vast infrastructure to delegate code execution.

There is currently a modern wave of mobile device applications that utilise cloud computation to perform tasks that would otherwise be too computationally expensive. Examples of this include: text, speech, and facial recognition.

Apple’s Siri for example makes use of natural language processing using cloud services to return queries posed by the end user. [4] Similar types of system have been developed by Google (now) and Microsoft (Cortana) that use computation offloading to perform the ”heavy lifting” involved in natural language processing to power voice activated searching and assistance.

Cloud computing in this context is an ideal fit as the size of the data to be computed on is relatively low (a short, compressible audio clip), resulting in short transmission time to the cloud. At the server side of the application, enough compute power can be brought to use to parallelize the task at
almost any level of granularity. The result, generally a small set of links or short sentence can be returned to the device at minimal networking cost.

In the general case cloud computing can be hugely advantageous in reducing time and energy costs, particularly in computationally intensive and parallelizable tasks. [10] The burden placed on the application creator is to employ offloading at the correct time, and in the correct manner to ensure the runtime and energy advantages conveyed justifies the overhead costs of transmitting the task over the network.

Another consideration when designing an application whose resources are based in the cloud concerns the financial cost of resources being used on the server.

Many companies offer the computational resources available at their disposal in various pricing models. Most commonly customers choose to charge developers either: by number of hours computation used, by number of times resources accessed, or utilising a 'free at point of use' model featuring user data as a commodity.

This cost consideration is a departure from the standard way that application programmers think about computing, which generally concerns energy and time as overheads.

Somewhat uniquely to mobile devices, there is a large amount of variability when considering the cost of network transmission times. This is a result of the diverse array of networking technologies each mobile device has available to it. The three primary technologies generally considered in mobile device networking have competing and complementary attributes.

- **3G**
  Mobile technology ubiquitous in most reasonably densely populated areas offering relatively constant online access. Offers slow transfer times by modern standards.

- **LTE**
  Also known as '4G', a mobile technology relatively common amongst larger cities. Offers much improved transfer speeds and lower rate of power consumption over 3G, but not as commonly available in less densely populated environments.

- **Wi-Fi**
  Offers great network transfer speed and comparatively low power consumption. Only usable when a private personal or business network
is made available to the user.

As well as network speed and power usage varying with the best technology available to the device, standard wireless networking issues such as connection strength, interference, and physical obstruction occur in use. These issues paint a more complex networking picture for everyday use in all technologies.

It has been observed through experiment that LTE consumes 1.62 times the power than Wi-Fi on downlink and 2.53 times more on data uplink. 3G performs worst of all, consuming 20.16 times as much energy than Wi-Fi on downlink and 34.77 times as much on uplink.[9]

A similar picture is shown for network transfer times. 3G connections get an average 6.1Mbps throughout the UK, while 4G averages 15.1 Mbps in comparison. [13] Wi-Fi benefits from being a short range technology and as such is used for communications within a smaller area, thus sustaining speeds of 40Mbps+ even on older networking hardware.

Wi-Fi experiences its own, unique limitations when connecting to the cloud. As Wi-Fi technology is reliant upon the private network of a business, institution, or residential location it is therefore subject to the limitations of that particular network. Research shows that the average broadband connection in the UK receives just 17.8Mbit/s download and 2.1Mbits/s upload.[14]

Despite networking limitations, computation offloading in mobile devices is a growing trend in modern mobile computing. Cloud computing resources have the potential to compensate for adding network overhead by making use of the extensive resources available to lower the computation time of the task.

In the average case for most systems that utilise computation offloading the end user is unaware that the majority of their applications workload is being done off the device.

2.1 Offloading Locally

As has been shown, offloading to the cloud can provide resources and energy above what is available on the client device. Also shown is that networking technology as well as server cost provides a reasonably significant bottleneck in current offloading technology.
Much of the drawbacks discussed for mobile computation offloading can not be completely alleviated within the technology currently available. Applying more resources can generally only be advantageous on a case by case basis. Additionally issues shown with cloud offloading are: adding extra resources is at odds with lowering the financial burden involved in offloading computation, in the short term at least fixed line internet speeds are slow to improve enough to show reasonable improvements in runtime speeds.

The solution investigated here is to localise computation offloading in a local area network. By bringing external resources into a private network we can shift responsibility for network cost, speed, and resource distribution to a locally available network owner.

To enable this approach resources for computing should have a high degree of availability while being easy to manage and affordable enough that they can be reasonably expected to compete in their own domain against commercially available cloud solutions.

By networking mobile devices in a private network we can build a small, manageable, local computation cloud. Such a system would be capable of maintaining network speeds well in excess of the average UK residential broadband speeds today. The proposed system would split computation in a democratised fashion amongst devices belonging to end users negating concerns about financial cost.

This design has been shown to be capable of showing an increase in speed of 50% and decrease in energy usage by 26% from offloading computation in a local cloud structure through practical experimentation. [8]

From this it can be determined that there is a great deal of value in offering the programmer the opportunity to offload to devices in a local cluster. If we can provide a straightforward and practical framework to do so then we may provide a valuable problem solving tool to unlock new capabilities in the mobile application market.
Chapter 3

Local Cluster Computing approach

The MDC framework is a system for distributed computing on mobile devices using an asynchronous message passing architecture. Code is written by the user using standard Android based Java. Parallel operations are written using the Java Callable for splitting computation.

When implemented in a user program the framework will allow the deployment of a local network with a user button or application event.

An MDC network is made up of a central "host" device that initiates its own network creation. This device acts as an access point for its own network, allowing secondary devices to connect to offer computation resources. Devices connected in this fashion to a host are referred to as "clients".

While network terminology used borrows from conventional Wi-Fi paradigms, the technology employed allows a new way of thinking about Wi-Fi networking. The notion of an access point for example is in the peer-to-peer model entirely implemented in software. This allows roles within the network to be dynamically allocated and potentially rapidly switched between devices to efficiently manage the network.

3.1 Project aims

The aim of the project is to create an open source framework for computation offloading. This should enable the programmer to design a system that will make use of the compute power of nearby Android devices.

Research has shown that it is possible to distribute computation amongst
networked Android devices and obtain improvements in runtime and energy use. [8] This research relies on matrix multiplication computations that are hard-coded into the system, representing a single potential use-case. The aim of the MDC framework is to extend this concept of local offloading into a generalised system that can be used by any computation that can be sufficiently parallelised.

The primary goal of the project is to demonstrate reasonable improvement in performance, as previously defined for parallel programs in real world networking scenarios.

Secondary goals also include: aiming to require minimal re-writing or disruption to the programmers Android application to unlock MDC potential. To aid this the framework is targeted to compliment existing technologies and programming patterns by extending applications in a minimally invasive way. This is achieved by making as much use as possible of existing Java technologies for parallelisation.

### 3.2 Framework Structure

The cluster manager framework is a lightweight, modular application structure requiring less than 100KB of application space. This allows for it to be dropped into any modern Android application to provide mobile device cluster functionality with minimal addition or modification to the core application.

MDC is designed around a distributed memory model of computation. Each device includes its own local memory and resources. Due to this data to be computed on must be distributed to each device on the network through the Wi-Fi direct channel created between devices.

Data transfer between devices is done in the user application by sending tasks to the framework for distribution handling. Management of data between devices is handled autonomously by framework components.

The primary components, known as managers, are built using the Android service components. An Android service is a component designed to work without user interface, running instead in the background on the main thread. In this respect it is well suited to co-ordinate the MDC. [3].

Figure 3.1 Shows the way in which the three manager services are structured and their relation to the user created activity. Additional classes used
The MDC is managed and interfaced with by the ClusterManagerService class. This is the only class with which the user interacts to access the framework. The system has been designed in this way to ensure the users main activity requires binding with only one service, handling network setup, system use, and the sending/receiving of tasks and results. This simplifies setup and helps to reduce the burden placed on the programmer.

The Network Manager Service is the single largest framework component, it is responsible for autonomously managing the local network. Because the Android service runs on the main thread, we wish all the components making use of network activity to have their own, independent threads that will resist halting the main activity or user interface while network events occur. Figure 3.1 shows the private classes of the network manager that execute their own threads.

Host:

- Server Thread
  Starts when a network is launched and continues for the duration that the cluster is required. Listens on a port for a device to join and request client status by transmitting their return IP Address.

- Send Task Thread
Transmits a Java callable to the host address supplied by the network manager.

- Receive Results Thread
  Listens for results being transmitted back along with the address of their parent client and returns the value(s) to the network manager.

Client:

- Client Registration Thread
  Runs once when the device joins the cluster. Sends the devices IP address to the host for future addressing then dies.

- Computation Thread
  Launches after the client registration thread dies. Listens for tasks coming from the host, passes the task to the devices callable manager and returns the result to the host when finished its computation.

The callable manager service is the execution point for all callable objects in the system. This service exists to provide individual device feedback. This module of the MDC is responsible for reporting the status, availability, and current load so that the host may make improved decisions about which device to send tasks to.

3.2.1 System Requirements

Before the beginning of the first iteration of system construction, functional and non-functional system requirements were drawn up. These serve to provide a method for prioritising the functionality during the construction phase of the project, and serve as a tool for monitoring the success or failure of project construction.

Initial requirements for the system focus on the feasibility and exploring the performance qualities of the implementation. These requirements are outlined in tables 3.1 & 3.2.
### Functional Requirements of the framework

<table>
<thead>
<tr>
<th>Item</th>
<th>Requirement</th>
<th>Priority</th>
</tr>
</thead>
<tbody>
<tr>
<td>FR 1.1</td>
<td>The system will provide an interface that allows a programmer to implement the execution of a distributed application across one or more networked Android devices.</td>
<td>Essential</td>
</tr>
<tr>
<td>FR 1.2</td>
<td>The system will manage distribution of tasks between any number of connected devices up to the network capacity without programmer input.</td>
<td>high</td>
</tr>
<tr>
<td>FR 1.3</td>
<td>When 0 devices are available to compute an impending task the system will revert to performing local execution without distribution or change to the task or results.</td>
<td>high</td>
</tr>
<tr>
<td>FR 2.1</td>
<td>The network for a device will be managed as autonomously as the technology allows, minimizing the interaction necessary with the application programmer.</td>
<td>medium</td>
</tr>
<tr>
<td>FR 3.1</td>
<td>The system must be capable of showing runtime performance gains equal to or faster than a task executed locally when distributed over a running cluster.</td>
<td>Essential</td>
</tr>
</tbody>
</table>

Table 3.1: Functional Requirements of the framework

### Non-Functional Requirements of the framework

<table>
<thead>
<tr>
<th>Item</th>
<th>Requirement</th>
<th>Priority</th>
</tr>
</thead>
<tbody>
<tr>
<td>NFR 1.1</td>
<td>The system will maintain a modular construction that allows for clear and straightforward maintenance and extension for potential future developers.</td>
<td>Essential</td>
</tr>
<tr>
<td>NFR 2.1</td>
<td>The framework will be released on an open source license that allows for the future development and use of the technology.</td>
<td>Essential</td>
</tr>
</tbody>
</table>

Table 3.2: Non-Functional Requirements of the framework
3.2.2 User Perspective

A crucial aspect in the system design is that the programmer can remain unaware of the technical details of the underlying architecture without affecting usability.

The user is required to make approximately 16 lines of prescribed additions (detailed in fig 3.2) to the main logic component of their program in order to make use of the framework.

Beyond these initial setup tasks, if desired, the programmer is only required to interact with the task send and result receive functions as defined in the framework documentation. A great deal more performance and tuning can be done by picking up further information about the network such as the number of devices currently available and their currently available compute power.

Key functions description from the documentation:

- `send(Callable task, int taskID)`
  
  Passes callable ‘task’ to the manager with integer id ‘taskID’ to be potentially executed remotely. Integer value is used to retrieve the results of the computation or check its status in the network.

- `receive(int ID)`
  
  Returns a Java object that contains the result corresponding to the integer ID that was assigned when the task was sent.

As a result, the task left for the programmer is to identify potential parallelism in a sequential program in a way that maximises the potential for speedup of the code. It is up to the programmer to make use of the Java Callable interface to manage communication by deciding which portions of the program are transferred to be done on remote client devices.

For Android applications already taking advantage of multi-core architectures the programmer would have likely already made the choice between thread, runnable, and/or callable. The semantic gap between runnable and callable is noticeably small enough that there should be little trouble translating between the two.

In this case the jump from a single device application to one executed on the cloud is very small. The majority of the task remaining is performance tuning to ensure that the largest amount of performance possible is obtained from the system.
Java threads should also provide relatively little challenge to re-write for use within the framework.

With a relatively light overhead on programmer time to invoke the framework, the primary challenge remains the decision where to switch to computing on the cluster or execute tasks locally on the device.

In an ideal scenario we wish to invoke cluster computation when we can be certain that the runtime and energy costs of local execution exceeds the overheads incurred by transmitting data across the network. Chapter 5 details the cost/benefit analysis of various real-world scenarios for mobile cluster computing.

Due to the dynamic nature of the MDC model the programmer can never be certain of the precise number of devices that will be available for computing within the network. It is indeed possible that the user application will attempt to execute as a cluster without any devices available at all.

In this scenario, or where all devices within the cluster are currently busy, tasks are executed locally by the host. In this instance the results of the computation are stored and retrieved in the same way as if they had been executed by any number of devices remotely.

As far as the framework user is concerned, they are able program to a single interface that will run regardless of current cluster conditions.

**User-System installation requirements**

In order for devices to participate in a cluster it is a requirement that, as a minimum, they have some version of the framework installed. This may be satisfied in a number of different ways.

The user may use the framework as part of the application that it is included with. This takes the form of a per-application approach where the end-user launches the framework as part of the interaction taking place in the parent application. In this scenario the end-user need not necessarily know that they are participating in a cluster and may only be concerned with the end result of their application processes.

Another scenario is that the user launches the framework as its own stand-alone application. The user here may wish to merely offer their devices resource for use to another device within the area.

In this scenario the framework merely serves the role of participating in
the computation of any application that wishes to connect to it. The code of
the parent application is simply sent to the device, executed on its processor
and the result returned without further investigation from the client.

There are no inherent restrictions on the framework implemented in one
application carrying out computations for its implementation in another. In
this way an application designer may want to take advantage of the computa-
tion indifference of the framework to exchange computation resources on
a one-for-one or token based system, according with the required design.

3.2.3 WiFi Direct

The system is made possible by, and is centred around, the Wi-Fi Direct
standard. The peer-to-peer networking functionality implemented as the
Android Wi-Fi P2P API provides a library of functions available in all An-
droid devices released since October 2011.[2]

Wi-Fi Direct makes up just 2 classes of the framework and are managed
exclusively through the network manager service. The core controlling logic
of the technology is written in a Broadcast Receiver class that is tasked with
listening to the network using the device Wi-Fi network interface card. This
class registers network changes such as devices becoming available or leaving
the network. These actions trigger branches within the framework that are
hard coded into the Network Manager Service to provide the most optimal
service possible for the given network conditions.

The vast majority of networking functionality within the framework is
done through Wi-Fi direct. However, due to it’s structure there does not
exist a way to directly handle addressing client devices from the host.

For this reason the framework is required to access the device network
card directly through the operating system. This is done to retrieve the
devices local network IP address for transmission to the host.

This transmission is done at the first point of connection between client
and host device, where the client transmits its own IP address to be stored
for future task sending by the host. Such a function is not required to be
replicated by the host as its IP address remains 192.168.49.1 across any Wi-
Fi direct network. As such the host may be addressed by the client with ease.
3.2.4 User system requirements

So that the user can drop the framework into their device there are some unavoidable minor additions that must be made to the user application main activity. Figure 3.2 shows the minimal main activity a user would be to required to create in order to implement the MDC framework.

The additions here that are required over the minimal Android main activity are:

- **clusterManagerIntent**
  
  Used to drive the launch of the cluster manager service that drives the rest of the MDC application, seen below in the startService() call.

- **ServiceConnection**
  
  Used to set up and maintain communication with the cluster manager service during the applications lifecycle.

Additional elements for layout are provided within the framework without implementation in the user activity. It was decided that the user should be left to decide on how devices and connections are displayed to the end user. This allows for a great deal of customisation of the network user interface, allowing it to be tailored to the application that it is being used for.

In order for the application to access the networking resources required 4 new permissions must be inserted into the android manifest if not already present for the parent activity. These are:

- **android.permission.ACCESS_WIFI_STATE**
  
  Allows the MDC framework to examine the current usage of WiFi networking devices to determine availability for use.

- **android.permission.CHANGE_WIFI_STATE**
  
  Grants the framework permission to change the network connection of the device to form or join a cluster.

- **android.permission.ACCESS_NETWORK_STATE**
  
  Grants permission for examining the additional device networking hardware to allow secondary connections to gather data outside of the cluster.

- **android.permission.INTERNET**
  
  Allows access to internet resources for either the current device or potentially additional devices on the cluster via the local network.
3.3 Security

The user should be naturally concerned with data security when using a wireless network to distribute information amongst a multi-user system.

Wi-Fi Direct uses WPA2-PSK encryption technology to encrypt traffic transmitted over its network. [1] WPA2 is a relatively modern encryption used in Wi-Fi networking, proven to be currently effective in ensuring secure transmission.

Due to the ease of joining the cluster network, it would be relatively trivial for an adversary to connect with the sole intention of gaining potentially confidential data. To this degree, encryption is only effective in protecting against an adversary collecting distributed data of all devices at once.

The programmer should take great care that sensitive, protected, or confidential data is not distributed to secondary devices within the network.

3.4 Comparative Approach

In Spontaneous Mobile Cloud App [11] Alisdair Law outlines a system to exploit distributed mobile computation on iOS devices. This approach makes use of networking technologies used here that have been adapted for the iOS environment.

This environment uses of a super-set of the C language known as objective-C. As a result the solution implemented supports programs implemented in C sub-sets of C, of which includes technologies designed for finding parallelism cost effectively.

The experimental approach taken here shows promising results when distributing inherently parallel tasks amongst iOS based mobile devices. The methodology used in construction of the task distribution system follows much of the same principles as the MDC framework; focusing on a set of modularly designed managers to distribute and run tasks over a local area peer-to-peer Wi-Fi network.

Due to security restrictions present in in the operating system used there are a number of challenges present in the execution of code on a device. The system is designed to only allow code that has a valid, signed certificate to be executed. This presents a significant obstacle to be overcome in a code distribution system.
Combining both device user bases in a single project would yield a large amount of value in potential computing device in a network; this however presents a number of very large technical challenges to be overcome.

In comparison to iOS, the Android application and framework layers are written using Java. Android does however allow the programmer to use C (or C++) when required. Such an instance may involve the need for additional performance and/or advanced memory management over that which Java can provide.

Making use of this capability, it is conceivable that such a system may take C code sent from an iOS device and execute it on an Android device. Performing the task in the inverse direction to execute Java code on iOS efficiently provides the additional challenge of requiring the implementation of a Java Virtual Machine on the remote device.

Enough challenges are present in both scenarios that it is unlikely to see performance gains comparable to that seen in homogeneous OS implementations. This nevertheless presents an interesting research opportunity. Such a prototype would likely raise interesting alternative inter-platform communication opportunities, regardless of the applicability to distributed computation.
public class MainActivity extends ActionBarActivity{
    private ClusterManagerService clusterManagerService;
    private boolean clusterManagerBound;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        Intent clusterManagerIntent = new Intent(
            this, ClusterManagerService.class);
        bindService(clusterManagerIntent, clusterManagerServiceConnection, Context.BIND_AUTO_CREATE);
        startService(new Intent(this, ClusterManagerService.class));
    }

    private ServiceConnection clusterManagerServiceConnection =
        new ServiceConnection() {
            @Override
            public void onServiceConnected(ComponentName name, IBinder service)
            {
                ClusterManagerService.LocalBinder binder =
                    (ClusterManagerService.LocalBinder) service;
                clusterManagerService = binder.getService();
                clusterManagerBound = true;
            }
            @Override
            public void onServiceDisconnected(ComponentName name)
            {
                clusterManagerBound = false;
            }
        };

    @Override
    public boolean onCreateOptionsMenu(Menu menu)
    {
        getMenuInflater().inflate(R.menu.menu_main, menu);
        return true;
    }

    @Override
    public boolean onOptionsItemSelected(MenuItem item)
    {
        int id = item.getItemId();
        if (id == R.id.action_settings) {
            return true;
        }
        return super.onOptionsItemSelected(item);
    }

    public void send(Callable c, int id){
        clusterManagerService.send(c, id);
    }

    public Object receive(int id){
        return clusterManagerService.receive(id);
    }
}

Figure 3.2: Minimal main activity in order to implement MDC
Chapter 4

Experimental Work

Experimental results were obtained by using a suite of 5 Nexus 7 (2013) tablet devices and a Motorola Moto G phone (2013) of lesser resources. All testing was done on devices that had no applications installed besides the factory defaults and MDC testing applications.

<table>
<thead>
<tr>
<th></th>
<th>Memory</th>
<th>CPU</th>
<th>Wi-Fi Technology</th>
<th>Battery</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nexus 7</td>
<td>2GB</td>
<td>Snapdragon S4 Pro, 1.5GHz</td>
<td>802.11 a/b/g/n</td>
<td>3950mAh</td>
</tr>
<tr>
<td>Moto G</td>
<td>1GB</td>
<td>Quad-core 1.2GHz Cortex-A7</td>
<td>802.11 b/g/n</td>
<td>2070mAh</td>
</tr>
</tbody>
</table>

In order to gather results about the performance and usability of the network a range of usage scenarios were considered. Each experiment was performed on the network separately with 0 through to 5 devices connected. Runtime information was gathered using the Log data generated at main activity level of the program.

Uniquely to the experimental set-up when 1 or more devices are connected to the host then code execution is delegated entirely to client devices, leaving the host to perform purely the role of network co-ordinator.

Network co-ordination here is performed exclusively by the Moto G device, using the tablet devices as clients.

Experimentation shows that the role of network co-ordinator takes up very little of even the limited resources of the Moto G device. Using the Android SDK profiler Figure 4.1 shows that with 4 devices connected as clients the co-ordinator remains more that 75% idle.
4.1 Integral Pi Calculator

Initial testing was carried out using the MDC framework to construct an application that calculates $\pi$ using integral calculation. This implementation was chosen as it provided computation that can increase workload and degree of parallelism in a simple manner.

Figure 4.2 shows the code whose instances are sent to the MDC manager for computation. These instances are managed by the PiJavaCalc class (figure 4.3). This class is responsible for taking the number of available devices from the MDC framework and distributing the calculation amongst the maximum number of devices available.

4.2 Runtime Results

Figure 4.4 shows the resultst obtained from offloading $\pi$ calculations to available devices within the network.

\[\text{Using local execution only on Moto G device}\]
\[\text{Using local execution only on Nexus 7 device}\]
class PIworker implements Callable, Serializable {
    private int myid;
    public PIworker(int id) {
        myid = id;
    }
    public Object call() {
        double d, s, x;
        d = 1.0/n;
        s = 0.0;
        for (int i=myid+1; i<=n; i+=numDevices) {
            x = (i - 0.5)*d;
            s += 4.0/(1.0 + x*x);
        }
        return (d*s);
    }
}

Figure 4.2: Pi worker class that is executed on each device

<table>
<thead>
<tr>
<th>Number Devices:</th>
<th>0^t</th>
<th>0^d</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>5 Million Steps:</td>
<td>439</td>
<td>269</td>
<td>471</td>
<td>520</td>
<td>521</td>
<td>693</td>
<td>681</td>
</tr>
<tr>
<td>20 Million Steps:</td>
<td>1590</td>
<td>1067</td>
<td>1031</td>
<td>990</td>
<td>913</td>
<td>844</td>
<td>840</td>
</tr>
<tr>
<td>40 Million Steps:</td>
<td>3166</td>
<td>2145</td>
<td>2402</td>
<td>1510</td>
<td>1113</td>
<td>1159</td>
<td>1050</td>
</tr>
</tbody>
</table>

Table 4.1: Integral pi calculator runtime Results
public PiJavaCalc(int numberSteps, MainActivity activity) {
    n = numberSteps;
    numDevices = clusterManager.numberAvailableDevices();
    Double sum[] = new Double[numDevices];

    for (int i=0; i<numDevices; i++) {
        Callable dev = new PIworker(i);
        // Call send on the callable,
        // gives an ID used to retrieve the result
        activity.send(dev, i);
    }

    for (int i=0; i<sum.length; i++) {
        // Receive results from each device
        // & sum for Pi approximations
        Object result = activity.receive(i);
        String res = String.valueOf(result);
        sum[i] = Double.parseDouble(res);
        pi = pi + sum[i];
    }
    Log.i("Pi calculation yields:", String.valueOf(pi));
}

Figure 4.3: Main Body of the π calculator responsible for sending and receiving callable objects
Initial testing with 5 million step calculations showed that the time taken to perform the calculation was too low to gain improvement once network overheads were added.

![Figure 4.4: Runtime results graph of offloading $\pi$ integral calculation to n devices](image)

Analysis of the log files from the operation showed that while gains were made on the time spend on the calculation itself (about 200 millisecond faster than local execution on the less powerful Moto G device), the time taken to setup and complete the network transfer processes means the entire runtime taken is a net loss over local sequential computation.

Testing the same application with an increased workload shows far more interesting results. At 20 million step computations we get an immediate and clear boost to the computation runtime that exceeds the overhead spent on networking. Despite still only a single device still doing work on the calculation. Suggestions for why this may be the case are discussed in section 5.

Beyond this initial increase a steady but small improvement in runtime
is shown as further devices are added to the cluster.

When the workload is increased to 40 million steps we continue to see runtime improvements beyond the addition of a single, more powerful device. In this size of calculation we continue to see a significant boost as we add secondary and tertiary devices to compute on.

4.2.1 Increasing Computation Size

Based on initial runtime results further testing was done on increased computation size to measure the performance achieved under heavy load.

Figure 4.2 shows the averaged runtimes of each test. Results were gathered in the same manner as initial testing to observe average network performance.

Table 4.2: Integral pi calculator Large calculation runtime Results

<table>
<thead>
<tr>
<th>Number Devices:</th>
<th>0(^1)</th>
<th>0(^2)</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>80 Million Steps:</td>
<td>6488</td>
<td>6334</td>
<td>7199</td>
<td>3803</td>
<td>2710</td>
<td>2020</td>
<td>1685</td>
</tr>
<tr>
<td>120 Million Steps:</td>
<td>9845</td>
<td>9512</td>
<td>6796</td>
<td>5520</td>
<td>3730</td>
<td>2932</td>
<td>2428</td>
</tr>
<tr>
<td>160 Million Steps:</td>
<td>12930</td>
<td>12669</td>
<td>8504</td>
<td>4753</td>
<td>4990</td>
<td>3749</td>
<td>3704</td>
</tr>
</tbody>
</table>

In such large calculations the performance improvement between local device execution on the Moto G versus the Nexus 7 was noticeably more modest than in earlier testing. Despite this, offloading to just one tablet device showed a reasonable speedup compared to the local execution runtime on either device.

The additional runtime performance gained appears to be largely a result of separating calculation operations from the user interface work done on the main thread of the host.

Figure 4.5 shows more complexity and greater variation in performance when computing over the MDC. Taking into account two scenarios that show a slight performance plateau, the general trend shows a steady performance increase as additional devices are added to the network.

Over all testing inputs, as a general rule it holds that as the amount of computation required grows the amount of performance unlocked by the

---

\(^1\)Using local execution only on Moto G device  
\(^2\)Using local execution only on Nexus 7 device
Figure 4.5: Runtime results graph of offloading large scale $\pi$ integral calculations

MDC framework increases.

Figure 4.6 shows a speedup graph plotted over the 5 largest testing inputs. This shows a runtime improvement in the system, scaling generally larger with higher inputs.

4.3 Networking

In the construction of a system that aims to speed up computation by utilising network distribution, the primary timing concern is the networking operations used to achieve task distribution.

A number of concerns arise when considering the network performance of task distribution. The largest among them is throughput: the ability of the network to carry data between nodes in a timely fashion. As has been
shown in chapter 2, current Wi-Fi technology is capable of high enough throughput to justify its use in the MDC.

A second concern that arises is due to the wireless nature and popular uptake of Wi-Fi technology. Due to these factors we may expect to see some variation in networking performance as a result of radio frequency interference. For this reason certain Android devices by manufacturer choose to limit the number of connections to Wi-Fi Direct devices to a maximum of 8.

During the course of testing no effects of interference were found on using a relatively low number of devices. This obstruction however, represents a hard limit for the size of cluster certain devices may support. For devices from other manufacturers it may be the case that adding devices above this number will become increasingly less likely to yield further speedup regardless of computation or ability of the algorithm to be executed in parallel.

During runtime testing it was found that the overhead time, regardless
Figure 4.7: Average Pi Calculation Time breakdown
of computation size, was consistently in the 500-600 millisecond range for small cluster networks (1-2 devices). This time may be considered the overhead cost of computing using the MDC. Figure 4.7 shows a computation independent breakdown of the time taken to reach important steps in the MDC calculation using 2 client devices.

As shown, the largest amount of time in the networking process is taken up by establishing socket connections to client devices. This also proved to also be the most variable element in the process, with connection times ranging from 81-430ms.

One of the advantages of using a TCP connection is guaranteed delivery and correctness of the data transferred. This saves small amount of resources in implementing checks ourselves to ensure correct data delivery. The penalty for these guarantees however is paid in the time taken to establish socket connection.

It can be confidently asserted that the TCP socket connection is acting as a bottleneck and as such, should be a primary target for seeking additional performance.

In seeking additional performance from this component we should examine more time effective ways to connect TCP sockets, consider solutions to gaining more use out of sockets such as: keeping them open for multiple operations, and consider alternative technologies for network transfer.

Experimental work should be done in examining the use of UDP for networking operations. This will include the relative merits of trading connection reliability against offset setup times. Much of this work is required already in order to investigate the use of broadcast functionality in the MDC system.

4.4 Energy use

When discussing computing on mobile devices, more than with alternate personal computing devices, significant attention must be paid to the amount of energy required.

Modern devices have shown to be power hungry as a result of the demands placed on their use. These devices are also strictly capped at the amount of energy available due to the limitations of currently available battery technology.
One of the most attractive elements to utilising the MDC would be to realise energy efficiency in computation offloading. Results shown in earlier studies describe a 26% reduction in energy use when computing using a similar structure. [8] This represents a goal for the MDC framework to achieve when computing over tasks allocated at the time of application writing.

Difficulties arise in measuring the energy efficiency performance in Android applications. Within the Android system there currently does not exist the tools required to measure energy consumption at a level accurate enough to measure individual app usage.

Relatively few solutions are currently available for monitoring energy efficiency of Android mobile devices. Current benchmarking tools exist as the result of intensive research into the topic [15], these offer excellent breakdown of CPU and network card power consumption by application. Unfortunately the software only reliably supports a very limited subset of hardware. The devices used in testing the implementation of the MDC framework were not officially supported by the software and results obtained from testing were too wildly variable and inaccurate to draw reliable conclusions.

Of the generic market solutions for measuring energy use, none could offer the detail and component breakdown required to determine accurately application energy performance.

Previous work in this field has focused on making use of hardware solutions to monitor the amount of energy drawn by application processes. [8][10][7] This approach appears to yield the most accurate and reliable results. This however is not possible within the current MDC project as the battery packs of each of the current testing suite devices are sealed into the device at point of manufacturer.

Initial testing with coarse tools available appears to show promising results in line with previous research by showing reduced power consumption by distributing computation. The tools used in obtaining these results however means firm conclusions can not be drawn. The only statement we can make with reasonable confidence is that energy performance is not drastically worsened by the distribution of tasks.

Future work will focus on utilising the MDC in hardware devices supported by [9] potentially alongside utilising their open source approach to utilise devices used for framework development. An accurate hardware measurement would also be highly desirable in examining framework performance with great detail.
Chapter 5

Experimental Analysis

Results of testing on the MDC framework show that there is clear and powerful performance gains to be found in distributing computation amongst local area mobile devices.

The largest potential gains are clearly to be found in offloading from computationally limited devices to devices with comparatively greater resources.

Experimental results show that the current system has network overhead of 500-600 milliseconds with a single device connected to perform computation. Additional devices being added to the network appears to add approximately 150 milliseconds overhead per device added.

Thus in any application seeking performance gains using MDC, the initial runtime overhead of the single device application should be at a minimum 600 milliseconds to begin to see performance improvement.

For applications with runtime overhead well in excess of this range, noticeable speedup can be found, where parallelism is possible, by using the MDC framework.

5.1 System Capabilities

One of the more surprising elements of the system, from experimental results, is its ability to gain runtime performance with a single client network configuration.

As shown in figure 4.6 modest speedup can be gained by using just a single device as a client. This is despite current framework configuration prohibiting the network host performing computation while acting as a point
of contact for secondary computing devices.

A runtime improvement is to be expected when offloading to devices with better resources. Though similar, though less noticeable runtime improvements were also observed when the client/host relationship was reversed between devices. This can be explained by both the additional resources brought by the tablet device acting as client and separation of concerns between UI interaction and heavy duty calculation.

This result opens new avenues in utilising the MDC framework. With a single client configuration it is quite feasible that a programmer may design a system that utilises just 2 or more devices belonging to the same user.

Such a system may utilise a more powerful device to do resource intensive work while a more portable device captures data and/or provide user interface functionality. An example may be real-time video encoding utilising a mobile phone and tablet computer.

As well as the runtime speed advantage, this setup has the advantage of not requiring the secondary device to use the substantial amount of energy required to power an energy intensive screen. Thus the secondary device is exposed to doing useful amounts of work while making the power savings of having a single point of contact with the user.

The intended configuration of the network, many clients per single host, remains the primary purpose of the framework and the one at which it delivers its greatest benefit. Particularly in large calculations the advantages of multiple client devices is clear from the experimental result shown.

In its present state the system is most likely to be used in either a multi-user game or media rendering context.

Multi-user games have the potential to draw users in to connect to the system. This can be done: either using task distribution to render complex, expensive in-game graphics elements that would not otherwise be possible, or using the game to draw in users a side function while calculations are done for the host in the background.

Media rendering has the need for large amounts of compute power and could fully utilise a system that brings down rendering time while being flexible enough to construct for a single purpose.

Further use cases such as extending desktop computation are targeted in iterations 2 and 3 in the software’s development. One of the main drivers
behind open sourcing the project is that relatively obscure use cases may begin to be supported by software developers implementing functionality into the project to match their own needs.

The license used to promote the project under (GNU V2) ensures that developers are free to use and modify the project as they wish. The requirements of this license are such that modifications and additions to the code should be made available to be combined into the main body of the code.

In this way it is planned that the code can continue to grow and evolve to support new use cases and seek further performance advancements.

5.2 Limitations Of This Approach

Within the MDC framework there are certain amounts of restrictions, both due to the design of the system and its early stage of development.

In the first category, the system is completely ambiguous in what it chooses to distribute and run. It is the role of the programmer making use of the framework to decide when and how to make use of the parallelism offered.

The system will naively accept any task to distribute over the network, sending it to be computed regardless of whether it will aid in the overall result. No prompts or errors are sent to tell the programmer that the way they are distributing tasks may be flawed or not return results.

Another limitation is: at this stage of development distributed processes are in themselves sequentially executed on the remote device. While the program as a whole is being executed in parallel each of its individual parts may only be executed on a single core of each device.

Modern mobile devices are very commonly multi-core machines. Thus it is reasonable to expect to be able to distribute a parallel computation for a device to execute utilising as much of its resources as possible.

By making use of more cores in each device we can unlock greater performance without exposing a great deal more network overhead.

Social considerations must also be taken into account. For clients on the network, the application is essentially requesting that they surrender CPU cycles and battery power to a host device that may or may not benefit them.
This may be addressed in a user application with incentives to the end user such as in-game tokens, or preferential position in a queue for the next host, depending on the application and context.
Chapter 6

Conclusions & Future Work

In this work mobile device offloading has been motivated for computationally intensive applications. Local area offloading has been motivated as an alternative to conventional and comparatively expensive use of cloud infrastructure.

In an idealised system it would be feasible to offload any given task to a similarly networked device for computation with the aim of improving performance of the system as a whole. This is unfortunately not a realistic possibility in current technology.

The approach taken in this paper is an attempt to find and explore some of the capabilities possible in local offloading with currently available technologies.

It has been shown that a generalised framework can be created to effectively and efficiently distribute computation tasks amongst mobile devices on a peer-to-peer network. Potential performance gains in hard-coded calculations have been realised in a generic framework for any computation.

The functional and non-functional system requirements laid out in section 3.2.1 have been fully realised within the system implementation. In addition to this, promising runtime performance increase has been shown to be possible in such a system with a strong potential for improvements in energy use.

In combination with parallel research carried out on the iOS device, the distributed mobile approach to seeking runtime and energy improvements has been validated. Both systems exist as a proof of concept to show that there exists the possibility to explore the topic further and continue to expect advancements in system performance.
Through this work an open source framework for local area mobile device cluster computing on the Android platform has been implemented and distributed. This framework allows a user to drop a small, simple set of components into a new or existing application to realise powerful mobile cluster computation.

By releasing the framework as an open source project the intent is to make available to software developers the underlying code that provides the MDC content. This enables tuning and optimisation that may otherwise have been unavailable. It is hoped that changes made to the framework to facilitate new use cases can be generalised and incorporated back into the main project. In this way new functionality can be added to continue the expanding of the framework for future use.

Many of the most critical updates that will address functionality and performance for the framework have been discussed. These additions are open to testing and development now that the core functionality of the proposed system has been implemented.

One of the primary goals for the next development cycle is to continue to extend the structure and definition of the MDC to include personal computers such as desktops and laptops as part of the local network.

This has the potential to allow vast speedup and resource sharing of a mobile device using a vastly more powerful machine when used as a client. The potential also exists to improve runtime performance of intensive applications using the PC as a host device.

Extended work is also required on system documentation. Extensive write-up and simple tutorials could help new users traverse the learning curve of a simple but completely unfamiliar system.

It is the hope that this and other like-minded research aids in opening new avenues in energy efficient offloading in the future.
Bibliography


Chapter 7

Appendices

1

Appendix A - Experimental Results

Table 1: Integral pi calculator runtime chart with 5,000,000 steps over n clustered devices

<table>
<thead>
<tr>
<th>Number Devices:</th>
<th>0(^1)</th>
<th>0(^2)</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Run:1</td>
<td>449</td>
<td>272</td>
<td>478</td>
<td>526</td>
<td>322</td>
<td>726</td>
<td>828</td>
</tr>
<tr>
<td>2</td>
<td>454</td>
<td>257</td>
<td>446</td>
<td>551</td>
<td>682</td>
<td>842</td>
<td>638</td>
</tr>
<tr>
<td>3</td>
<td>482</td>
<td>271</td>
<td>647</td>
<td>602</td>
<td>536</td>
<td>768</td>
<td>568</td>
</tr>
<tr>
<td>4</td>
<td>433</td>
<td>267</td>
<td>592</td>
<td>485</td>
<td>563</td>
<td>722</td>
<td>583</td>
</tr>
<tr>
<td>5</td>
<td>430</td>
<td>273</td>
<td>519</td>
<td>553</td>
<td>704</td>
<td>692</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>425</td>
<td>274</td>
<td>502</td>
<td>543</td>
<td>575</td>
<td>604</td>
<td>750</td>
</tr>
<tr>
<td>7</td>
<td>429</td>
<td>268</td>
<td>457</td>
<td>364</td>
<td>595</td>
<td>743</td>
<td>719</td>
</tr>
<tr>
<td>8</td>
<td>434</td>
<td>255</td>
<td>414</td>
<td>496</td>
<td>519</td>
<td>655</td>
<td>623</td>
</tr>
<tr>
<td>9</td>
<td>439</td>
<td>271</td>
<td>320</td>
<td>532</td>
<td>484</td>
<td>675</td>
<td>590</td>
</tr>
<tr>
<td>10</td>
<td>432</td>
<td>283</td>
<td>510</td>
<td>589</td>
<td>376</td>
<td>437</td>
<td>817</td>
</tr>
<tr>
<td>Average runtime:</td>
<td>439</td>
<td>269</td>
<td>471</td>
<td>520</td>
<td>521</td>
<td>693</td>
<td>681</td>
</tr>
</tbody>
</table>

\(^1\)Using local execution only on Moto G device
\(^2\)Using local execution only on Nexus 7 device
<table>
<thead>
<tr>
<th>Number Devices:</th>
<th>$0^1$</th>
<th>$0^2$</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Run:1</td>
<td>1574</td>
<td>1089</td>
<td>1339</td>
<td>1012</td>
<td>806</td>
<td>1059</td>
<td>853</td>
</tr>
<tr>
<td></td>
<td>1583</td>
<td>1033</td>
<td>1303</td>
<td>924</td>
<td>1150</td>
<td>779</td>
<td>1186</td>
</tr>
<tr>
<td></td>
<td>1609</td>
<td>1072</td>
<td>1256</td>
<td>1004</td>
<td>973</td>
<td>812</td>
<td>811</td>
</tr>
<tr>
<td></td>
<td>1582</td>
<td>1070</td>
<td>1256</td>
<td>1004</td>
<td>973</td>
<td>842</td>
<td>772</td>
</tr>
<tr>
<td></td>
<td>1602</td>
<td>1069</td>
<td>1294</td>
<td>1004</td>
<td>665</td>
<td>727</td>
<td>579</td>
</tr>
<tr>
<td>Average runtime</td>
<td>1590</td>
<td>1067</td>
<td>1031</td>
<td>990</td>
<td>913</td>
<td>844</td>
<td>840</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Number Devices:</th>
<th>$0^1$</th>
<th>$0^2$</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Run:1</td>
<td>3177</td>
<td>2160</td>
<td>2389</td>
<td>1473</td>
<td>1498</td>
<td>1259</td>
<td>1195</td>
</tr>
<tr>
<td></td>
<td>3147</td>
<td>2165</td>
<td>2408</td>
<td>1448</td>
<td>1325</td>
<td>1163</td>
<td>1053</td>
</tr>
<tr>
<td></td>
<td>3179</td>
<td>2059</td>
<td>2541</td>
<td>1669</td>
<td>1089</td>
<td>1205</td>
<td>1101</td>
</tr>
<tr>
<td></td>
<td>3159</td>
<td>2149</td>
<td>2458</td>
<td>1511</td>
<td>1030</td>
<td>1205</td>
<td>1109</td>
</tr>
<tr>
<td></td>
<td>3165</td>
<td>2163</td>
<td>2456</td>
<td>1510</td>
<td>1097</td>
<td>1280</td>
<td>947</td>
</tr>
<tr>
<td></td>
<td>3163</td>
<td>2137</td>
<td>2218</td>
<td>1734</td>
<td>940</td>
<td>995</td>
<td>951</td>
</tr>
<tr>
<td></td>
<td>3170</td>
<td>2179</td>
<td>2347</td>
<td>1160</td>
<td>870</td>
<td>1005</td>
<td>997</td>
</tr>
<tr>
<td>Average runtime</td>
<td>3166</td>
<td>2145</td>
<td>2402</td>
<td>1510</td>
<td>1113</td>
<td>1159</td>
<td>1050</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Number Devices:</th>
<th>$0^2$</th>
<th>$0^1$</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Run:1</td>
<td>6421</td>
<td>6573</td>
<td>7358</td>
<td>3843</td>
<td>2753</td>
<td>2181</td>
<td>1760</td>
</tr>
<tr>
<td></td>
<td>6445</td>
<td>6556</td>
<td>7253</td>
<td>3829</td>
<td>2729</td>
<td>1929</td>
<td>1793</td>
</tr>
<tr>
<td></td>
<td>6268</td>
<td>6422</td>
<td>7104</td>
<td>3865</td>
<td>2711</td>
<td>1960</td>
<td>1621</td>
</tr>
<tr>
<td></td>
<td>6269</td>
<td>6441</td>
<td>7142</td>
<td>3799</td>
<td>2752</td>
<td>1989</td>
<td>1681</td>
</tr>
<tr>
<td></td>
<td>6269</td>
<td>6462</td>
<td>7137</td>
<td>3681</td>
<td>2604</td>
<td>2039</td>
<td>1566</td>
</tr>
<tr>
<td>Average runtime</td>
<td>6334</td>
<td>6488</td>
<td>7199</td>
<td>3803</td>
<td>2710</td>
<td>2020</td>
<td>1685</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Number Devices:</th>
<th>$0^2$</th>
<th>$0^1$</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Run:1</td>
<td>9605</td>
<td>10305</td>
<td>6845</td>
<td>5527</td>
<td>3800</td>
<td>3077</td>
<td>2569</td>
</tr>
<tr>
<td></td>
<td>9475</td>
<td>9628</td>
<td>6866</td>
<td>5516</td>
<td>3705</td>
<td>3001</td>
<td>2504</td>
</tr>
<tr>
<td></td>
<td>9491</td>
<td>9913</td>
<td>6849</td>
<td>5524</td>
<td>3734</td>
<td>2840</td>
<td>2423</td>
</tr>
<tr>
<td></td>
<td>9513</td>
<td>9682</td>
<td>6770</td>
<td>5561</td>
<td>3737</td>
<td>2895</td>
<td>2382</td>
</tr>
<tr>
<td></td>
<td>9474</td>
<td>9695</td>
<td>6652</td>
<td>5472</td>
<td>3674</td>
<td>2849</td>
<td>2262</td>
</tr>
<tr>
<td>Average runtime</td>
<td>9512</td>
<td>9845</td>
<td>6796</td>
<td>5520</td>
<td>3730</td>
<td>2932</td>
<td>2428</td>
</tr>
<tr>
<td>Number Devices:</td>
<td>0^2</td>
<td>0^4</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>----------------</td>
<td>-----</td>
<td>-----</td>
<td>-----</td>
<td>-----</td>
<td>-----</td>
<td>-----</td>
<td>-----</td>
</tr>
<tr>
<td>Run:1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>12848</td>
<td>13097</td>
<td>8924</td>
<td>4625</td>
<td>4936</td>
<td>3722</td>
<td>3160</td>
</tr>
<tr>
<td>3</td>
<td>12756</td>
<td>12843</td>
<td>6752</td>
<td>4751</td>
<td>4936</td>
<td>3722</td>
<td>3160</td>
</tr>
<tr>
<td>4</td>
<td>12786</td>
<td>12960</td>
<td>8918</td>
<td>5118</td>
<td>3747</td>
<td>3031</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>12776</td>
<td>12846</td>
<td>8910</td>
<td>5719</td>
<td>4903</td>
<td>3723</td>
<td>3025</td>
</tr>
<tr>
<td>Average runtime:</td>
<td>12669</td>
<td>12930</td>
<td>8504</td>
<td>4753</td>
<td>4990</td>
<td>3749</td>
<td>3407</td>
</tr>
</tbody>
</table>
.2 Appendix B - Code Appendix

.2.1 ClusterManager Framework

ClusterManagerService class

```java
package com.honsproj.ian.clustermanager;

import com.honsproj.ian.clustermanager.Adapter.DeviceListAdapter;
import com.honsproj.ian.clustermanager.Callable.CallableManagerService;
import com.honsproj.ian.clustermanager.Network.NetworkManagerService;

public class ClusterManagerService extends Service {

    private ServiceHandler mServiceHandler;
    private Handler mHandler;
    private Looper mServiceLooper;

    private static NetworkManagerService netManagerService;
    private static CallableManagerService callManagerService;
    boolean netManagerBound, callManagerBound, isNetCoordinator = false;

    private ArrayList<WifiP2pDevice> availableDeviceArrayList;

    //Binder passed to clients
    private final IBinder clusterServiceBinder = new LocalBinder();

    /**
     * Class used for the client Binder. Because we know this service always
     * runs in the same process as its clients, we don't need to deal with IPC.
     */
```
public class LocalBinder extends Binder {
    public ClusterManagerService getService() {
        // Return this instance of LocalService so clients can call public methods
        return ClusterManagerService.this;
    }
}

public ClusterManagerService() {
}

public void discoverPeers(Activity discoveringActivity) {
    netManagerService.discoverPeers(discoveringActivity);
}

@Override
public int onStartCommand(Intent intent, int flags, int startId) {
    Toast.makeText(this, "Cluster Manager service starting", Toast.LENGTH_SHORT).show();
    Log.i("Cluster Manager", "onStartCommandIssued");
    // For each start request, send a message to start a job and deliver the
    // start ID so we know which request we’re stopping when we finish the job
    Message msg = mServiceHandler.obtainMessage();
    msg.arg1 = startId;
    mServiceHandler.sendMessage(msg);
    // If we get killed, after returning from here, restart
    return START_STICKY;
}

// Receives messages from the thread
private final class ServiceHandler extends Handler {
    public ServiceHandler(Looper looper) {
        super(looper);
    }
}

@Override
public void handleMessage(Message msg) {
}
@Override
public void onCreate() {
    super.onCreate();

    HandlerThread thread = new HandlerThread("Thread\_name", android.os.Process.THREAD_PRIORITY_BACKGROUND);
    thread.start();

    mServiceLooper = thread.getLooper();
    mServiceHandler = new ServiceHandler(mServiceLooper);

    Intent netManagerIntent = new Intent(this, NetworkManagerService.class);
    bindService(netManagerIntent, netManServiceConnection, Context.BIND_AUTO_CREATE);

    Intent callManagerIntent = new Intent(this, CallableManagerService.class);
    bindService(callManagerIntent, callManServiceConnection, Context.BIND_AUTO_CREATE);

    Log.i(ClusterManagerService.this.toString(), "Service\_created");
}

@Override
public void onDestroy() {
    Toast.makeText(this, "service \_done, \_ClusterManagerService\_destroyed", Toast.LENGTH_SHORT).show();
    unbindService(netManServiceConnection);
    unbindService(callManServiceConnection);

    Log.i(ClusterManagerService.this.toString(), "Service\_Destroyed");
}
@Override
public IBinder onBind(Intent intent) {
    return clusterServiceBinder;
}

private ServiceConnection netManServiceConnection = new ServiceConnection() {
    @Override
    public void onServiceConnected(ComponentName name, IBinder service) {
        //Bound to a local service
        NetworkManagerService.LocalBinder binder = (NetworkManagerService.LocalBinder) service;
        netManagerService = binder.getService();
        netManagerBound = true;
    }
    @Override
    public void onServiceDisconnected(ComponentName name) {
        netManagerService = null;
        netManagerBound = false;
    }
};

private ServiceConnection callManServiceConnection = new ServiceConnection() {
    @Override
    public void onServiceConnected(ComponentName name, IBinder service) {
        CallableManagerService.LocalBinder binder = (CallableManagerService.LocalBinder) service;
        callManagerService = binder.getService();
        callManagerBound = true;
    }
    @Override
    public void onServiceDisconnected(ComponentName name) {
        callManagerService = null;
        callManagerBound = false;
    }
};
public DeviceListAdapter
    getNetworkDeviceListAdapter()
    {
        return netManagerService.getDeviceListAdapter();
    }

public void joinNetwork(WifiP2pDevice device)
    {
        // Transmit a message to the device connected to to indicate ID & availability
        if (device.isGroupOwner() == true)
            {
                netManagerService.joinNetwork(device);
                callManagerService.setIsClient(true);
            }
    }

public WifiP2pDevice getNetworkDevice(int index)
    {
        return netManagerService.getDevice(index);
    }

public ArrayList<WifiP2pDevice>
    getAvailableDeviceArray()
    {
        availableDeviceArrayList = netManagerService.getAvailableDeviceArray();
        return availableDeviceArrayList;
    }

public void createNetwork()
    {
        Log.i("CLUSTERMANAGER", "Network Creation Launched");
        netManagerService.createNetwork();
        // Created a network so the running device is Group Owner and should act as network co–ordinator
        callManagerService.setIsClient(false);
    }

public void send(Callable c, int id)
    {
        Log.i("Send Task", "attempting to process callable");
        netManagerService.sendTask(c, id);
        Log.i("Send task", "Task finished");
    }

public Object receive(int id)
// Returns the result of a computation performed using the distributed system
return netManagerService.getResult(id);

public int numberAvailableDevices()
{
return netManagerService.numberAvailableDevices();
}

public static Future sendToCallManager(Callable c)
{
return callManagerService.submit(c);
}
NetworkManagerService Class

```java
package com.honsproj.ian.clustermanager.Network;

import com.honsproj.ian.clustermanager.Adapter.DeviceListAdapter;
import com.honsproj.ian.clustermanager.ClusterManagerService;

public class NetworkManagerService extends Service {

    private static int SERVER_NEGOTIATION_PORT = 1254;
    private static int TASK_COMMUNICATION_PORT = 1255;
    private static int RESULT_COMMUNICATION_PORT = 1256;

    private WifiP2pManager mManager;
    private WifiP2pManager.Channel mChannel;
    private WifiBroadcastReceiver mReceiver;
    private IntentFilter WifiIntentFilter;
    private WifiPeerListListener mPeerListListener;
    private DeviceListAdapter deviceListAdapter;
    private WifiP2pDeviceList WifiDeviceList;

    private static String host;
    private static ArrayList<WifiP2pDevice> availableDeviceList;
    private static ArrayList<String> busyDeviceList = new ArrayList<String>();
    private static ArrayList<String> clientList = new ArrayList<String>();
    private static Map<String, Integer> deviceToIdMap = new HashMap<>();
    private static Map<Integer, Object> idToReturnObjectMap = new HashMap<>();

    private static ReceiveResultThread resultThread;
    private static boolean offerClient = false;
```
private static boolean isClient = false;
private static boolean isHost = false;

public NetworkManagerService() {
    super();
}

// Binder passed to clients
private final IBinder serviceBinder = new LocalBinder();
/**
 * Class used for the client Binder. Because we know this service always
 * runs in the same process as its clients, we don't need to deal with IPC.
 */
public class LocalBinder extends Binder {
    public NetworkManagerService getService() {
        // Return this instance of LocalService so parent manager can call public methods
        return NetworkManagerService.this;
    }
}

@Override
public void onCreate(){
    mManager = (WifiP2pManager) getSystemService(Context.WIFI_P2P_SERVICE);
    mChannel = mManager.initialize(this, getMainLooper(), null);
    mPeerListListener = new WifiPeerListListener(
            NetworkManagerService.this);
    mReceiver = new WifiBroadcastReceiver(mManager,
            mChannel, this, mPeerListListener);

    availableDeviceList = new ArrayList<WifiP2pDevice>();
    deviceListAdapter = new DeviceListAdapter(
            getBaseContext(), availableDeviceList);

    WifiIntentFilter = new IntentFilter();
    WifiIntentFilter.addAction(WifiP2pManager.
            WIFI_P2P_STATE_CHANGED_ACTION);
WifiIntentFilter.addAction(WifiP2pManager.
    WIFI_P2P_PEERS_CHANGED_ACTION);
WifiIntentFilter.addAction(WifiP2pManager.
    WIFI_P2P_CONNECTION_CHANGED_ACTION);
registerReceiver(mReceiver, WifiIntentFilter);

@Override
public int onStartCommand(Intent intent, int flags,
    int startId) {
    return super.onStartCommand(intent, flags, startId);
}

public WifiP2pManager.ConnectionInfoListener
    connectionListener = new WifiP2pManager.
    ConnectionInfoListener() {
    @Override
    public void onConnectionInfoAvailable(
        WifiP2pInfo info) {
        if (info.groupFormed && info.isGroupOwner)
        {
            // This device will be the host,
            // sending out tasks to be computer
            // Run Server to accept clients
            Log.i("Group info", info.toString());
            ServerThread serverThread = new
                ServerThread();
            serverThread.start();
            resultThread = new
                ReceiveResultThread();
        }
        else if (info.groupFormed && info.
            groupOwnerAddress != null &&
            offerClient){
            // This device will act as
            // a client, donating processor time to
            // the host
            Log.i("Group formed", "connected as
                client");
            host = info.groupOwnerAddress.
                getHostAddress();
            // Make a connection to the host and
            // transmit client IP for future
            // addressing
            ClientRegistrationThread
        }
clientRegisterThread = new ClientRegistrationThread();
clientRegisterThread.start();
isClient = true;
//clientRegisterThread.join();
try {
    while (info.groupFormed == true) {
        Log.i("Start ReceiverThread:", "Still connected to host");
        //Once host has received address listen for incoming tasks
        ComputationThread
            computeThread = new ComputationThread();
            computeThread.start();
            computeThread.join();
    }
} catch (InterruptedException e) {
    Log.i("receive thread ERROR", e.toString());
}
else {
    Log.i("Error Occurred", "Device is neither client nor host");
    Log.i("Conditions", info.toString() +" " + String.valueOf(offerClient));
}
}

public void discoverPeers(Activity discoveringActivity) {
    Log.i("Discovering peers", "Attemping to find devices");
    mManager.discoverPeers(mChannel, new WifiP2pManager.ActionListener() {
        @Override
        public void onSuccess() {
            Log.i("Peer discovery", "success");
            Toast.makeText(getBaseContext(), "Discovery Initiated", Toast.LENGTH_SHORT).show();
        }
    });
public void onFailure(int reasonCode) {
    Log.i(“Peer discovery”, ”Failure”);
    Toast.makeText(getBaseContext(), ”Discovery failed” + reasonCode, Toast.LENGTH_SHORT).show();
}
}

public void setDeviceList(Collection&lt;WifiP2pDevice &gt; devList) {
    availableDeviceList.clear();
    availableDeviceList.addAll(devList);
    deviceListAdapter.notifyDataSetChanged();
}

public ArrayList&lt;WifiP2pDevice&gt; getAvailableDeviceList() {
    return availableDeviceList;
}

public WifiP2pDeviceList getPeers() {
    Log.i(”getPeers called”, ”Returning list of available peers”);
    // Returns list of nearby devices that are available via Wi-Fi Direct
    mManager.requestPeers(mChannel, mPeerListListener);
    mPeerListListener.onPeersAvailable(WifiDeviceList);
    return WifiDeviceList;
}

public void joinNetwork(WifiP2pDevice device) {
    offerClient = true;
    /*
     * Provides the resource to join an existing network in order to donate device computation time to a foreign device
     *
     */
    WifiP2pConfig config = new WifiP2pConfig();
    config.deviceAddress = device.deviceAddress;
    config.groupOwnerIntent = 0;
mManager.connect(mChannel, config, new
WifiP2pManager.ActionListener() {
    @Override
    public void onSuccess() {
        Log.i("NetworkManager_Service", "Connect_Success");
    }
    @Override
    public void onFailure(int reason) {
        Log.i("NetworkManager_Service", "network_join_failed");
    }
});

public void createNetwork(){
    isHost = true;
    // We wish to use the current device as its own group owner,
    // inviting secondary devices to connect and contribute power
    mManager.createGroup(mChannel, new
        WifiP2pManager.ActionListener() {
            @Override
            public void onSuccess() {
                Log.i("NetworkManager", "Network_Created");
            }
            @Override
            public void onFailure(int reason) {
                Log.i("NetworkManager", "Network_creation_failed, Reason_code:" + String.valueOf(reason));
            }
        });
    }
}

@Override
public void onDestroy(){
    super.onDestroy();
    deviceListAdapter.clear();
    availableDeviceList.clear();
    unregisterReceiver(mReceiver);
    mPeerListListener = null;
    mChannel = null;
    mManager = null;
}
@Override
public IBinder onBind(Intent intent) {
    return serviceBinder;
}

public DeviceListAdapter getDeviceListAdapter() {
    return deviceListAdapter;
}

public WifiP2pDevice getDevice(int index) {
    return availableDeviceList.get(index);
}

public ArrayList&lt;WifiP2pDevice&gt;
getAvailableDeviceArray() {
    return availableDeviceList;
}

public void sendTask(Callable c, int id) {
    Log.i("NetworkTransferService", "Sending task");
    if (availableDeviceList.size() &gt; 0) {
        Log.i("NetworkTransferService", "Device available for task to be sent");
        String client = getClient(); // Pop client address from the list of connected devices
        busyDeviceList.add(client); // Push client address on to the list of devices currently working
        deviceToIdMap.put(client, id); // Map the computation ID to the client address
        SendTaskThread sendTaskThread = new SendTaskThread(c, client);
        sendTaskThread.start();
        if (resultThread.isAlive()) {
        }
        else if (resultThread.getState() == Thread.State.NEW) {
            resultThread.start();
        }
    }
}
else { // If there are no available clients for computation then we must execute the callables on the host device
    Log.i("Local Execution", "No client devices connected to receive callable, executing locally");
    Object returnedObject = null;
    try{
        returnedObject = c.call();
    }catch(Exception e){
        Log.i("Local Execution Failed", e.toString());
    }
    // Put the execution result in the same location as if it had been executed remotely
    idToReturnObjectMap.put(id, returnedObject);
}
}

public int numberAvailableDevices() { return clientList.size(); }

public static String getHost() { return host; }

public static void addClient(String client) {
    clientList.add(client);
    if(busyDeviceList.contains(client)){
        busyDeviceList.remove(client);
    }
}

public String getClient() {
    // Pulls the first client that is connected from the list,
    // Removes client to ensure that they are not called more than once at a time
    String client = null;
    if(clientList.isEmpty() == false){
        client = clientList.get(0);
    }
clientList.remove(0);

return client;

public static boolean getClientStatus()
{
    return isClient;
}

public static boolean getHostStatus()
{
    return isHost;
}

public Object receiveResult(int id){
    if(idToReturnObjectMap.containsKey(id)) {
        Log.i("KEY_FOUND", "SUCCESS");
        Object returnObject = idToReturnObjectMap.get(id);
        idToReturnObjectMap.remove(id);
        return returnObject;
    }
    return null;
}

public Object waitReceiveResult(int id) {
    while (!idToReturnObjectMap.containsKey(id)) {
    }
    Object returnObject = idToReturnObjectMap.get(id);
    idToReturnObjectMap.remove(id);

    return returnObject;
}

public Object waitReceiveResult(int id, long maxTime) {
    int s = 1;
    while (!idToReturnObjectMap.containsKey(id)) {
        try {
            if(s==4){ return null; }
            wait(maxTime/4);
            s++;
        } catch(InterruptedException e){}
    }
    Object returnObject = idToReturnObjectMap.get(id);
    idToReturnObjectMap.remove(id);

    return returnObject;
}
private static String getMyDeviceIP() {
    // Scans through active network devices, returns the active local network address
    StringBuilder IFCONFIG = new StringBuilder();
    try {
        for (Enumeration<NetworkInterface> en = NetworkInterface.getNetworkInterfaces(); en.hasMoreElements();)
            NetworkInterface intf = en.nextElement();
        for (Enumeration<InetAddress> enumIpAddr = intf.getInetAddresses(); enumIpAddr.hasMoreElements();)
            InetAddress inetAddress = enumIpAddr.nextElement();
        if (!inetAddress.isLoopbackAddress() && !inetAddress.isLinkLocalAddress() && inetAddress.isSiteLocalAddress())
            if (inetAddress.getHostAddress().trim().matches("192.168.49.\[0-2][0-9]\[0-9]\[0-9]\[0-9]\[0-2][0-9]?\[0-9]?\[0-9]?\[0-9]?\"))
                IFCONFIG.append(inetAddress.getHostAddress().trim());
    }
    catch (SocketException e) { Log.i("NetManager, getIPERROR", e.toString()); }
    return IFCONFIG.toString();
}

private static class ComputationThread extends Thread {
    public void run() {
        Log.i("Receiver Thread", "receiver thread started");
        try {
            id; idToReturnObjectMap.remove(id);
            return returnObject;
        }
    }
}
boolean resultAccepted = false;
host = NetworkManagerService.getHost();
String myDeviceIP =
    NetworkManagerService.getMyDeviceIP();
// Create a socket and wait for the host to attempt a connection
Log.i("RECEIVER_THREAD", "waiting for task");
ServerSocket serverSocket = new ServerSocket(
    TASK_COMMUNICATION_PORT);
serverSocket.setReuseAddress(true);
Socket socket = serverSocket.accept();
socket.setReuseAddress(true);

// Once connected create a stream that will accept an incoming callable object
Log.i("Receiver thread", "Socket connected. ready to receive callable");
InputStream inStream = socket.getInputStream();
    getInputStream();
ObjectInputStream objInStream = new ObjectInputStream(inStream);
Log.i("Computation Thread", "Callable ready to be read");
Callable c = (Callable) objInStream.readObject();
// c needs to be sent to the Cluster Manager
Log.i("Callable Received:", c.toString());

Object sampleResult =
    ClusterManagerService.
    sendToCallManager(c).get();
Log.i("COMPUTATION_RESULT",
    sampleResult.toString());
socket.close();
objInStream.close();
inStream.close();
serverSocket.close();

while (resultAccepted==false){
    try {
        Log.i("Sending back result", "Ready to send");
        Socket resultSocket = new Socket();
        resultSocket.setReuseAddress(true);
        resultSocket.bind(null);
        Log.i("Sending back result", "Ready to send");
        resultSocket.connect((new InetSocketAddress(host,
            RESULT_COMMUNICATION_PORT)));

        Log.i("Sending Result", "Socket connected");
        Log.i("Result: ", "attempting to write data");
        //Once object has finished being executed, return the result to the host device
        OutputStream outStream = resultSocket.getOutputStream();
        ObjectOutputStream objOutStream = new ObjectOutputStream(outStream);
        objOutStream.writeObject(myDeviceIP); //TODO: Consider revising feature with MAC (Easier to get at host, less socket writing to do)
        Log.i("Sending Result", "");
My_IP_written!

//Now that the host knows
who we are, send the
result that the host
desires
objOutStream.writeObject(
sampleResult);

InputStream
approvedInStream =
resultSocket.
getInputStream();
ObjectInputStream
approvedObjInStream =
new ObjectInputStream(
approvedInStream);

resultAccepted = true;
Log.i("end_of_send",
String.valueOf(
resultAccepted));
Log.i("Sending_Result", "
Result_written!");
objOutStream.close();
outStream.close();
}
catch (SocketException p) {
resultAccepted = false;
Log.i(""
SocketExceptionThrown",
p.toString());
}

Log.i("Recieve_Thread:", "Result_
returned_to_host_device");
catch(SocketException e){
Log.i("Receive_SocketERROR\l", e.
toString());
}
catch(IOException e){

private static class ReceiveResultThread extends Thread {
    public void run() {
        try {
            while (true) { //TODO: Change to while
                busy devices > 0
                Log.i("Results\_Receive\_Thread", ""
                        Starting\_SOcket\_process");
                //Listen for a client returning
                the result of a computation
                ServerSocket serverSocket = new
                        ServerSocket(
                        RESULT\_COMMUNICATION\_PORT);
                serverSocket.setReuseAddress(true);
                Socket socket = serverSocket.
                        accept();
                socket.setReuseAddress(true);
                Log.i("Results\_Receive\_Thread", ""
                        Socket\_accepted");

                //Receive client IP Address so we
                can match the input to what was
                sent
                InputStream inStream = socket.
                        getInputStream();
                ObjectInputStream objInStream =
                        new ObjectInputStream(inStream)
                        ;

                String clientInetAddress = (String
                        ) objInStream.readObject();
            }
        } catch (ClassNotFoundException e) {
            Log.i("Receive\_Thread\_ERROR\_3", e.
                        toString());
        } catch (Exception e) {
            Log.i("Receive\_Thread\_ERROR\_4", e.
                        toString());
        }
    }
}
// Store the client address to give a method of addressing results
int taskID = deviceToIdMap.get(clientInetAddress);

// Now we know who we are talking to, get the result of their computation
Object returnedObject = (Object) objInStream.readObject();

idToReturnObjectMap.put(taskID, returnedObject);

// Add the device back into the computation pool and remove reference to previous task
NetworkManagerService.addClient(clientInetAddress);
deviceToIdMap.remove(clientInetAddress);

Log.i("FINISHED RECEIVE OBJECT", clientInetAddress);
Log.i("OBJECT RECEIVED:", returnedObject.toString());

objInStream.close();
inStream.close();
serverSocket.close();
socket.close();

} catch (IOException e){ Log.i("Error Receiving Results", e.toString());
} catch (ClassNotFoundException e){Log.i("Error Receiving Results", e.toString());
}

private static class SendTaskThread extends Thread {

// Creates a thread that will manage sending a callable to a client device and
private String candidateClientAddress;
Callable callable;

public SendTaskThread(Callable c, String clientAddress) {
    candidateClientAddress = clientAddress;
    callable = c;
}

public void run() {
    int port = TASK COMMUNICATION PORT;
    try {
        if (candidateClientAddress == null) {
            //No candidates for computation
        } else {
            //Set up a socket connection with waiting client
            Log.i("Sending to:",
                    candidateClientAddress + ",");
            Socket socket = new Socket();
            socket.setReuseAddress(true);
            socket.bind(null);
            InetAddress clientAddress =
                    InetAddress.getByName(
                            candidateClientAddress);
            socket.connect((new
                            InetSocketAddress(clientAddress,
                                    port)));

            //Once connected create an output stream and write the callable object to it
            Log.i("SendTaskThread", "Socket connected, task ready to be sent");
            OutputStream outStream = socket.getOutputStream();
            ObjectOutputStream objOutStream =
                    new ObjectOutputStream(outStream);
            objOutStream.writeObject(callable);
Log.i("Send\_Task\_Thread", "Object sent:" + callable.toString());

objOutStream.close();
outStream.close();
socket.close();
}
}
}
}

private static class ServerThread extends Thread{
    // Creates a thread that continuously listens on the host device for client devices wishing to connect
    public void run(){
        try {
            while(true) { // TODO: Change to conditional
                // Create a socket and wait for a client to attempt to connect to it
                Log.i("Server\_socket", "Group formed, waiting for client connection");
                ServerSocket serverSocket = new ServerSocket(
                    SERVER\_NEGOTIATION\_PORT);
                serverSocket.setReuseAddress(true);

                Socket socket = serverSocket.accept();
                socket.setReuseAddress(true);
                Log.i("Server\_socket", "Socket accepted");

                // Receive and store client credentials
                InputStream inStream = socket.getInputStream();
                getInputStream();
                ObjectInputStream objInStream =
                    new ObjectInputStream(inStream)
String clientInetAddress = (String) objInStream.readObject();
Log.i("address received:",
    clientInetAddress);

clientInetAddress =
    clientInetAddress.trim();
NetworkManagerService.addClient(
    clientInetAddress);

objInStream.close();
inStream.close();
socket.close();
serverSocket.close();
}
}catch(IOException e){
    Log.i("ConnectionInfoAvailable", e.toString());
}
}catch(ClassNotFoundException e){
    Log.i("error reading input", e.toString());
}

private static class ClientRegistrationThread
    extends Thread{

    // Connects to a running host and transfers IP address that can be used for future connections

    public void run(){
        host = NetworkManagerService.getHost();
        Log.v("PrintLN", "Group Owner Address: "+
            host);

        try{

            // Connect to the waiting host server socket
            Log.i("Attempting to connect", "
                Connecting to" + host.toString());
            Socket socket = new Socket();
            socket.setReuseAddress(true);
            socket.bind(null);
            socket.connect((new InetSocketAddress(
host, SERVER_NEGOTIATION_PORT));
Log.i("Connection Success", "Connected to host");

// Send inet address credentials for successful communication
OutputStream outStream = socket.getOutputStream();
ObjectOutputStream objOutStream = new ObjectOutputStream(outStream);
String deviceIP = NetworkManagerService.getMyDeviceIP();

Log.i("Writing address: ", deviceIP);
objOutStream.writeObject(deviceIP);
objOutStream.close();
outStream.close();
Log.i("Finished writing", "closing streams and socket");
socket.close();
}
catch (IOException e) { Log.i("ClientRegThread ERROR", e.toString());
}
}
WiFi BroadcastReceiver Class

package com.honsproj.ian.clustermanager.Network.Utilities;

import com.honsproj.ian.clustermanager.Network.NetworkManagerService;

public class WifiBroadcastReceiver extends BroadcastReceiver {
    private WifiP2pManager mManager;
    private WifiP2pManager.Channel mChannel;
    private NetworkManagerService mService;
    private WifiPeerListListener mPeerListListener;
    private InetAddress gOAddress;
    private WifiP2pInfo info;

    public WifiBroadcastReceiver(WifiP2pManager manager, WifiP2pManager.Channel channel, NetworkManagerService service, WifiPeerListListener peerListListener) {
        super();
        this.mManager = manager;
        this.mChannel = channel;
        this.mService = service;
        this.mPeerListListener = peerListListener;
    }

    @Override
    public void onReceive(Context context, Intent intent) {
        String action = intent.getAction();
        if (WifiP2pManager.WIFI_P2P_STATE_CHANGED_ACTION.equals(action)) {
            Log.d("WIFIBROADCAST\onReceive", "P2PSTATECHANGEDACTION");
            // Check to see if Wi-Fi is enabled and notify appropriate activity
            int state = intent.getIntExtra(WifiP2pManager.EXTRA_WIFI_STATE, -1);
        }
    }
}
if (state == WifiP2pManager.
WIFI_P2P_STATE_ENABLED) {
    // Wifi P2P is enabled
    Log.i("WIFI_P2P State", "WiFi_P2P Enabled");
} else {
    // Wi-Fi P2P is not enabled
    Log.i("WIFI_P2P State", "WiFi_P2P disabled");
}
else if (WifiP2pManager.
WIFI_P2P_PEERS_CHANGED_ACTION.equals(action)) {
    Log.i("WIFIP2P Peers Changed", "PEERS_CHANGED_ACTION");
    // Call WifiP2pManager.requestPeers() to get a list of current peers
    if(mManager != null){
        mManager.requestPeers(mChannel, mPeerListListener);
    }else{
        Log.i("mMananger ERROR", "mManager==null");
    }
}
else if (WifiP2pManager.
WIFI_P2P_CONNECTION_CHANGED_ACTION.equals(action)) {
    Log.i("WIFIBROADCAST onReceive", "CONNECTION_CHANGED_ACTION");
    // Respond to new connection or disconnections
    if (mManager == null) {
        Log.i("WiFiBroadcastReceiver", "mManager==null");
        return;
    }
    WifiP2pInfo info = (WifiP2pInfo) intent.getParcelableExtra(WifiP2pManager.
EXTRA_WIFI_P2P_INFO);
    Log.i("BROADCAST_REC WifiInfo", info.toString());
    NetworkInfo networkInfo = (NetworkInfo) intent.getParcelableExtra(
WifiP2pManager.EXTRA_NETWORK_INFO);
Log.i("BROADCAST_REC.netInfo", networkInfo.toString());

if(networkInfo.isConnected()){
    Log.i("networkInfo.isCONNECTED", networkInfo.toString());
    mManager.requestConnectionInfo(
        mChannel, mService.
        connectionListener);
}
}
WiFi peerListListener Class

```java
package com.honsproj.ian.clustermanager.Network.
    Utilities;

    NetworkManagerService;

/**
 * Created by ian on 14/01/15.
 */
public class WifiPeerListListener implements WifiP2pManager.PeerListListener{

    NetworkManagerService netManService;

    public WifiPeerListListener(NetworkManagerService service){
        this.netManService = service;
    }

    @Override
    public void onPeersAvailable(WifiP2pDeviceList peers) {
        netManService.setDeviceList(new ArrayList<
            WifiP2pDevice>(peers.getDeviceList()));
        if (netManService.getAvailableDeviceList().
            size() == 0) {
            Log.d("OnPeersAvailable()", "No _devices _found");
            return;
        } else{
            netManService.setDeviceList(peers.
                getDeviceList());
        }
    }
}
```
package com.honsproj.ian.clustermanager.Callable;

public class CallableManagerService extends Service {
    /*
     * In this class we need to check the status of the
     * ClusterExecutorServices running and feed more
     * tasks when available.
     * We should report hunger and fullness to the
     * ClusterManagerService which may co-ordinate
     * with other devices services.
     */

    private final IBinder serviceBinder = new LocalBinder();
    private ExecutorService executorService;
    private boolean busyLock = false;
    private boolean isClient = false;

    public CallableManagerService() {}

    @Override
    public void onCreate() {
        Log.i(CallableManagerService.this.toString(),
                "Service created");
        executorService = Executors.newCachedThreadPool();
    }

    public CallableManagerService getService() {
        // Return this instance of LocalService so
        // clients can call public methods
        return CallableManagerService.this;
    }

    public class LocalBinder extends Binder {
        public CallableManagerService getService() {
            // Return this instance of LocalService so
            // clients can call public methods
            return CallableManagerService.this;
        }
    }
}
@Override
public int onStartCommand(Intent intent, int flags, int startId) {
    super.onStartCommand(intent, flags, startId);
    Log.i(CallableManagerService.this.toString(), "Service started");
    return super.onStartCommand(intent, flags, startId);
}

public void setIsClient(boolean clientStatus) {
    isClient = clientStatus;
}

public Future submit(Callable c) {
    Log.i("CallManagerService: ", c.toString());
    if (isClient == true) {
        // If I am a client, then I am the one that should be doing the work, and will execute the task as appropriate
        busyLock = true;
        // check if executor service busy. If not:
        Future callableFuture = executorService.
        submit(c);
        return callableFuture;
    } else {
        Log.i("CallManagerService", "executorService was busy");
        return null;
    }
}

@Override
public IBinder onBind(Intent intent) { return serviceBinder; }
}
.2.2 Test Modules

PiCalculator Class

package com.honsproj.ian.clustermanager.TestDriver;

import android.util.Log;

import com.honsproj.ian.clustermanager.ClusterManagerService;

public class PiJavaThread implements Serializable {
    private static final long serialVersionUID = 4841206184480558287L;

    int n, numThreads;
    double pi = 0.0;
    List<Integer> notReceived = new ArrayList<Integer>();

    public PiJavaThread(int nd, MainActivity activity) {
        ClusterManagerService clusterManager = activity.getClusterManagerService();
        n = nd;
        Log.i("PiJavaThread", "Called java Pi Calculator");
        numThreads = clusterManager.
                numberAvailableDevices();
        Double sum[] = new Double[numThreads];
        for (int i=0; i<numThreads; i++) {
            Log.i("Worker called", "Worker: " + String.valueOf(i));
            Callable dev = new PIworker(i);
            Log.i("Num_THREADS", String.valueOf(numThreads));
            // Call send on the callable, giving an ID that will be used to retrieve the result
            activity.send(dev, i);
            notReceived.add(i, i);
        }
        int r = 0;
        int p = notReceived.size();
    }
while (notReceived.isEmpty()==false) {
    for (int i = 0; i < p; i++) {
        if (notReceived.contains(i)) {
            Object result = activity.receive(i);
            if (result == null) {
            } else {
                Object o = (Object) i;
                notReceived.remove(o);
                String res = String.valueOf(result);
                sum[r] = Double.parseDouble(res);
                r++;
            }
        }
    }
}
for (int i=0; i<sum.length; i++) {
    if (sum[i]==null) sum[i]=0.0;
    pi = pi + sum[i];
}
Log.i("PI\approx", String.valueOf(pi));
notReceived.clear();

class PIworker implements Callable, Serializable {
    int myid;
    public PIworker(int id) {
        myid = id;
    }
    public Object call() {
        Log.i("WORKER CALLED", String.valueOf(myid));
        double d, s, x;
        d = 1.0/n;
        s = 0.0;
        for (int i=myid+1; i<=n; i+=numThreads) {
            x = (i-0.5)*d;
            s += 4.0/(1.0+x*x);
        }
        return (d*s);
    }
}
PiCalculator Class

package com.honsproj.ian.clustermanager.TestDriver;

import com.honsproj.ian.clustermanager.ClusterManagerService;
    Utilities.DeviceClickListener;
import com.honsproj.ian.clustermanager.R;

public class MainActivity extends Activity implements View.OnClickListener{

    /*
     * Test activity that drives the cluster manager framework.
     * NO management code should be placed here. To be replaced by functional
     * programs.
     */
    private ClusterManagerService clusterManagerService;
    private boolean clusterManagerBound = false;
    private ArrayList<WifiP2pDevice> availableDeviceList;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        //TEMP ROWS TO ALLOW DEBUGGING OF NETWORK FEATURE IF NETWORKING IS DONE ON THE UI THREAD.
        //StrictMode.ThreadPolicy policy = new
        //StrictMode.ThreadPolicy.Builder().permitAll().build();
        //StrictMode.setThreadPolicy(policy);

        Intent clusterManagerIntent = new Intent(this,
            ClusterManagerService.class);
        bindService(clusterManagerIntent,
            clusterManagerServiceConnection, Context.


BIND_AUTO_CREATE);
startService(new Intent(this,
          ClusterManagerService.class));

Button discoverBtn = (Button) findViewById(R.
id.discoverButton);
discoverBtn.setOnClickListener(this);

Button createNetworkBtn = (Button) findViewById(R.
id.create_network_btn);
createNetworkBtn.setOnClickListener(this);

Button adderBtn = (Button) findViewById(R.
id.adderButton);
adderBtn.setOnClickListener(this);

availableDeviceList = new ArrayList<
          WifiP2pDevice>();

private ServiceConnection
clusterManagerServiceConnection = new
ServiceConnection() {
    @Override
    public void onServiceConnected(ComponentName
name, IBinder service) {
        Log.i("Service Connection", "onServiceConnected");
        ClusterManagerService.LocalBinder binder =
          (ClusterManagerService.LocalBinder)
          service;
        clusterManagerService = binder.getService
();
        clusterManagerBound = true;
    }

    @Override
    public void onServiceDisconnected(
        ComponentName name) {
        clusterManagerBound = false;
    }
};

@Override
public void onClick(View v) {
    final View T = v;
    switch (v.getId()){
        case R.id.discoverButton:
            // Connect to an existant network
            Log.i(MainActivity.this.toString(), "Discover Button Pressed");
            clusterManagerService.discoverPeers(this);
            ListView deviceListView = (ListView)
                findViewById(R.id.deviceListView);
            deviceListView.setAdapter(
                clusterManagerService.getNetworkDeviceListAdapter());
            deviceListView.setOnItemClickListener(
                new DeviceClickListener(
                    MainActivity.this));
            Log.i("Discovery Finished", "end of peer discovery");
            break;
        case R.id.create_network_btn:
            Log.i("Create Network button", "creating autonomous network");
            clusterManagerService.createNetwork();
            // create a new network
            break;
        case R.id.adderButton:
            Context context = this;
            getApplicationContext();
            Log.i("Adder Button Pressed", "creating and Sending Added");
            long startTime = System.currentTimeMillis();
            PiJavaThread pi = new PiJavaThread
                (120000000, this);
            long endTime = System.currentTimeMillis();
            long totalTime = endTime - startTime;
            Log.i("Runtime analysis", "Runtime of PiCalc:" + String.valueOf(totalTime));
    }
break;
}

@Override
public void onDestroy() {
    super.onDestroy();
    unbindService(clusterManagerServiceConnection);
}

@Override
public boolean onCreateOptionsMenu(Menu menu) {
    // Inflate the menu; this adds items to the action bar if it is present.
    // getMenuInflater().inflate(R.menu.menu_main, menu);  // # Menu deactivated for present
    return true;
}

@Override
public boolean onOptionsItemSelected(MenuItem item) {
    // Handle action bar item clicks here.
    // The action bar will automatically handle clicks on the Home/Up button, so long
    // as you specify a parent activity in AndroidManifest.xml.
    int id = item.getItemId();
    // noinspection SimplifiableIfStatement
    /*
     * if (id == R.id.action_settings) {
     *     return true;
     * }
     */
    return super.onOptionsItemSelected(item);
}

public void send(Callable c, int id) {
    clusterManagerService.send(c, id);
}

public Object receive(int id) {

    return clusterManagerService.receive(id);
}

public WifiP2pDevice getDevice(int index){
    availableDeviceList = clusterManagerService.getAvailableDeviceArray();
    return availableDeviceList.get(index);
}

public ClusterManagerService getClusterManagerService(){
    return clusterManagerService;
}